UNIVERSITY OF
EDINBURGH

Edinburgh Regional Computing Centre

EMAS 2900: User's Guide

A GUIDE TO THE USE OF THE
EDINBURGH SUBSYSTEM OF THE
MULTI-ACCESS SYSTEM EMAS 2900

New Routine OPENSM

A new routine OPENSM has been written, which can be used instead of SMADDR. There is no intention to phase out SMADDR. The routine must be specified:

<u>externalroutinespec</u> OPENSM(integer CHAN, MODE, c
                                <u>integername</u> AD, LENGTH)

CHAN is used to specify the channel number (in the range 1-80) used for the DEFINE call for the file being used.

MODE can take one of the following values:

    0 connect file in any mode
    1 connect file for reading only
    2 connect file for reading and writing

AD   returns the address of the start of data (exactly the same as the
     <u>result</u> of a call of SMADDR) or zero if the file cannot be connected
     <u>in the</u> requested mode.

LEN returns the length of the data in the file in bytes, if the
     connection was successful (as with SMADDR)

Notes

     This routine will be useful in several situations e.g.,

     * When a user does not want to alter the contents of a file
        protection is provided by specifying read mode.

     * When a user wants to write to a file belonging to another user
        then this routine allows him to check that he has write access
        permission, and that no other user has the file connected in a
        conflicting mode.

     * It allows a user to deal in a controlled way with a failure to
        connect a file.

Edinburgh Regional Computing Centre

EMAS 2900 User's Guide

**A Guide to the use of the Edinburgh Subsystem of the
Multi-Access System EMAS 2900**

Second Edition: January, 1980

PREFACE

EMAS 2900 is an implementation of the Edinburgh Multi-Access System for ICL 2900 series computers. This Guide describes the facilities provided by the Edinburgh Subsystem of EMAS 2900, and its appearance to users. The first three chapters describe the System as a whole, but only to place the Subsystem in context. Readers wishing further details of the structure of EMAS 2900 should consult Reference 1.

EMAS was originally implemented on an ICL 4-75 computer, and EMAS 2900 is thus a re-implementation. The appearance to a user of EMAS 2900 is similar but not identical to that of the 4-75 System. Likewise, this manual is based on the 4-75 EMAS User's Guide, although some sections have been rewritten, the order of presentation in some cases has been altered, and there are numerous differences of detail.

In preparing the Guide I have been assisted by many members of staff of ERCC, particularly Roderick McLeod, who wrote the previous User's Guide, wrote much of the new material in this Guide and, as the person currently responsible for the maintenance and development of the Edinburgh Subsystem, has been my main source of information. The other members of the EMAS 2900 project team have readily provided information and many proof-read the entire Guide, as did some of the Advisory Service staff and the Documentation Officer, Neil Hamilton-Smith. Annette Marnoch typed the new material and assisted with some of the editing of the existing text. The Guide was printed by the ERCC Reprographics department, who also drew the figures and added the artwork.

It is hoped to keep this manual reasonably up-to-date, and page-replacement updates will be issued from time to time. Other sources of information about EMAS 2900 are described in Chapter 4.

John M. Murison
March 1979

PREFACE TO THE SECOND EDITION

In the relatively short time since the first edition of the Guide was published there have been a large number of additions to the facilities in the Edinburgh Subsystem, and also some changes in the existing facilities. Details are as follows:

Chapter 2: Job and file input and output

There are a number of new scheduling parameters (see Table 2.1), and some examples of file and job input are included (pp. 2-4 to 2-6). Note that DEST=NEWFILE is now available.

Chapter 8: Editing character files

Subsystem editor: the form F<filename>-MOD is now available; see p. 8-8. Edinburgh Compatible Context Editor: this is now invoked by the command ECCE rather than E, and the files specification has also changed. The following have changed or are new features: C, C-, %L, %U, secondary input, secondary output. L and R can now be used in SHOW and RECAP.

Chapter 10: Magnetic tape file handling

Magnetic tape facilities are now available as part of the Edinburgh Subsystem. Access may be restricted, especially to interactive users.

Chapter 11: Compilers, object files and program loading

Dynamic loading is now available; see p. 11-7.

Chapter 16: Running work in background mode

A background job control language is described (see pp. 16-3 to 16-21). A number of new Subsystem commands related to this new facility are also described.

Chapter 17: Accounts, usage information and ancillary commands

The command DOCUMENTS is now available; the command QUEUES has been withdrawn.

Appendix 5: Site-specific information

This is a new appendix, comprising summaries of the current EMAS 2900 installations and the System parameters chosen for them. This appendix might not be included in your copy of the Guide: if it is missing contact your local Computing Service, who should have copies of the part relevant to the EMAS 2900 service(s) available locally.

A large number of minor changes to the text, in some cases reflecting minor changes to the Subsystem, have also been incorporated.

<div align="right">

John M. Murison
January 1980

</div>

CONTENTS

LIST OF TABLES

LIST OF FIGURES

EMAS 2900 is a general-purpose, multi-access operating system which can run on most of the larger computers in the ICL 2900 range. The important characteristics of EMAS 2900 are:

* Multi-user - in theory any number of users can access the System simultaneously. In practice, the number is determined by the configuration of the comuter system.

* 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 Appendix 1 for a description of paging).

EMAS 2900 shares the resources of the computer between the users in an equitable fashion, and in such a way that each user appears to have a computer to himself. This computer is made up of a virtual processor, a virtual memory of 48 Mbyte currently, and file storage space on disc. The word "virtual" in this list of components can be read as "apparent" or "effective". The illusion implied by this terminology is clearly necessary, since the real main store of the real computer - normally between 1 and 8 Mbyte - is far less than 48 Mbyte, let alone (48 x no. of users) Mbyte; and the number of processors - normally one, in some configurations two - is far less than the number of users. Thus a very fast computer with a modest amount of main store and 1 or 2 processors is made to appear to be a large number of much slower machines, each with an enormous main store. Further, while the user's virtual machine is of necessity much slower than the real 2900 computer, the user will not in general make constant demands on it: for example, sometimes he will be editing files (requiring relatively little computing), sometimes he will be compiling programs (requiring quite a lot of computing), sometimes he will be doing nothing at all. The System has to be able to cope with this variability without wasting resources.

STRUCTURE

The System structure summarised below is also illustrated in Figure 1.1. A fuller description is given in Reference 1.

Processes

In sharing the resources of the machine, the System must work with a basic unit of scheduling. This unit is known as a process. It is an autonomous activity which competes with other processes for the resources available. Each user has a process. The purpose of EMAS 2900 can thus be described as "providing a suitable environment for each user process". In achieving this the System will have a variety of tasks to perform, some of them related directly to the users' requirements, some concerned with the housekeeping entailed in organising the System as a whole. These tasks are also associated with processes and also compete for their share of the resources available.

Processes fall into two classes. First, "Supervisor processes", resident in main store, i.e. not paged. Secondly, "paged processes", which run in the virtual machines described above; these are created dynamically, for instance when a user logs in.

Supervisor processes

The components of EMAS 2900 which are implemented as Supervisor processes are as follows:

Local Controllers

Each paged process has an associated Local Controller, whose main function is to manage the process's virtual processor and virtual memory. It ensures that the process stays within its current allocation of resources; if it requires more, the Local Controller will ask the relevant component of the Global Controller (see below) for more.
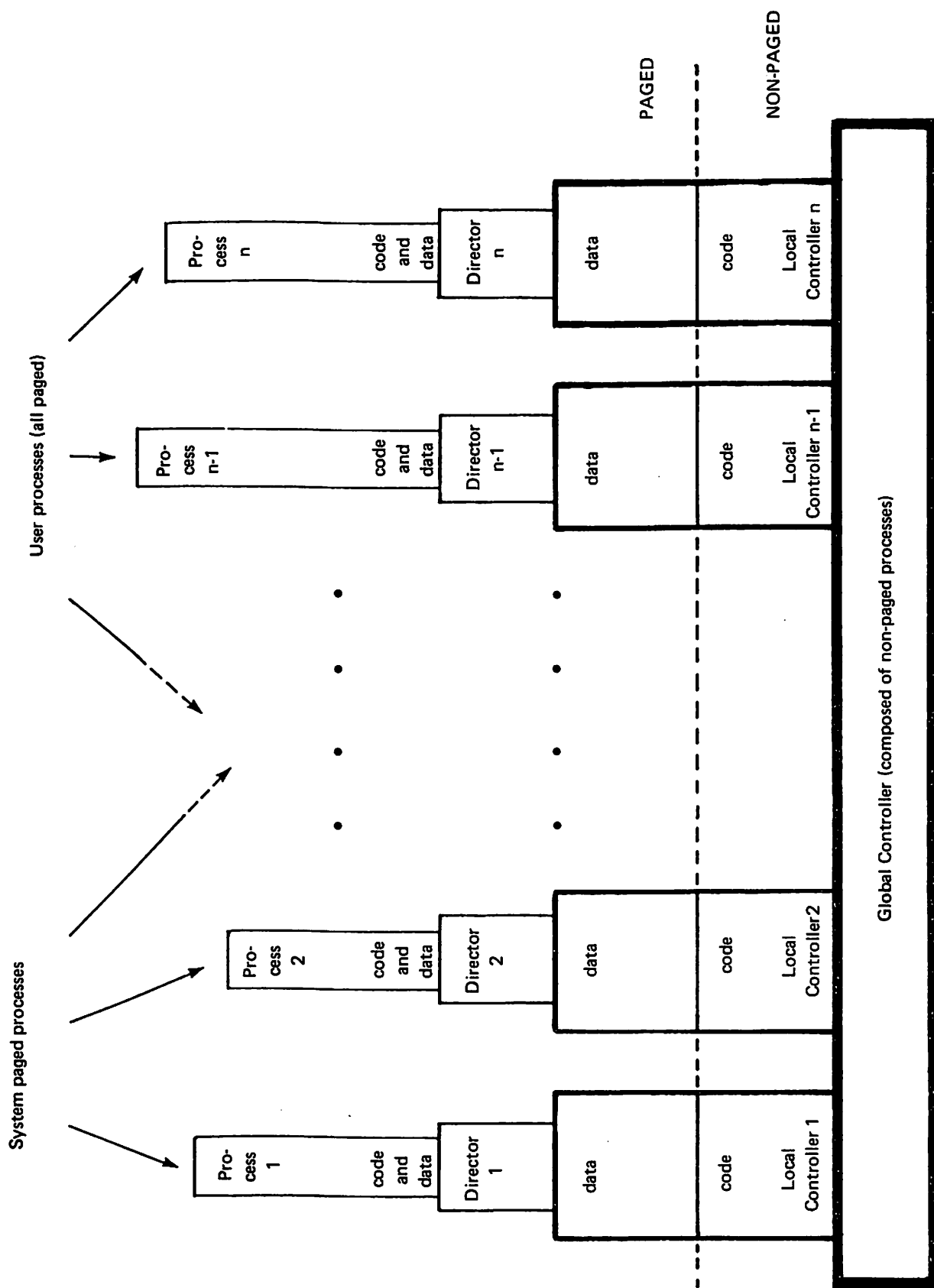
Figure 1.1: EMAS 2900 System Structure

The Local Controllers are unusual in that their code is resident, while their data space is located in the virtual memories of their associated paged processes.

Global Controller

The Global Controller is central to EMAS 2900, and is responsible for the following:

* Scheduling: Part of the Global Controller, known as the Scheduler, is concerned with sharing available main store and central processor time between the various Local Controllers competing for these resources. Since EMAS 2900 is primarily intended 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 2.

* Paging: Part of the Global Controller, known as the Paging Manager, is provided for Local Controllers to use. Its function is to bring pages into store as required. As this service is part of the Global Controller, it can take account of sharing of pages in main store. In other words, if a Local Controller has already requested a certain page, the next Local Controller to ask for it can be told that it is already in main store. This phenomenon occurs very frequently, particularly for editor and compiler pages, and helps to make good use of the available main store.

* Controlling peripheral devices: The Global Controller contains a number of device drivers, which control the principal rotating peripherals (disc, drum) and general peripherals (card reader, line printer, magnetic tape, etc.).

Paged processes

Each active user of EMAS 2900 has a paged process. The Supervisor processes described above provide each paged process in the System with a virtual machine, but not with software facilities. These facilities are provided by two components, which are both located in the virtual memory of the paged process in question (see Figure 1.2): the Director, and the Subsystem.

All the services provided for a user by EMAS 2900 are requested by calls on Director routines, which thus form a "procedural interface" between a user process and the rest of the System. These services, however, are at a basic level and are intended as the primitives from which a higher-level interface between the user and the System is to be built. Such an interface is known as a Subsystem, which in EMAS 2900 provides such things as a command interpreter, text editors, compilers, a program loader, subroutine libraries, etc. The appearance of EMAS 2900 to a user is determined by the appearance of his Subsystem.

Most of this Guide is concerned with the facilities provided by the Edinburgh Subsystem, which is the standard one. However, as his Subsystem resides in his own virtual memory, it is quite feasible for a user to use a different Subsystem, or indeed no Subsystem, without affecting other users. Thus it is possible to have a number of different Subsystems in use simultaneously in EMAS 2900, each making use of the facilities provided by Director, but each appearing different to its users.

Note that the JOBBER subsystem, which was designed to provide a low-overhead development environment for scientific programs, is not described in this Guide. Details of the JOBBER subsystem and its availability should be sought from your local Advisory Service.

The Director, which is a collection of routines, 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's program part of the virtual memory is made inaccessible (by a hardware protection mechanism), to prevent access to the Director's code and tables. This mechanism ensures that faults in the Subsystem or in user programs cannot result in corruption of the Director's tables. Since these tables include information about, for example, other users' files, this mechanism is essential to preserve the security of the System. Subsystem faults or user program faults 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
(Mbyte)

48   ┌─────────────┐
     │             │
     │             │
     │  Subsystem, │         Note: The shaded area is not accessible
     │  User Programs│        to the process when the Subsystem or a
     │  and Files  │         user's program is being executed.
     │             │
     │             │
 8   ├─────────────┤
     │▨Director Code▨│
     │▨and Tables and▨│
     │▨Local Controller▨│
     │▨Tables▨▨▨▨▨│
 0   └─────────────┘
```
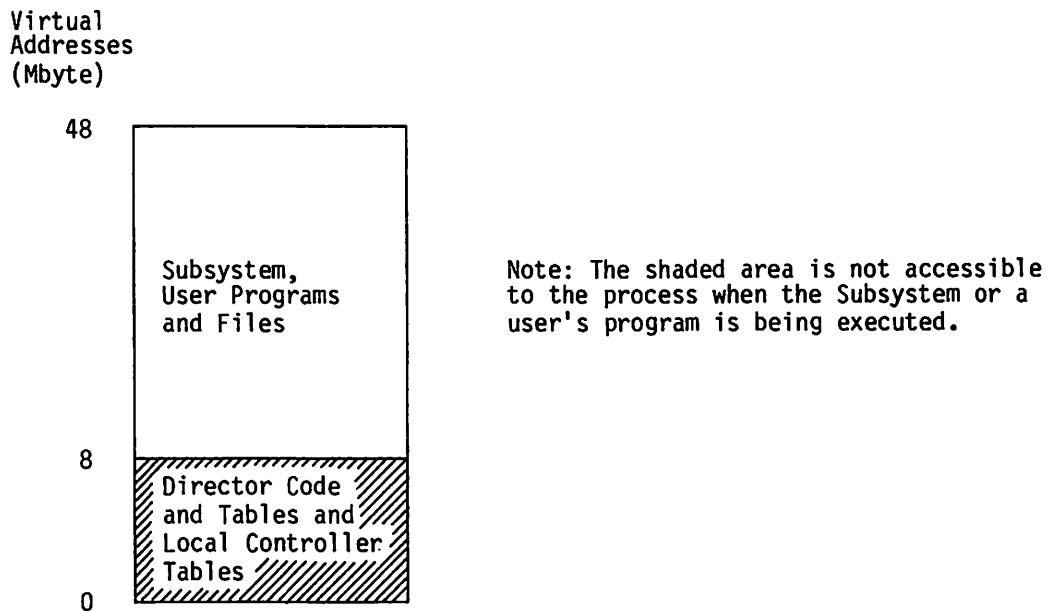
Figure 1.2: Layout of a Virtual Memory

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

* organising the disc file store (see below)

* communicating with System processes (see below)

* communicating with an interactive terminal (see Chapter 3)

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

Paged System processes

Apart from the paged processes owned by users, there are a number of paged processes
belonging to the System. In many ways they are similar to user processes but they have
some special attributes:

* They are privileged - that is, they can access information and obtain services which
  are not available to user processes.

* They normally run without any interactive terminal.

The three most important paged System processes are the SPOOLR process, the VOLUMS process
and the DIRECT process.

The SPOOLR process is responsible for:

* handling input files and jobs from slow devices, e.g. card readers (see Chapter 2)

* handling the output of files to line printers, etc. (see Chapter 2)

* scheduling batch jobs (see Chapter 16)

* controlling communications with remote terminals (see Chapter 2)

The VOLUMS process is mainly concerned with the organisation of the backup and archive
stores, described below.

The DIRECT process is responsible for the verification of names and passwords at log on
(see Chapter 3), and for carrying out consistency checks on the file system (see below).

HARDWARE CONFIGURATION

This manual does not contain a detailed description of a suitable hardware configuration, or of any part of such hardware. Some configurations in use, at the time of writing, are summarised in Appendix 5.

THE FILE SYSTEM

Files are used in EMAS 2900 for a wide variety of purposes. They can contain programs and data, as in conventional systems, but they are also used to hold temporary information such as the variables and arrays for a running program. The file system is divided into two parts: the disc store and the archive store. A file in the disc store is available for use whenever the user logs on to the System. A file in the archive store is held on magnetic tape and must be transferred to the disc store before it can be used.

All files are made up of one or more pages of information, the size of each page being 4096 bytes. (The hardware works with pages of 1024 bytes, but EMAS 2900 uses "extended pages" (epages), currently comprising 4 hardware pages.) As explained in Appendix 1, when a file is to be accessed, it is connected, i.e. its 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 appropriate part of the user's virtual memory. One way in which EMAS 2900 differs from some other systems is that it frees the user from any concern about the physical layout of his file on backing store. The System controls the way in which a file is stored on disc or on tape. The file storage hierarchy is summarised in Figure 1.3.

Note that in the description following, Edinburgh Subsystem commands provided for accessing and manipulating files are used for illustrative purposes. It should be appreciated, however, that any Subsystem will have equivalent facilities (given to it by the Director), although they might be presented to users in a different way. The Edinburgh Subsystem is sometimes referred to below as "the Subsystem".

```
        ┌ Archive files
        └
      ┌ Disc files
      │   ┌ Connected files
      │   │   ┌ Pages in main store
      │   │   └
      │   └
      └
```
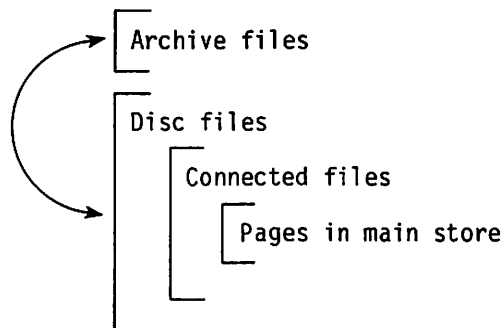
Figure 1.3: Storage Hierarchy

THE DISC FILE STORE

The disc file store is organised by the Directors of each paged process acting in co-operation. A Director is only concerned with files as sequences of pages; it is not concerned with the internal structure or contents of files. Thus those Subsystem functions which are not concerned with file types, e.g. the functions initiated by the commands RENAME and DESTROY, can be passed straight on to the Director.

Naming files

Each file in disc storage has a unique name which comprises two parts: the name of its owner (the six-character "ownername") and a file name given by the owner to distinguish it from his other files (the "filename"). In the Edinburgh Subsystem the two fields are separated by a full stop. Files created by users have filenames of between 1 and 11 characters which must all be upper case letters or numerals, the first character being a letter. Examples of valid full filenames, using the Edinburgh Subsystem convention:

```
ERCC06.FILEABCXYZ1
ERCC99.N
LMPTO1.FIRE
```

The filenames of files created by the Subsystem usually start with the characters "SS#" or
"T#" (depending upon whether they are permanent or temporary) to avoid conflict with files
created by the user. For example:

```
ERCC06.SS#OPT
ERCC99.T#LIST
```

Note that when referring to his own files a user does not normally have to prefix the
filename with his ownername. Thus user LMPTO1 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 2900 without risk of corruption or illegal access.

Access permission

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

```
READ
WRITE
```

In addition, permission to EXECUTE a file (i.e. to run a program) can be given. However,
since it is also necessary to have READ access permission in order to execute a file,
EXECUTE permission on its own is not useful and so READ access permission is assumed by
the Director to imply EXECUTE permission also.

Any or all or none of these permissions can be given to a file in respect of

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

When created, a file has both types 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 PERMIT is used to set access permission information for a file. Its
use is described in Chapter 5. When determining what access permission is allowed to a
user who wishes to access a file (by connecting it in his virtual memory), 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 with access
  permission the resulting permission is undefined.

1-6

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 (see table below).

The following modes of file connection can be used:

* READ - 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 mode. It cannot be written to or modified.

* READ and EXECUTE - 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 and EXECUTE mode. It cannot be written to or modified but it can be executed.

* WRITE - 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 can be written to or modified.

  It is also possible (though not currently via the Edinburgh Subsystem) for a user with WRITE access permission to specify that the file being connected may be shared, i.e. connected in other users' virtual memories, but only if these other users each have WRITE permission and have likewise allowed the possibility of sharing.

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 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 above information is summarised in Table 1.1:

Connect Mode	Whether allowed if already connected in another virtual memory	Can be read from	Can be executed	Can be written to or altered
READ	Yes, if connected in READ mode	Yes	No	No
READ and EXECUTE	Yes if connected in READ and EXECUTE mode	Yes	Yes	No
WRITE	No, unless connected in WRITE mode and all users involved have allowed the possibility of sharing	Yes	Yes	Yes

Table 1.1: File Connection Conditions

The structure of the disc file store

The disc file store is divided into a number of separate parts, each of which resides on a separate disc-pack. One result of this is that the effect of a disc 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 contains his files.

The user file index

Each user has a file index, which resides on the same disc-pack as all his disc files. A file index has three sections:

* System File Information (SFI): this contains information about the user's process which has to be retained between sessions, such as metering information (Chapter 17) and the passwords he has selected (Chapter 17).

* File descriptors: there are about 100 of these, depending on the size of the index. One is required for each file owned by the user in the disc 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 currently 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 (the number depends 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: 1 cell for each 32 epages, apart from the first 32 epages

    * for each file permitted to a specific user or user group: 2 cells for each separate permission

    * for each file currently on OFFER to another user (see below): 2 cells

Information from a user's file index in respect of a particular file or of all his files can be obtained by use of various Subsystem commands; see Chapters 5 and 6.

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

An upper limit on the size of each disc file can be imposed, on individual users separately, by the System Manager; an upper limit on each user's total disc file space can likewise be imposed by the System Manager. The current limits for some EMAS 2900 services are given in Appendix 5.

The size of an existing file can be altered by its owner, to a value between 1 epage and the upper limit imposed. This function is requested by the Subsystem, as required.

Transfer of ownership of a file

The Subsystem commands OFFER and ACCEPT (Chapter 5) result in a file being transferred from one user to another. The information relating to the name, the size, the location in the disc store 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.

Back-up of the disc file store

In order to protect users' files from hardware and System faults, they are copied onto magnetic tape, i.e. "backed up", periodically (once a day at Edinburgh), 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 (set by the System Manager) of non-use. The length of this period is given for some EMAS 2900 services in Appendix 5.

* 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 mode to cause back-up to take place.)

If, because of a System failure, it is necessary to recover users' files from back-up tapes, the files are copied to the disc store 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.

* Disc files which have been DESTROYed by the user after being backed up might be copied back to the disc file store in the event of the back-up of a number of files being required, e.g. as the result of a System failure. It would then be necessary for the user to DESTROY them again.

Automatic re-prime

If a user's file is corrupted as a result of a System failure or hardware fault, it is deleted automatically at the next IPL. At the same time a message of one of the following forms is sent to the user and it will be printed when he next logs on for a foreground session:

        **VOLUMS dd/mm/yy hh.mm UNCHERISHED FILE file LOST

        **VOLUMS dd/mm/yy hh.mm CHERISHED FILE file LOST - WILL BE RELOADED

If the file was CHERISHed, an automatic re-prime is initiated to transfer the latest copy in the back-up store to the disc store. If this succeeds a message of the following form is printed on the user's terminal, immediately if he is logged on, or when he next logs on:
        **VOLUMS dd/mm/yy hh.mm file RELOADED

THE ARCHIVE FILE STORE

Information about a user's archived files is contained in his archive index, which resides on the same disc-pack as his file index and disc files.

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

* as a direct consequence of the Edinburgh Subsystem command ARCHIVE (or equivalent) being used in respect of the file

* indirectly as a result of CHERISHing a file but not accessing it for a period set by the System Manager (see Appendix 5).

The commands ARCHIVE and RESTORE (for restoring an archived file to the disc store), are described in Chapter 5. Unused files are moved into the archive store in order to free the limited space in the disc store for material which is being actively used.

There is currently no limit to the number of files a user can have in the 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; the command DISCARD (Chapter 5) is provided for the purpose.

Unlike files in the disc store, two or more files on archive can have the same name. Their dates of archiving are then used to distinguish between them.

The input and output of files and jobs to and from an EMAS 2900 System is controlled by the paged System process SPOOLR (see Chapter 1). Users do not access file input and output devices directly, e.g. card readers or line printers. Instead they instruct the SPOOLR process to carry out operations on these devices with reference to particular files or jobs.

SPOOLR works with documents, a document being made up of scheduling parameters plus the file or job to which they relate. SPOOLR does not examine the contents of the file or job, only the values of the associated scheduling parameters. From these values it determines whether a file is to be sent somewhere, or a job is to be executed; in the latter case, the process in which the job is to be executed, the conditions associated with the running of the job, and the destination of output from the job are also determined by the values of the scheduling parameters.

INPUT

The scheduling parameters required and the method of specifying them when submitting files or jobs for input to an EMAS 2900 System are described below. The processing of a job once it has been sent by SPOOLR to the specified destination process depends on the particular Subsystem in use. The job control statements, etc., required by the Edinburgh Subsystem are described in Chapter 16.

Input is usually achieved by submitting the file or job, with appropriate document scheduling parameters, to be read by a card reader or paper tape reader. An appropriate EMAS 2900 Input Card should accompany the cards or paper tape(s); these should be available at Job Reception points. Alternatively, input can be sent from terminals connected to a network to which the EMAS 2900 mainframe is also connected; and files and jobs can also be transferred from other processors connected to the network. The details of these operations are subject to change as networks are extended and modified; consult the HELP information (described in Chapter 4) for current facilities relating to a particular EMAS 2900 service.

Card reader input

Cards can only be read in a mode involving translation from IBM 029 card code to ISO internal code. The file produced is an Edinburgh Subsystem character file (see Chapter 4 for details). The following rules apply:

* All 80 columns are read.

* Trailing spaces are deleted; that is, all the spaces following the last non-space character on each card are excluded from the file.

* Newline characters are inserted in the file to signify the end of each card.

Paper tape input

Paper tape can only be read in one mode: ISO-coded 8-hole even-parity. 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 characters are ignored when they are adjacent to newline characters.

* Trailing spaces (space characters immediately before a newline character) are ignored.

* All paper tapes must have at least 12 inches of run-out at both ends.

Remote processor input

Files are normally constructed on the remote processor with the document scheduling parameters included; they are then sent via the communications network to the EMAS 2900 System. The following rules apply:

* The final //DOC statement (see below) may be omitted from the file sent.

* The maximum size of file which can be handled by SPOOLR is 8 Mbyte. However, a lower limit may apply; consult your local Advisory Service for the limits applying to the relevant network and EMAS 2900 service.

Scheduling parameters for file and job input

A document is input in the following form:

```
//DOC parameter=value,parameter=value,...

    contents of file or job

//DOC
```

The first DOC statement marks the start of a document and contains the scheduling information, specified as a series of parameter assignments; the available parameters and their meanings are given in Table 2.1. Each assignment is terminated by a comma or newline (or end of card). If a line finishes with a comma, this implies that the DOC statement is continued on the next line. As many lines as desired may be used.

There are no spaces before //, or between // and the word DOC; however, at least one space must follow DOC. Thereafter spaces are ignored unless they lie between the first non-space character and the last non-space character of the parameter value. Thus

```
//DOC PASS = 1234,    USER    = ERCC99,
    DEST= FILE, NAME= NEW1,
    DELIV = Chemistry King's Buildings
```

is equivalent to

```
//DOC PASS=1234,USER=ERCC99,DEST=FILE,NAME=NEW1,DELIV= Chemistry King's Buildings
```

Note that the DELIV parameter value can include spaces (as above), but not commas.

Notes on the scheduling parameters

* If a file is being created from input, the parameters USER, PASS, NAME and DEST must all be specified; the other parameters are not relevant.

* If a batch job is being submitted, the parameters USER, PASS and DEST must be specified; the other parameters are optional. When a JOBBER job is being submitted, however, only the DEST parameter is mandatory.

* Not all of the parameters given in Table 2.1 have to be specified - there are defaults. The defaults can differ for each document destination. See Appendix 5 for details.

* Only one file can be created per //DOC ... //DOC set of statements.

* The //DOC statements must both start on a new line; this means that for paper tape input there have to be line feed characters before and after both //DOC statements.

* The NAME chosen for an input file can be the same as the name of an existing disc file belonging to the specified USER. If it is, the existing file will be overwritten if DEST=FILE was used. However, if DEST=NEWFILE was used, the existing file will be retained and the one being read in will be discarded.

* Files are CHERISHed automatically on input.

* Files may be placed immediately in output queues for printing or routing to a remote processor. This is done by specifying DEST as the appropriate output device mnemonic (see "Output device mnemonics", below).

Parameter	Meaning
USER	The six-character username of the process receiving the file or executing the job.
PASS	The background password of the USER specified (see the Edinburgh Subsystem command PASSWORD, Chapter 17).
NAME	If the DEST specified is FILE or NEWFILE (see below) it is the name to be given to the disc file to be created. It must be 11 characters or less, starting with a letter. A member of a partitioned file (an Edinburgh Subsystem file type) may not be specified. If the DEST specified is not FILE or NEWFILE then this parameter is the name to be associated with the document as it is queued, executed or printed by the System. This name will be used in output from the DOCUMENTS command (see Chapter 17).

DEST	The destination of the document. Possible destinations are:
	
	    FILE
	    NEWFILE
	    BATCH
	    JOBBER
	
	and any valid output device, e.g. LP, GP, LP15 (see "Output device mnemonics", below). If DEST is specified as FILE then a file of the given NAME (see above) is created from the input document. If an existing file belonging to USER (see above) has the same name then it is overwritten. However, if DEST is specified as NEWFILE then a file of the specified name must not already exist; if it does it will not be overwritten - instead the input document will be ignored.

DELIV	Up to 31 characters of delivery information to be used on any output produced. If DELIV is not specified the default delivery for the user is used. Note that specifing DELIV does not change one's default delivery.
TIME	If the document is a job, the OCP time in seconds required.
RERUN	If the document is a job, whether the job is to be rerun if caught in a machine crash. If the document is an output file, whether it should be reprinted if caught in a machine crash. Should be specified as YES or NO.
COPIES	The number of copies of output required. Valid values 1-255.
FORMS	The forms type required for output. Valid values 1-255. Consult your local Advisory Service for information on the values defined with respect to a particular installation.
ORDER	The relative priority of execution of this document with respect to other documents with the same DEST and belonging to the same user. Thus, for example, if there are two BATCH jobs belonging to the same user in the System at the same time with ORDER values of 2 and 5, the job whose ORDER is 2 will be run first. Documents should be submitted in the required running order. Note that this parameter can also be used to determine the ordering of documents being sent to an output device. Valid values 1-255.

AFTER	The document is to be held in the queue until after the date and/or time specified. Valid values are of the form:
	
	    date time
	    time date
	    date
	    time
	
	where "date" is in the form dd/mm/yy, and "time" is in the form hh.mm.ss. See example (8) below.

Table 2.1: File and Job Scheduling Parameters

```
1)      //DOC USER=AAAA99,PASS = ABAB,NAME=DATAFILE23,DEST=NEWFILE
            1  2  3  4  5  6
            72 341 896 247 839
            .
            .
            .
            -1 -1
        //DOC


2)      //DOC USER=ABCD01,DEST=FILE ,NAME=PROGA,PASS=XZ09
        %BEGIN
        %INTEGER X,Y,Z
            .
            .
            .
        %ENDOFPROGRAM
        //DOC


3)      //DOC DEST= BATCH,USER=AAAA99,PASS=XK22, NAME=BATCHJOB,
        TIME=3,DELIV=JCMB Room 2020
        SPSS(CONTROL=*,LISTING=.OUT)
        .DATA
        DATA LIST FIXED/1 AGE 1-2 SEX 3 INCOME 4-6
        INPUT MEDIUM CARD
        N OF CASES 10
        CROSSTABS TABLES=SEX BY AGE, INCOME
        STATISTICS ALL
        READ INPUT DATA
        221937
        841395
        932168
        201904
        932027
        441439
        342596
        341439
        632374
        741439
        FINISH
        .ED
        //DOC


4)      //DOC DEST=JOBBER
        //JOB (ERCC99,G.SMITHERS)
        //OPTIONS(NOLIST,LABELS)
        //FORTE
            <program text>
        //DEFINE FILE (4,*)
            <data for channel 4>
        //DEFINE FILE (2,OUTF1)
        //RUN
            <data for (default) input channel 5>
        //ENDJOB
        //DOC


5)      //DOC DEST=BATCH,USER=AAAA99,PASS=XK22,ORDER=1,
            TIME=300
            .
            .
            .
        //DOC
```

6)   //DOC DEST=BATCH,USER=AAAA99,PASS=XK22,ORDER=2,
            TIME=200
            .
            .
            .
        //DOC

7)   //DOC DEST=BATCH, USER=AAAA99, PASS=XK22, ORDER=3,
            TIME=100
            .
            .
            .
        //DOC

8)   //DOC DEST=BATCH,USER=XXYY77,PASS=)31(,NAME=AFTERJOB,
            AFTER=10/05/79 12.30.00
        GENSTAT(CONTROL=*)
        .DATA
        'REFERENCE' ANOVA(1) ''
        COMPLETELY RANDOMISED 5X3X4 FACTORIAL DESIGN, TAKEN FROM DAVIES,O.L:
        DESIGN AND ANALYSIS OF INDUSTRIAL EXPERIMENTS (OLIVER AND BOYD 1954) P.291
        WEAR RESISTANCE OF VULCANISED RUBBER
        TREATMENT FACTORS: A - 5 QUALITIES OF FILLER
        B - 3 METHODS OF PRETREATMENT OF THE RUBBER: C - 4 QUALITIES OF RAW RUBBER.
        ONLY ONE REPLICATE, THUS 3 FACTOR INTERACTION USED FOR ERROR. ''
        'UNITS' $ 60
        'FACTORS' A $ 5 : B $ 3 : C $ 4
        'GENERATE' A,C,B
        'HEADING' HW='' RESISTANCE OF VULCANISED RUBBER''
        'DESCRIBE' WEAR $; HW
        'READ/PRIN=DE' WEAR
        'TREATMENTS' A*B*C
        'ANOVA/LIMA=2' WEAR
        'RUN'

        404 478 530 381 429 528 316 376 390 423 482 550
        392 418 431 239 251 249 186 207 194 410 416 452
        348 381 460 327 372 482 290 315 350 383 376 496
        296 291 333 165 232 242 158 279 220 301 306 330
        186 198 225 129 157 197 105 163 190 213 200 255
        'EOD'
        'CLOSE'
        'STOP'
        .ED
        .ENDJOB
        //DOC

Notes on the examples

1) Creation of disc file AAAA99.DATAFILE23; a file of the same name must not already
   exist. User AAAA99's background password must be ABAB.

2) Creation of disc file ABCD01.PROGA. Note that the space following FILE is ignored.

3) A SPSS batch job using the Edinburgh Subsystem job control language (described in
   Chapter 16). Note that the delivery information (parameter DELIV) is not enclosed in
   quotes and that, because of the rules about spaces in DOC statements, the space
   characters in the delivery information are preserved. Note also that SPSS may not be
   available on all EMAS 2900 services. Consult your local Advisory Service for
   information on availability.

4) Note all that is required (DEST=JOBBER) for JOBBER input. Note also that the job
   control statements required for JOBBER jobs are not described in this Guide. (This
   example is artificial in the sense that a number of JOBBER jobs would normally be
   batched together by Job Reception staff before being submitted, in which case the user
   would not supply the //DOC statements. Consult your local Advisory Service for
   information on the availability of a JOBBER subsystem.)

5),6),7) If these three background jobs were submitted together in that order, they would be run in the order specified via parameter ORDER - i.e. (5) then (6) then (7). If ORDER had not been specified, the order of running would have been (7) then (6) then (5), because of the TIME parameter settings; other things being equal, the shortest job will be run first.

8) This shows how a job can be held until after a specific date and time. Note that the use of AFTER does not guarantee that it will be run at the time specified, only that it will be run after this time. The use of GENSTAT in this example should not be taken to mean that it is available on all EMAS 2900 services. Consult your local Advisory Service for information on its availability.

OUTPUT

Information can be sent to output devices by two methods:

* By an explicit Subsystem command such as LIST or SEND (see Chapter 6).

* 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 DOCUMENTS is available to tell the user whether any documents of his are waiting in output queues (see Chapter 17).

In addition to these methods for sending information to output devices, it is possible to send a document from an input device directly to an output device. In this case the user's disc files are not involved. The document should be submitted for input, as described above, but with the DEST parameter specifying an output device (e.g. LP, CP).

For example:

```
//DOC  DEST=LP15, DELIV=Smith Chemistry K.B., FORMS=3,
       COPIES=2
   15  16  17   3   9 -10
   11   1   3   9  16   4
    .
    .
    .
//DOC
```


Output device mnemonics

SPOOLR uses mnemonics to refer to output devices, for example LP for the line printer. When using the Edinburgh Subsystem, however, users should prefix the mnemonic with a full stop; thus .LP. The dot is used to distinguish the device from a file name, since a device mnemonic or a file name can often be specified for a particular parameter in a Subsystem command. In the rest of this chapter, the Edinburgh Subsystem convention of prefixing a mnemonic with a full stop will be followed. However, when a mnemonic is specified as the value of the DEST parameter (see Table 2.1), the dot must not be given.

If the output device is not connected directly to the EMAS 2900 mainframe 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 or in the EMAS 2900 Information Card. Refer to Appendix 5 to find out which devices are directly connected to some specific EMAS 2900 mainframes. Users of installations not described in Appendix 5 should contact their local Advisory Service for this information.

Table 2.2 gives the names and mnemonics of available output devices. The files can be character or data files (see Chapter 7), except when being sent to a graph plotter (data files only) or a matrix plotter (character files only); however, as the graph plotter and matrix plotter software provided (see Reference 3) generates the appropriate file type, no special action is required by the user.

Device	Mnemonic
Line Printer	.LP
Card Punch	.CP
Paper Tape Punch	.PP
Binary Paper Tape Punch	.BPP
Graph Plotter	.GP
Graph Plotter for liquid ink jobs	.SGP
Matrix Plotter	.MP

Table 2.2: Output Device Mnemonics (Edinburgh Subsystem Convention)

CHARACTERISTICS OF INDIVIDUAL OUTPUT DEVICES

Line Printer

* If the device is defined as .LP then lower case letters are converted to upper case if the line printer cannot 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 some of the line printers accessible from EMAS 2900 might not be able to do this.

Card Punch

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

* Any lines longer than 80 characters are split and continued on the next card.

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. In the process of checking the need for the parity bit, the 8th bit of the given byte is first cleared to zero. Thus if the byte has a numerical value of more than 127, the setting of the 8th bit will be ignored.

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

Graph Plotter

The graph plotter should be accessed via the graphics routines provided; these are described in Reference 3. The device .SGP is used to indicate that "special" facilities are required, usually liquid ink.

Matrix Plotter

The matrix plotter should be accessed via the routines provided; these are described in Reference 3. Further information can be obtained from your local Advisory Service.

The primary method of accessing EMAS 2900 is via an interactive terminal. There are two basic types of terminal - video or hard copy - with a considerable range of facilities and operating speeds.

Method of connection

An EMAS 2900 mainframe normally has some peripheral devices directly connected to it (e.g. a card reader and line printer). Apart from these, it communicates with the "outside world" through one or more Front End Processors (FEP). An FEP, which is itself a computer, is connected in turn to a communications network. The design of such a network, and the facilities it may provide, are beyond the scope of this Guide.

However, an interactive terminal used to access EMAS 2900 must be connected to the relevant network in some way, and this is normally via a Terminal Control Processor (TCP). This chapter describes how to select options relating to the TCP's control of the user's terminal. It assumes that the functional appearance, to an interactive terminal user, of the TCP corresponds exactly with the TCPs operated by ERCC. Users of EMAS 2900 services not operated by ERCC should consult their local Advisory Services to establish whether these or equivalent options are available to them.

Direct connection and dial-up connection

Most terminals are connected permanently to a TCP and can be used immediately to access EMAS 2900. Others are connected via a telephone, a Modem (Modulator/Demodulator) or acoustic coupler, and a PO telephone circuit to the switched public network. Before this arrangement can be used to access EMAS 2900, it is necessary to establish a link to a TCP by dialling the appropriate telephone number (031-667 1071 for the ERCC local network) 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, or with none, on the terminal from those typed. For example, during log on (see below) this facility is used to suppress the printing of the password.

* It is possible to type input whilst the terminal is printing output from the computer. The input characters are stored by the TCP, and echoed when the output line has been printed (or partially printed if the TCP has not yet received the end of the output line).

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 ERCC TCP.

Control character functions

Table 3.1 below 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

The "Response" column gives the characters printed by the TCP on the user's terminal as a result of his typing the appropriate control function character. The "Message" column gives the character(s) received by the EMAS 2900 mainframe via the communications network.

Character	Type as CTRL+	Effect	Response	Message	Note
CR (RETURN)		Terminate current line	CR-LF	LF	
EM	Y	Terminate current input	* CR-LF	EM	
CAN	X	Cancel current line	† CR-LF		
ESC		Escape to INT:	CR-LF INT:		1
HT	I	Tabs to next available tab position. Tabs set by default at 6,9,12,15,18,40,80 (N.B. This means that HT at start of line causes 6 spaces, not 5 spaces)	spaces	spaces	
DEL (RUBOUT)		Cancel previous character	\char		2
DC2	R	Repeats current input line, removing any "rubbed out" characters	CR-LF current line		3
DLE	P	Causes next character input to be treated as "binary"			4
SOH	A	Enter SET mode	CR-LF SET [		5

Table 3.1: TCP Control Characters

Notes

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

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

    * A single letter followed by CR. When the Edinburgh Subsystem is used, this is interpreted by the Subsystem itself; see Chapter 4.

    * Text of between 2 and 15 characters followed by CR. This constitutes a user interrupt, which can be detected by using the IMP function INTERRUPT (see Chapter 12).

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

See also "Video echo mode" (Table 3.2).

3. DC2 (CTRL+R) can be used to see what the current input line looks like, minus any rubbed out characters. The line need not be complete, that is, the user can add to it after using DC2.

4. The effect of pressing DLE (CTRL+P) is to cause the next character input to be treated as binary data, i.e. passed on to the mainframe by the TCP whatever its numerical value. Any control function normally associated with the character - any of those in the table above - is not initiated. It is only echoed if its numerical code value lies in the range 32-127 (see Table A2.1 in Appendix 2).

5. SET mode is used to send commands to the TCP itself to change characteristics of its operation. The details of doing this are described below. Note that the EMAS 2900 mainframe is not involved ·in this process in any way - the user is communicating only with the TCP. However, in order that the TCP can be aware of the user's presence, the user must be logged on, or the log-on sequence (see below) must at least have been started, before setmode can be entered.

Setmode

The effect of a user typing SOH (see above) on his terminal is to cause any characters on the current line to be discarded and the prompt "SET [" to appear. The user then specifies one or more setmode commands. The commands available are given in Table 3.2.

In general a command is a single upper or lower case letter (always echoed as upper case), followed by either:

a) a single space then one or more parameters, or

b) CR (RETURN), to terminate the setmode, or

c) a single space then another command, except after T (Tabs).

When the user types CR to terminate the setmode line, this is echoed as]CR-LF. Any error in the line will cause the string *?* to be printed before the], and the whole line is ignored. The input format must be followed precisely: for example, multiple spaces where only one is expected will cause the whole line to be rejected.

Note that EOT (CTRL+D) typed in at any point of a setmode line will cause the terminal to be disconnected from EMAS 2900 immediately.

Examples

1) To set the page height to 24 and the line width to 50:

    SET [H 24 W 50]

    The user must then type DC1 (CTRL+Q) (see "H" in Table 3.2)

2) To enter Video echo mode and set the first 4 tab positions:

    SET [V T 10,15,20,25,*]

3) To set the Delete character to "?":

    SET [D ?]

4) Error on line width setting:

    SET [W 999 *?*]

C	Cancel character	Replace the current Cancel character (default CAN) by the given parameter. (Note 1)
D	Delete character	Replace the current Delete (RUBOUT) character (default DEL) by the given parameter. (Note 1)
G	Graph mode	(No parameter) Enters Graph mode: that is, with respect to terminal output, all format controls are disabled, thus allowing characters with numeric values in the range 0-255 to be sent to the terminal. The only exception is the character with numeric value 10, which currently has CR inserted in front of it before being output on the user's terminal. Selection of U (Upper mode - see below) automatically terminates Graph mode.
H	Page height	Enters paging mode with "page height" equal to the given parameter x. After x lines of output have been sent to the terminal, no further terminal dialogue will be permitted until DC1 (CTRL+Q) has been typed. Note that output includes echoed input. N.B. A minimum page height of 5 is enforced. The terminal will be in the "end of page" state at the end of the setmode which sets the page size, i.e. it <u>must</u> be restarted with DC1 (CTRL+Q). A height of zero means no paging. (Note 2)
I	INT character	Replace the current INT: character (default ESC) by the parameter. (Note 1)
L	Lower mode	(No parameter) No conversion of lower to upper case is carried out on the input. This is the converse of U (Upper mode).
P	Pads	The given parameter is the number of pad characters required (default 0). (These are non-printing characters sent by the TCP to the interactive terminal following each CR. Some types of terminal require the resulting pause in printing to enable their print mechanism to return to the left hand margin.) (Note 2)
R	Return character	Replace the current Return character (default CR) by the given parameter. (Note 1)
T	Tabs	Set the Tab vector. Up to seven parameters (Note 2) may be given, separated by spaces or commas. An * instead of a number terminates the parameter list and causes all remaining parameters to be set to the current line width (see W below). If a parameter is less than its predecessor it will be set to the current line width. The parameter string must end with CR, i.e. no commands can follow T. Note that a tab value is interpreted here as the number of characters which have effectively been input following a use of the tab. This does not correspond with normal usage, in which the tab value gives the position of the <u>next</u> character to be typed. The default vector is 6, 9, 12, 15, 18, 40, 80.
U	Upper mode	(No parameter) All lower case letters typed by the user are translated by the TCP to upper case. All letters are thus echoed and sent to the EMAS 2900 mainframe as upper case (default). Selecting this mode terminates Graph mode (see G, above).
V	Video echo	(No parameter) Enters Video Echo mode to cause backspace, space, backspace to be echoed for Delete; that is, it removes the deleted character from the screen. N.B. This does not work on all videos. When already in Video echo mode, specifying Video echo mode causes a return to the normal delete mode.
W	Line width	Sets the current line width (default 72) to the value of the parameter, which must be in the range 15-160 inclusive.

Table 3.2: TCP Setmode Commands

Notes

1. Since the local editing functions (Rubout etc.) are available in setmode, non-printing characters should be entered using the DLE function since this will bypass any special meaning a character may have. For example, to swop over the current delete and cancel characters, the following setmode line could be given:

    SET [D (DLE)(CAN) C (DLE)(DEL)]

    where (DLE) etc. indicate that the character is non-printing.

    There is no consistency check on the current values of Return, Delete, INT and Cancel, it being up to the user to ensure that no two are the same. Valid characters must have numeric values in the range 2-127.

2. All numeric parameters must be decimal numbers in the range 0-255 inclusive. The particular setmode command may impose further restrictions on the parameter value (e.g. "W").

It is possible to specify setmode commands from the EMAS 2900 mainframe, as well as from the user's terminal. The method of doing this (Edinburgh Subsystem command SETMODE) is described in Chapter 17. However, it is not possible, at the time of writing, to determine from the Subsystem which setmode commands are currently selected.

Type ahead

An important characteristic of the terminal support mechanism on EMAS 2900 is that it allows 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 the user must appreciate 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 (see below) is not printed 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 or RECAP facilities (see Chapter 8) avoid 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 can be set by

* the Subsystem

* a Subsystem utility, e.g. a text editor

* a user program calling the routine PROMPT or FPRMPT (see Chapters 12 and 13)

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

LOGGING ON

Before a user can log on to the System from an interactive terminal he has to obtain an accredited username (see Chapter 17). The username has associated with it two passwords (see Chapter 17). The first of these, the "foreground password", is used for interactive access. In order to log on the following steps should be taken:

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

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

* Press the space bar, several times if necessary. If there is no response press the CR
  key.

* To the prompt "HOST:" reply with the name of the particular EMAS 2900 service to which
  you wish to log on; for example, 2970, 2980, EMAS. Contact your local Advisory Service
  if you do not know the name given to the relevant service.

* 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.

The communications network will pass the information received to the EMAS 2900 mainframe
specified. The System will then determine whether the user can log on. If he can, an
appropriate message will be printed. If the Edinburgh Subsystem is being used, the
following information is also output:

* the version number of the Subsystem in use

* the date and time

* the number of users currently logged on

* the number of the file system holding the user's index and files (see Chapter 1)

* (sometimes) a message of the day (the "foreground message")

Thereafter the prompt "Command:" will appear.

Alternatively, a message similar to one of the following, 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 your local Advisory Service.
PROCESS RUNNING	A background job is running in your process (see Chapter 16), or another person who shares the username is currently logged on. Note that it is possible to have several processes associated with a single username running simultaneously, but this is at the discretion of the System Manager.
NO USER SERVICE	This indicates a machine fault, or an attempt to log on outwith the service period. Where a service run by ERCC is involved, ring the ERCC answering service (031-667 7491) for information.
INVALID USER	This means either that the username was mistyped, or that the disc-pack containing the user's file index (see Chapter 1) is not on-line.
INVALID PASSWORD	This means that the password was typed incorrectly.

LOGGING OFF

In normal circumstances, a user of the Edinburgh Subsystem is at "command level" when he
wishes to log off, or he can get to command level by stopping or aborting the program
currently executing. He can then use the command STOP or the command QUIT (see
Chapter 17).

As explained earlier in this chapter, the user can disconnect his terminal at any time, by
typing SOH (CTRL+A) to get into TCP setmode, and then typing EOT (CTRL+D). However, this
should only be used when some System failure has apparently occurred which makes it
impossible to return to command level.

This chapter provides an introduction to the EMAS 2900 Edinburgh Subsystem. It explains the Subsystem's command language, describes its interactive terminal interrupts, introduces the file types provided, gives sources of further information about it, summarises its functions in logical groups and indicates how these functions are covered by the rest of this Guide.

The standard subsystem

The phrase "standard subsystem" is used to emphasize that the subsystem described (the Edinburgh Subsystem) is the one provided as a standard part of EMAS 2900. 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 15 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 16). Subsystem commands are typed according to the following rules:

* Each command must start on a new line.

* If the command requires one or more parameters, these should be typed after the command, normally enclosed in parentheses.

* As an alternative form of command input, it is possible to omit the parentheses enclosing the parameters, in which case the command name must be typed with no embedded spaces, and must be separated from the first parameter by one or more spaces; see OPTION (Chapter 17). For clarity all examples in this manual use parentheses.

* Parameters must be separated by commas. To indicate that a parameter has been omitted an extra comma must be inserted, if more parameters follow.

* Spaces within parameters are ignored. When parentheses are used to enclose parameters, newlines are also ignored; thus a command can be broken at any point. When parentheses are not used, continuation onto another line is not allowed.

* After removing spaces, newlines and parentheses, the total length of the parameters must be not greater than 255 characters.

Examples of commands

    ALERT
    LIST(ABC)
    LIST(ABC,.LP)
    FILES(,IA)
    DESTROY(ABC,DEF,GHI)

Command level

As explained in Chapter 3, once a user has successfully logged on, the prompt "Command:" appears on his interactive terminal. This is produced by the Subsystem whenever it is awaiting a command from the user; the user's process is said to be "at command level". Once a suitable reply has been given (e.g. one of the examples above), the command is entered, and the output generated and input required depends on the command in question. When the command terminates (or is aborted by use of an interrupt - see below), the "Command:" prompt reappears on the user's terminal, and he then gives another command.

This is repeated until the user logs off (see Chapter 3).

At present, the Subsystem will only accept commands typed in upper case (i.e. capitals). However, an option whereby lower case may be used instead is currently being considered. The ALERT and HELP commands will give further details when the option is available.

Command formats

Each Subsystem command description in this Guide starts with a line giving the general form of a call of the command:

        commandname(parameter1, parameter2, parameter3, ...)

The following is a summary of the conventions used in these "general form" statements:

* Underlined parameters are obligatory, others optional.

* Lists consist of one or more items separated by commas; e.g. "ownfilelist" is a list of files belonging to the user giving the command.

* The character "/" is used to separate alternatives.

* Commonly used parameter types are described in Table 4.1, others with the individual commands.

file	Any file (or member of a partitioned file).
ownfile	"file" belonging to self.
.IN, .OUT	These device names represent the current primary input and current primary output. For an interactive user, they both refer to his terminal.
outdev	This is used with commands that generate output. In almost every case the default output device is the interactive terminal (.OUT), and this can be assumed unless it is stated otherwise. Output devices are described in more detail in Chapter 2. The name of an output device consists of a full stop followed by a mnemonic; for example, .LP means line printer.
out	"outdev" or "ownfile" (but not a member of a partitioned file), to be used for command output. Default .OUT. (It is sometimes useful to direct the output from a command into a file for subsequent examination using a text editor or user program. For example, the command ANALYSE can be used to obtain information about a file. If a file name, not an output device, is given as the third parameter, then the information is put into the file.)
source	Input for a compiler. It can be a single file, or more than one concatenated by "+"; for example, SPECS+SRCE3+ROUTS_A.
clist	Listing produced by a compiler. The default is an "ownfile" with the name T#LIST. Alternatives are "ownfile", "outdev" or .NULL (meaning no listing is to be produced).
chan	I/O channel number, in the range 1-80.

Table 4.1: Edinburgh Subsystem Command Parameter Types

Apart from normal interactive terminal input and output there is a mechanism whereby any operation can be interrupted. The method, described in Chapter 3, allows the user to input a message of up to 15 characters. Multiple-character interrupts can be detected by a running program, by use of the IMP string function INTERRUPT (see Chapter 12). Single-character input messages are used to control the Subsystem, as described in Table 4.2:

Interrupt	Effect
A	Abort current command or program and return to command level.
C	As for A except that additionally any input that has been typed ahead is lost.
Q	Abort current command, print diagnostics and return to command level.
T	Print out the CPU time and number of page turns since the start of the command being executed, and the number of users logged on, without affecting the command.

Table 4.2: Edinburgh Subsystem Single Character Interrupts

SUBSYSTEM FILE TYPES

There are several types of file recognised by the Subsystem. The type of a file is determined from information held at the start of the file. The file types are given below in Table 4.3 with the number of the main chapter describing their use.

Type	Use	Chapter
CHARACTER	Contains characters, e.g. program source	7
DATA	Contains binary data, either in discrete records or unstructured	7,9
OBJECT	Contains a compiled program or routines or both	11
CORRUPT OBJECT	Contains a compiled program or routines or both, with compilation errors	11
DIRECTORY	Contains information to associate routine entry names and aliases with object files - used by the program loader	11
PARTITIONED	Contains members, each of which is for some purposes equivalent to a complete file	below

Table 4.3: Edinburgh Subsystem File Types

MESSAGES

Messages output by the Subsystem

In general, simple commands do not produce any output if they work successfully. A few, indicated in Table 4.5 (at the end of this chapter), produce confirmatory messages. 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 3.

Operator messages

Apart from messages generated by the Subsystem there are messages sent to interactive terminals by the EMAS 2900 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.

User messages

By use of the command TELL (Chapter 17) a user of the System can send a message to any other user. The format of the message received is similar to that of an operator message.

PARTITIONED FILES

A partitioned file is a complete EMAS 2900 file which has all the normal attributes of a file: a name, access permissions, cherish status and so on. Its contents are called members, and each member is similar to a complete file, of any of the types listed above. Thus a single partitioned file could contain members which were character files, members which were data files, and even members which were partitioned files. For many purposes a member of a partitioned file can be used in the same way as a file of the same type, and in the descriptions of the relevant commands (in the chapters following) examples of this are given. There are, however, exceptions to the rule; these are noted below.

Creating a partitioned file

        NEWPDFILE(ownfile)

The command NEWPDFILE is used to create a partitioned file. It takes one parameter - the name of the file to be created; for example:

        NEWPDFILE(HOLD)

A file of the name given must not already exist.

Operations on a complete partitioned file

All of the general file utility commands described in Chapter 5 can be used with a whole partitioned file - for example PERMIT and CHERISH - since these commands are not concerned with the contents of a file. Furthermore, COPY (Chapter 6) can be used to copy a complete partitioned file, and ANALYSE (Chapter 6) can be used to obtain a list of its members.

Accessing individual members

In general individual members can be used wherever a file is to be read from. This includes members which are directory files; these may be nominated for searching by the loader (see Chapter 11 for details). It also includes members which are object files. These may be INSERTed in directory files or executed directly by the RUN command (see Chapter 11).

Note that it is not possible to access directly a member of a partitioned file which is itself a member of a partitioned file - the inner partitioned file would first have to be copied out of the outer one.

Table 4.4 indicates which of the standard commands can be used to access individual members of partitioned files.

Command	Note	Example
ALGOL	Source only	ALGOL(PD_ALGTEST,AY)
ANALYSE		ANALYSE(PD_FLENZ)
CONCAT	For input files	Conc:FILE23_MEMBER2
CONVERT	For input file only	CONVERT(AB_DATA,CHAR)
COPY	For input or output (see above); note that a member cannot be copied to another member in the same partitioned file	COPY(AB_SRCE,SRCE) COPY(AB_NAME,BC_NAME2) COPY(CD,EF_MEM1)
DEFINE	For input files only	DEFINE(5,FPD_INFO)
DESTROY	See "Destroying and renaming members", below	DESTROY(HOLD_FILEA)
DETACH		DETACH(ACT_JCL,2)
DETACHJOB		DETACHJOB(A_B3)
ECCE	Can be used for primary input or output file or for secondary input or output file	ECCE(PD_E1,PD_E2)
EDIT	Can be used for input file or for the I<filename> facility	EDIT(PD_EFILE,EFILE2)
FORTE	Source only	FORTE(FPD_SOURCE,Y)
IMP	Source only	IMP(DIRSRCE_CPUT,CPUTY)
INSERT	Object file member must have been copied into pdfile after compilation	INSERT(A_OBJ)
INSERTMACRO	Macros are described in Chapter 16	INSERTMACRO(MACLIB_COMP1)
LINK	For input files	Link:ABC_DEF
LIST	But not SEND	LIST(PD_OUTLIST,.LP)
LOOK		LOOK(DIRSPECS_CPUT)
OBEY		OBEY(PD_OBEY)
OBEYJOB		OBEYJOB(PD_OBEY)
REMOVE	Object file member must have been copied into pdfile after compilation	REMOVE(A_OB1,B_XZ)
REMOVEMACRO	Macros are described in Chapter 16	REMOVEMACRO(MAC_ONE)
RENAME	See "Destroying and renaming members", below; a member can only be renamed within the same partitioned file	RENAME(H_TEST1,H_OLDTEST)
RUN	Object file member must have been copied into pdfile after compilation	RUN(PD1_PROGZ)
SHOW		SHOW(DIRSPECS_CPUT)

Table 4.4: Commands which can Access Members of Partitioned Files

Naming individual members

Each member of a partitioned file must have a name of up to 11 upper case letters or digits, the first of which must be a letter. When an individual member is referenced, the member name is written after the partitioned file name and is separated from it by an underline (_). For example, member LIST1 in partitioned file HOLD could be accessed thus:

        LIST(HOLD_LIST1,.LP)

If the appropriate access permission existed, another user could access it. For example:

        LOOK(ERCC06.HOLD_LIST1)

Creating a member of a partitioned file

The command COPY must be used to create a member of a partitioned file. The first parameter is the name of the file whose contents are to be copied, the second is the name of the member, in the form "pdfile_member". Examples:

        COPY(SOURCE,HOLD_SOURCE77)
        COPY(ERCC27.FILES,HOLD_FILES)
        COPY(ALPHA_ITEM,BETA_ABC)

As indicated by the last example, the file being copied can itself be a member of a partitioned file. However it is not possible to copy a member of a partitioned file into the same partitioned file; thus COPY(A_B,A_C) is not allowed.

If a member of the same name already exists in the specified partitioned file its contents will be overwritten; if not, a new member will be created. Any type of file can be copied in this way, including a whole partitioned file. There is effectively no restriction on the number of members in a partitioned file, although the total size of the file is subject to the installation-defined limit for disc files (see Appendix 5).

Destroying and renaming members

The commands DESTROY and RENAME (Chapter 5) can be used in respect of individual members. Examples are given in Table 4.4 above.

Note that when a member is destroyed, the remaining members are compacted to use the space it occupied. This means that there is no need for an explicit "tidy" operation.

Efficiency considerations

Partitioned files are particularly suited to applications involving many small files. This is because file space is allocated in units of an epage (4096 bytes), which means that a file containing only a few hundred bytes of information contains a high proportion of wasted space. Even files larger than one epage often contain a significant amount of unused space because they too are rounded up to a full epage boundary. A member of a partitioned file, on the other hand, is only as large as it has to be.

Quite apart from the file space consideration, it is often convenient to be able to group sets of files together, and partitioned files provide a possible method. The following points should be noted, however, before embarking on the use of partitioned files:

* There is no significant difference between the cost of reading from a member of a partitioned file and reading from a conventional file containing the same information.

* The cost of adding a new member to a partitioned file is similar to the cost of making a copy of the same file.

* There can be a significant cost in destroying a member of a partitioned file. Note that a member is destroyed either by use of the explicit DESTROY command or as part of COPY if the member being created has the same name as an existing member. This cost will not be significant if the whole partitioned file is only a few epages long, but for a large partitioned file - say more than 100 epages - the cost will be noticeable. Thus partitioned files are less suitable for applications involving frequent replacement.

SS# AND T# FILES

The command OPTION, described in Chapter 17, enables a user to choose certain options
relating to the appearance - to him - of the Subsystem. As a consequence of the use of
OPTION, the Subsystem may create some permanent files on behalf of the user; their names
begin SS#. Currently there are three such: SS#OPT, SS#DIR and SS#JOURNAL. Further
details are given in Chapter 17.

The Subsystem may also create temporary files; their names begin T#. For example, the
name of the listing file produced by the compilers, if none has been specified, is T#LIST.

T# files, unlike SS# files, are destroyed when the user logs off. Furthermore, they are
not normally included in the output produced by the command FILES, whereas SS# files are.

SUBSYSTEM INFORMATION

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

 * the commands HELP and ALERT

 * the EMAS 2900 Information Card

 * Advisory Services

The command HELP

          HELP(subject/.ALL, out)

This command provides on-line information for EMAS 2900 users. If the command is typed
with no first 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(ANALYSE)

would give information about the command ANALYSE.

Apart from commands, there are a number of general headings about which information is
available. These may vary, depending upon the particular EMAS 2900 service used, but they
normally include the following:

          ADVISORY
          INTERRUPT
          SCHEDULE

If the first parameter is given as .ALL then the whole of the current HELP text is output.
Thus
          HELP(.ALL,.LP)

will list the complete HELP text on the local line printer.

The command ALERT

          ALERT(out)

This command provides information about recent changes to the service 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(.LP28)

ERCC EMAS 2900 Information Card

This quick reference information card is intended primarily for users of the EMAS 2900 services operated by ERCC. It provides a list of the currently available Edinburgh Subsystem commands and their parameters, and miscellaneous information relating to the ERCC services. ERCC intend to reprint it at least once each year. Copies are available from ERCC, The King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ.

Advisory Services

The staff of users' local Advisory Services will endeavour to answer questions about the Edinburgh Subsystem and the main programming languages. However, the computing services which offer access to EMAS 2900 for their users will give details of the level of support provided by their Advisory Services, and intending users of EMAS 2900 are recommended to establish first what support is available to them.

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 on 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.

* Commands related to manipulating user data.

* File editing commands.

* Compilers and associated commands.

* Commands associated with directories.

* Commands concerned with running work in background mode.

* Commands concerned with accounts and usage.

* Information commands, and other commands that do not conveniently fit into other categories.

Table 4.5 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); an asterisk appears in the "Output" column if it does.

* A page number in this manual of the main description of the command.

Table 4.6, on the last page of this chapter, gives the Subsystem commands in alphabetical order, with page numbers.

Group	Command	Purpose	Output	Page
General File Utilities	ACCEPT	Transfer file OFFERed by another user		5-4
	ARCHIVE	Mark file(s) for transfer to archive store		5-6
	CHERISH	Mark file(s) for backing up		5-6
	DESTROY	Destroy file(s) in disc store		5-2
	DISCARD	Destroy file(s) in archive store	*	5-7
	DISCONNECT	Remove file from virtual memory		5-3
	FILES	Obtain complete or partial list of files in disc and archive stores	*	5-1
	HAZARD	Remove CHERISH marker(s) for file(s)		5-6
	OFFER	Mark file for transfer to another user		5-4
	PERMIT	Allow other users access to a file		5-4
	RENAME	Change the name of a file		5-2
	RESTORE	Copy a file from archive to disc store	*	5-7
Type Specific File Utilities	ANALYSE	Obtain details of type, contents, access permission, etc. of a file	*	6-1
	CONCAT	Join two or more character files	*	6-3
	CONVERT	Convert a data file to a character file	*	6-3
	COPY	Copy a file	*	6-2
	LIST	List file on output device	*	6-4
	NEWPDFILE	Create new, empty, partitioned file	*	4-4
	SEND	List file on output device and destroy it	*	6-5
Manipulating Data	CLEAR	Break link set up by DEFINE		7-6
	DEFINE	Set up link between logical channel and particular file or output device, or get list of current links	*	7-3
	DEFINEMT	Set up link between logical channel and particular magnetic tape file		10-2
	NEWSMFILE	Create new file to be accessed via store mapping facilities		9-1
File Editing	ECCE	Edit character file	*	8-10
	EDIT	Edit character file	*	8-1

Table 4.5: Edinburgh Subsystem Command Summary
(continued on next page)

Group	Command	Purpose	Output	Page
File Editing (continued)	LOOK	Examine contents of character file	*	8-9
	RECALL	Examine file containing record of interactive terminal I/O	*	8-10
	RECAP	Examine file containing record of interactive terminal I/O	*	8-21
	SHOW	Examine contents of character file	*	8-21
Compilers and associated commands	ALGOL	Compile ALGOL 60 source file	*	11-1
	FORTE	Compile FORTRAN IV source file	*	11-1
	IMP	Compile IMP source file	*	11-1
	LINK	Join two or more object files	*	11-5
	PARM	Set compiler options, or get list of current options	*	11-2
	RUN	Execute program		11-5
Commands associated with Directories	ALIAS	Give alias name to a specified command, or remove all aliases associated with the command		11-10
	INSERT	Insert details of object file in current active directory		11-9
	INSERTMACRO	Insert details of character file containing a macro in current active directory		16-13
	NEWDIRECTORY	Create a new directory file if default size not adequate	*	11-8
	REMOVE	Remove reference to object file from current active directory		11-10
	REMOVEMACRO	Remove details of character file containing a macro from current active directory		16-3
	TIDYDIR	Tidy directory file		11-9
Background Mode	DELETEDOC	Remove job from background job queue	*	16-2
	DETACH	Put job into background job queue	*	16-2
	DETACHJOB	Put job into background job queue	*	16-3
Commands associated with accounting	METER	Print usage information for current session	*	17-2
	PASSWORD	Change foreground and/or background password		17-1
	USERS	Print number of currently active users	*	17-2

Table 4.5: Edinburgh Subsystem Command Summary
(continued on next page)

Group	Command	Purpose	Output	Page
Information and other commands	ALERT	Obtain information on recent changes in the service	*	4-7
	CPULIMIT	Set time limit for each command	*	17-3
	DELIVER	Set text for heading of line printer output, etc., or get current text	*	17-4
	DOCUMENTS	Print information about documents in System queues	*	17-4
	HELP	Get advice on using Subsystem	*	4-7
	MESSAGES	Inhibit or permit messages to the interactive terminal		17-5
	OBEY	Execute a sequence of commands	*	17-5
	OBEYJOB	Execute a sequence of commands	*	16-4
	OPTION	Set Subsystem options, or get list of options in effect	*	17-5
	QUIT	Terminate session	*	17-8
	SETMODE	Set characteristics of interactive terminal		17-9
	STOP	Terminate session	*	17-10
	SUGGESTION	Send suggestion to System Manager		17-11
	TELL	Send message to specified user, immediately or at his next log-on		17-11

Table 4.5: Edinburgh Subsystem Command Summary

Command	Page		Command	Page
ACCEPT	5-4		LINK	11-5
ALERT	4-7		LIST	6-4
ALGOL	11-1		LOOK	8-9
ALIAS	11-10		MESSAGES	17-5
ANALYSE	6-1		METER	17-2
ARCHIVE	5-6		NEWDIRECTORY	11-8
CHERISH	5-6		NEWPDFILE	4-4
CLEAR	7-6		NEWSMFILE	9-1
CONCAT	6-3		OBEY	17-5
CONVERT	6-3		OBEYJOB	16-4
COPY	6-2		OFFER	5-4
CPULIMIT	17-3		OPTION	17-5
DEFINE	7-3		PARM	11-2
DEFINEMT	10-2		PASSWORD	17-1
DELETEDOC	16-2		PERMIT	5-4
DELIVER	17-4		QUIT	17-8
DESTROY	5-2		RECALL	8-10
DETACH	16-2		RECAP	8-21
DETACHJOB	16-3		REMOVE	11-10
DISCARD	5-7		REMOVEMACRO	16-3
DISCONNECT	5-3		RENAME	5-2
DOCUMENTS	17-4		RESTORE	5-7
ECCE	8-10		RUN	11-5
EDIT	8-1		SEND	6-5
FILES	5-1		SETMODE	17-9
FORTE	11-1		SHOW	8-21
HAZARD	5-6		STOP	17-10
HELP	4-7		SUGGESTION	17-11
IMP	11-1		TELL	17-11
INSERT	11-9		TIDYDIR	11-9
INSERTMACRO	16-13		USERS	17-2

Table 4.6: Edinburgh Subsystem Commands (Alphabetical Order)

In Chapter 1 the concept of a file was introduced and the conventions relating to EMAS 2900 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 information and a compiled object file. This chapter describes the file manipulation commands provided by the Subsystem which use the basic file handling facilities provided by Director and which act on all types of file. The different types of file provided by the Edinburgh Subsystem, and the Subsystem commands specific to them, are described either in Chapter 4 or in the chapters following.

The command FILES

> FILES(mask, group, out)

This command is used to obtain a list of files belonging to the user. It takes up to three parameters:

> mask - Each filename in the group specified by the second parameter (see below) is compared with the mask and only those filenames which match are listed. The mask consists of up to three fields, where a field is either a string of explicit characters or the symbol "*", representing any characters; e.g.

ABC	selects file ABC	(one field)
ABC*	selects all files beginning with ABC	(two fields)
ABC	selects all files containing ABC	(three fields)
*ABC	selects all files ending with ABC	(two fields)

> If mask is omitted, all filenames in the group (see next parameter) are selected.

> group - The group of files to be included in the list is determined as follows:

I	files in the disc file store (excluding T# files but including SS# files); this is the default
C	CHERISHed files in the disc file store
H	HAZARDed (uncherished) files in the disc file store
A	ARCHIVEd files; this code may be combined with any of the above

> Three further codes are available which may be combined with the above:

S	single spacing (print file names down the page); printing across the page is the default, with the page width determined by the current setting of the OPTION command parameter ITWIDTH (see Chapter 17)
E	print extra information; causes a one-line summary of each of the user's disc files to be printed, including T# files
P	print information about the user's file index as a whole, including access permissions for .ALL (see "Setting access permissions on files", below)

> Any combination of the letters given above can be specified, in any order. Alternatively the parameter can be specified as "*", which has the same effect as specifying all the letters.

> On output a file name is preceded by "*" if the file is CHERISHed and by "**" if it has been nominated for archiving.

> out - This may be either an output device code (.OUT by default) or an output file name. If a file of the same name already exists it will be overwritten.

Note that, because of the parameter defaults, specifying FILES with no parameters causes a list of the user's disc files to be output on the interactive terminal.

CREATING, RENAMING AND DESTROYING FILES

Files can be created, renamed or destroyed only by their owners. Files are normally
created as a byproduct of certain commands being used. For example, the command EDIT can
create a new character file. If this file is compiled using the command IMP an object
file may be created. If the output from ANALYSE is directed to a file and a file of that
name does not exist, one will be created. The only commands which create files explicitly
are NEWPDFILE (Chapter 4), NEWSMFILE (Chapter 9) and NEWDIRECTORY (Chapter 11).

In general the following rule applies in relation to commands that create files
implicitly:

> 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.

Note: this rule does not apply to NEWPDFILE, NEWSMFILE or NEWDIRECTORY.

The command RENAME

        RENAME(ownfile, newname)

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

    ownfile   is the name of an existing file belonging to the user

    newname   is the new 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.

RENAME can also be used to rename a member of a partitioned file; e.g.

        RENAME(PD_OLDNAME,PD_NEWNAME)

(Partitioned files are described in Chapter 4.)

The command DESTROY

        DESTROY(ownfilelist)

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 to destroy a specified file if it is connected in another user's
virtual memory or if it is on OFFER (see below). Also, if a file is permitted to its
owner with an access permission of N (i.e. no access at all) it cannot be destroyed. This
can be used to protect a file from inadvertent destruction.

The command can also be used to destroy members of partitioned files; e.g.

        DESTROY(PD_MEM1,HOLD_ITEST)

(Partitioned files are described in Chapter 4.)

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 executed, it must be connected in the user's virtual memory. This operation is described in Chapter 1 and in Appendix 1. 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 ANALYSE

* 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 they are invoked:

DESTROY

RENAME

OFFER

PERMIT

SEND

COPY (disconnects the file being copied if it belongs to another user)

The command DISCONNECT

DISCONNECT(filelist/.ALL)

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. The form DISCONNECT(.ALL) can be used to disconnect all but essential process files.

There are several situations in which the command is useful:

* To protect a file. Since the file 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.

  Note that the System automatically ensures that the disc copy of a connected file is kept up to date. There is thus no need to use DISCONNECT for this purpose. This does not mean, however, that the output file specified when a text editor is invoked (see Chapter 8) is kept up to date as the editing proceeds: the text editor operates on a temporary copy, and the specified output file is not involved until the editing session is terminated.

* 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 (48 Mbyte) 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.

Offering a file to another user

        OFFER(<u>ownfile</u>, user)

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.
For example:

        OFFER(ABC,ERCC98)

would offer the file ABC to user ERCC98.

Note that once a file is on offer 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. For example:

        OFFER(ABC)

would revoke the offer of the file made to ERCC98 in the example above.

Accepting a file from another user

        ACCEPT(<u>file</u>, newname)

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 first example above had been WXYZ38 then
the user ERCC98, to whom it had been OFFERed, would type

        ACCEPT(WXYZ38.ABC)

The effect of this would be to transfer the file ABC from user WXYZ38 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(WXYZ38.ABC,NEWABC)

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

SETTING ACCESS PERMISSIONS ON FILES

        PERMIT(<u>ownfile/.ALL</u>, user, mode)

Chapter 1 contains a description of the access permission mechanism. This section
describes the use of the command PERMIT. This command, which can only be used in respect
of one's own files, takes three parameters:

ownfile   is the name of a file belonging to this user. This parameter can be given
           instead as .ALL, in which case the specified access permission is applied to
           all of this user's disc files (i.e. to his whole file index). Files created
           subsequently will also have the specified access permission.

user   is one of the following:

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

mode   is one of the following:

R or null   READ (and EXECUTE).
W   WRITE.
N   No access.
C   Cancels an access permission given previously to an individual
              user (other than the owner) or a group of users.

"mode" can also be specified as a combination of the first two modes, viz. RW
or WR.

Notes

* When a file is created it has default access permissions of RW 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 extra space in the file index
  (see Chapter 1).

* Permission to access a directory file (Chapter 11) does not automatically give
  permission to access object files referenced by that directory file.

* In order to check the access permissions on a file belonging to oneself, use the option
  "P" of command ANALYSE (described in Chapter 6).

* In order to check the list of permissions granted to all one's files, use the option "P"
  of command FILES (described above).

Examples

    PERMIT(ABC)

This permits the file to everyone else with the default access permission READ (and
EXECUTE).

    PERMIT(DOUBLE,ERCC23,W)

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

Multiple permissions

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

    PERMIT(PERTEST)
    PERMIT(PERTEST,Y?????,N)
    PERMIT(PERTEST,ERCC28,RW)

Write permission

The following restriction should be noted in respect of write permission being given to
other users: write permission does not allow a user, other than the owner, to alter the
physical size of a file. (This is not the same as altering the amount of information held
within it.)

COMMANDS RELATED TO BACKUP STORAGE

CHERISH(ownfilelist)

HAZARD(ownfilelist)

As explained in Chapter 1, some files are periodically copied into a backup store; the frequency of this operation is chosen by the System Manager of the EMAS 2900 service. All files which are likely to be difficult to reconstruct in the event of file system corruption should be marked for backing up, 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 is 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. Files created from card or paper tape input, however, are cherished on creation; see Chapter 2.

* 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 disc storage for one of the following reasons:

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

* Because the file has not been used for a significant period of time (see Appendix 5 for details of how long with respect to specific EMAS 2900 services), and it has been moved by the System in order to free space in the disc store. Note that this only applies to files that are CHERISHed: un-CHERISHed files are destroyed if they have not been used during the period of time.

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

The command ARCHIVE

ARCHIVE(ownfilelist)

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

There are a number of reasons for using this command:

* To reduce the charge for keeping files on the System (see Chapter 17).

* To clear space in the file index.

* To dispose of files that are not currently required but may be needed at some later date.

Obtaining a list of files in the archive store

The command FILES described at the beginning of this chapter can be used to obtain a list of some or all of a user's files in the archive store.

Moving files from the archive store to the disc store

         RESTORE(ownfile, date)

The command RESTORE is used to copy a file from the archive store to the disc store. Note that the copy in the archive store is not deleted as a result of this command. The command takes two parameters:

ownfile   the name of the file being restored

date   the date of archiving (optional). This should be typed exactly as it appears
           in the FILES output, i.e. in the form dd/mm/yy.

By default the most recent copy of a file is restored; the date is only needed when an earlier copy is required.

The access permissions of the RESTOREd file are as for a newly created file, i.e. all modes to its owner and none to anyone else. The file is un-CHERISHed.

Examples

         RESTORE(KERN27S)
         RESTORE(IMP907A,23/12/77)
         RESTORE(DATA27,01/02/79)

RESTORE will fail immediately if:

* A file of the same name already exists in the user's 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 reference in the archive index to the requested name, or, if a date is
  used, no reference to a file of the requested name archived on that date.

If the RESTORE command is successfully interpreted then a request is sent to the VOLUMS process to carry out the operation (see Chapter 1). The user can then proceed to give other commands to EMAS 2900. The file is normally recovered from the archive store within 15 minutes.

The RESTORE operation can fail when the VOLUMS process attempts to copy the file to the disc 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.

A message from the VOLUMS process will be typed on the interactive terminal, indicating the successful restoration of the file or a reason for failure. This message will be typed when the user next logs on if he logs off before the file is restored.

Destroying files in the archive store

         DISCARD(controlfile)

The command DISCARD is used to delete files in the archive store. The command takes effect immediately: this means that following a call of DISCARD a call of FILES(,A) will confirm the deletion of the specified files.

DISCARD can be used with no parameter to delete small numbers of files. After typing DISCARD the user will be prompted on his terminal for the name and archive date of each file to be deleted. To terminate the command, reply .END. The following example should make this clear:

```
Command:DISCARD
File date:ACCW370 27/02/77
File date:IROUT21X 22/02/78
File date:.END
```

Notes

* The date must be typed exactly as printed by FILES, i.e. in the form dd/mm/yy.

* Anything following the date on the line will be ignored.

* Each file is deleted from the archive index as soon as its name and date have been read and checked.

DISCARD can alternatively take as its parameter the name of a file containing names and dates of files to be deleted. The file must contain the names of files to be deleted in the format specified above, i.e. each name and date pair on a separate line.

Note that FILES(,A,filename) can be used as a convenient method of generating such a file. Using either the mask facility in FILES or a text editor it is possible to produce a selective list of archived files to be destroyed. An advantage of this method is that it is possible to check the list to ensure that it contains only files which are really unwanted before calling DISCARD.

The following examples should help to explain this:

To destroy all files ending in "Y" in the archive store:

Command:FILES(*Y,A,CFILE)

Command:DISCARD(CFILE)

To destroy all copies of file "MASTER" in the archive store:

Command:FILES(MASTER,A,CFILE)

Command:DISCARD(CFILE)

To destroy all files archived before 1978

Command:FILES(,A,CFILE)

Command:EDIT(CFILE)

(Use the editor to delete the names of all files with dates more recent than 31/12/77, thus leaving in CFILE the names of files archived before 1978)

Edit:E

Command:DISCARD(CFILE)

Note that since the command DISCARD ignores any information following the date there is no need to remove the "number of pages" figure from each line of the file created by FILES.

Chapter 5 describes the Subsystem commands 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.

Table 6.1 shows the available functions, relevant commands and allowed file types:

Function	Commands	Valid File (or PD File Member) Types					
		Character	Data	Object	Directory	Store Map	Partitioned
Provide details about contents	ANALYSE	*	*	*	*	*	*
Copy	COPY	*	*	*	*	*	*
Join together	CONCAT LINK	*	*				
Convert	CONVERT		*				
List on output device	LIST SEND	* *	* *				

Table 6.1: Summary of Type-Specific Commands

COMMAND GIVING DETAILS OF THE CONTENTS OF A FILE

The command ANALYSE

ANALYSE(file, option, out)

This command is used to obtain information about a particular file or member of a partitioned file. The parameters are as follows:

file   The name of the file or member to be analysed. It may belong to this user or, if permitted to him, to any other user.

option   This is used to specify the amount of information required. If it is omitted the minimum information is given, viz. the file's type and length and when it was last altered. If the parameter is given as "*", then all information relevant to the file (or member) type is given.

           Alternatively, one or a group of letters can be given. Their meanings are summarised in Table 6.2, together with the file types to which they relate. A letter is ignored if it is not relevant to the type of the file or member specified in the first parameter.

out   An output device or file (not a member of a partitioned file); default .OUT.

Examples

       ANALYSE(IMPPROG,PHS)
       ANALYSE(PROJECT23B MEMBER1,,.LP10)
       ANALYSE(DIRECTORYAB,*)

Letter	Relevant File Types	Information Given
A	directory	Aliases in the directory file, with the commands for which they are aliases.
E	object or directory	Entry point names; with directory files, does not include aliases, but does associate entries with inserted files.
F	directory	Inserted files.
H	object	History: source file compiled to produce file (and when), object files linked to produce file (and when), etc.
P	all types	Access permissions. If the file does not belong to the requesting user, his access permission to it. Also the number of other users currently using the file.
R	object	References to entry points and data areas external to the object file.
S	object	Structure: relative start addresses and lengths of areas within the file.

Table 6.2: ANALYSE Options

COMMANDS FOR COPYING AND JOINING FILES

The command COPY

COPY(file, ownfile)

This command is used to make a copy of a file. It can be used to copy any type of file, (including directory files, members of partitioned files and complete partitioned files) 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:

file   the name of the file or member to be copied

ownfile   the name of the file or member into which it is to be copied

Examples

        COPY(KERN27,BACKUP)
        COPY(ERCC27.TEST23,TEST23)
        COPY(PROGRAMAB1,PDXYZ_TEMPORARY1A)
        COPY(ERCC27.FILES_DATA1,D22_DATAX)
        COPY(HORSES_GRAYMARE,GRAYMARE1)

Notes

* If a file (not a partitioned 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. COPY will fail if the new copy name is the name of an existing partitioned file. (If destruction of a complete partitioned file is really intended, then an explicit DESTROY command must be used.)

* If no file of the given name exists then a new file will be created, with default cherish status and access permissions. Similarly, if a member of a partitioned file is given as the second parameter, either an existing member of the same name (if there is one) is overwritten, or a new member is created. This is discussed further in Chapter 4.

* If successful, COPY produces a confirmatory message.

The command CONCAT

CONCAT

This command is used to concatenate (up to 64) character files or partitioned file members to create an output character file. It reads the names of the files involved from the primary input device (normally the user's interactive terminal). If it is required to concatenate data files, or a mixture of data and character files, each data file involved should first be converted to a character file (see CONVERT, below).

The filenames or partitioned file member names 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 input file names are being read from the interactive terminal the prompt is "Conc:". The prompt for the output file name is "File:".

Example

```
Command:CONCAT
Conc:ERCC27.LIST
Conc:MINE_INFORMATION
Conc:FINALINFO
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 COPY, except that the output file <u>cannot</u> be a member of a partitioned file.

* The output file cannot have the same name as that of any of the input files.

* If successful, CONCAT produces a confirmatory message.

There is also a concatenation operator "+", which can be used when specifying file parameters for the commands DEFINE (input files only), DETACH, DETACHJOB and LIST, and for the commands which call the compilers: ALGOL, IMP and FORTE.

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 11.

COMMAND FOR CONVERTING A DATA FILE TO A CHARACTER FILE

The command CONVERT

CONVERT(<u>file</u>, <u>ownfile</u>)

The command CONVERT is used to convert a data file into a character file. Both parameters are obligatory:

file   The data file (or member) to be converted.

ownfile   The character file produced from the input file. It <u>cannot</u> be a member of a partitioned file.

Notes

* The output file cannot have the same name as the input file. If a file of the specified new name already exists then its contents are overwritten, but its cherish status and access permissions are preserved. Otherwise a new file will be created, with default cherish status and access permissions.

* If successful, CONVERT produces a confirmatory message.

* It is intended to enhance this command later to allow the user more control over the conversion process, and to include the conversion of character files to data files.

LISTING FILES ON OUTPUT DEVICES

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

The command LIST

        LIST(<u>file</u>, outdev, copies, forms, option)

This command takes an obligatory parameter and four optional ones:

file   The name of the file or member being listed; it can belong to this user or, if suitable access permission has been granted, to another user. Additionally, a concatenation of files or members or both may be given, using the concatenation operator "+"; see the examples below.

outdev   The abbreviated name for the device - see Table 2.2. The default is .OUT (normally the interactive terminal). Local devices or those connected via a communications network can be specified.

copies   This parameter can be used to specify the number of copies to be listed. If given, it must be an integer in the range 1-15. The default is 1.

forms   This parameter can be used when listing files on output devices which permit special forms printing. The parameter is given as a number in the range 0-255. Contact your local Advisory Service, or the relevant RJE terminal operator, for details of the forms codes currently available.

option   If the file or member is a data file or a character file containing format effectors (carriage control characters), then this parameter should be specified as FE; otherwise it is not required.

The LIST command can be used to list character or data files on character or binary devices. If a file is made up of bytes of user data which include newline characters where necessary but is unbroken by any control information, then it can be passed unmodified to an output device. Otherwise it will be converted to this form by LIST. Note that the FE option should be specified if a data file containing carriage control characters is to be listed. (Character files and data files are described in detail in Chapter 7.)

Table 6.3 indicates how the possible combinations are handled by LIST.

File Type	Device	
	Binary (.GP, .BPP, .SGP, .MP)	Character (.OUT, .LP, .PP, .CP)
Character	Unmodified	Unmodified
Data (unstructured)	Unmodified	Unmodified
Data (F format)	Unmodified	Newline characters between records inserted
Data (V format)	Record headers removed	Record headers replaced with newline characters

Table 6.3: Treatment of Various File Types by LIST

Examples

```
LIST(ALIST,.LP)
LIST(MND1136)
LIST(ERCC06.FILELIST,.LP15,,,FE)
LIST(MYPROG+YOURPROG+ERXY99.ANOTHER_PROG,.LP23,2)
LIST(GRAPHOUT,.GP)
```

LIST does not access devices directly, other than the user's terminal. Instead it makes a copy of the file, modifying it as indicated in Table 6.3, and sends it to the SPOOLR process to be listed when the required output device is available. This may be minutes or hours after issuing the command. The command DOCUMENTS (described in Chapter 17) can be used to determine whether any files are still awaiting listing in the EMAS 2900 output queues.

The command SEND

        SEND(ownfile, outdev)

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, unmodified, to the SPOOLR 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.

Note that SEND does not modify the file in any way before passing it on to the SPOOLR process. Thus data files which would require the insertion of newline characters or the removal of record headers cannot be output using SEND. In particular, it is not permitted to SEND files created by FORTRAN programs if they have the default record structure of V1024. These restrictions are summarised in Table 6.4:

File Type	Device Binary (.GP, .BPP, .SGP, .MP)	Character (.LP, .PP, .CP)
Character	OK	OK
Data (unstructured)	OK	OK
Data (F format)	OK	invalid
Data (V format)	invalid	invalid

Table 6.4: File Types Acceptable to SEND

Notes

* SEND uses T#LIST as the default "ownfile", and .LP (line printer) as its default output device; hence

        SEND

(no parameters) causes the default compiler listing file to be output on the line printer and destroyed.

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

* The input cannot be a member of a partitioned file (cf. LIST).

* The input cannot be a concatenation of files (cf. LIST).

* There are no "copies" or "forms" parameters (cf. LIST).

* Unlike LIST, SEND does not start the listing by giving the name of the file.

This chapter introduces the subject of manipulating data in files on EMAS 2900. It
describes the structure of the two types of file used most frequently for this purpose:

* Character files

* Data files

In addition it gives details of the following commands:

* DEFINE - links an I/O channel used in a program to a particular file or device, or
          lists the current channel-file definitions

* CLEAR - clears one or more links set by DEFINE

Chapters 12, 13 and 14 describe the handling of data within the programming languages IMP,
FORTRAN and ALGOL respectively. Chapter 9 describes data manipulation via direct mapping
of files, and Chapter 10 the use of users' magnetic tapes.

CHARACTER FILES

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

* when a text editor (the EDIT or ECCE commands) is used to create a new file or as a
  result of use of the EDIT command F or of the ECCE "secondary output" facility
  (Chapter 8)

* from cards or paper tape read in (Chapter 2)

* when stream output is generated from an IMP or ALGOL program, or from a FORTRAN program
  when the DEFINE parameter C has been specified (see below)

* when compiler listing files are generated

* as optional output files from commands; for example, FILES(,A,OUT) produces a character
  file OUT

* as output from the CONCAT or CONVERT commands, or when a character file is copied by
  use of the COPY command

The information held in a character file consists of a header, which contains current
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 by a
FORTRAN program they are used to divide it into records. There is no restriction on line
length imposed by the structure of the file.

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.

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 themselves 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

* as ECCE input files, primary and secondary

* as control input to the commands DETACH, DETACHJOB, DISCARD, OBEY and OBEYJOB

* as input to the commands CONCAT, COPY, LIST, LOOK, SEND, SHOW and TELL

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, if the DEFINE parameter C has not been specified (see below)

* as IMP or ALGOL sequential or direct access binary files created by OPENSQ or OPENDA

Format of data files

A data file consists of a header followed by one or more records. The records can either be fixed (F) length or variable (V) length.

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 lines of text of variable length, for example, they are inefficient because they then have to contain redundant trailing spaces.

Variable format data files contain records which each start with control information. Apart from the user data each record has a header of 2 bytes of control information.

Data files for input

Data files can be used as any type of input for FORTRAN programs and for binary sequential and direct access files in IMP. Apart from input to FORTRAN programs, they cannot be used where input character files are required. In such cases, however, they can be converted to character files (see CONVERT, Chapter 6). The most frequently occurring situation where this is necessary is when a data file is to be examined by use of a text editor. See also the C format option, in command DEFINE (below).

Carriage control characters

There is a convention in FORTRAN whereby the first character in each record of formatted output can be used to control the line spacing of the output device - for example, line printer or interactive terminal. Such characters are known as "carriage control characters" or "format effectors". There are a number of points to note about them:

* If output from a FORTRAN program has been directed to the interactive terminal or to a line printer (see DEFINE, below), then the first character of each record is assumed by the Subsystem to be a carriage control character and is treated accordingly; see Reference 4 for details.

* If the output is directed to a disc file, the carriage control characters, if present, are not interpreted in any way, but merely treated as part of the user's data in the file. Should it be required to list such a file subsequently, it is necessary to inform the LIST command (Chapter 6) that carriage control characters are present, so that they can be treated as such.

* If a disc file containing character information is to be generated by a FORTRAN program, it is usually advisable to specify via the DEFINE command that a standard character file, rather than a data file, is to be generated (format C - see below). In general a character file is easier to handle than a data file when character information is involved; for example, it can be examined directly by use of one of the text editors. Note however that, as before, carriage control characters are treated as part of the user's data in the file - they are not converted into combinations of newline, newpage and carriage return characters. Thus the remark in the previous note about the LIST command also applies to character files generated with carriage control characters.

THE DEFINE COMMAND

          DEFINE(chan, file/outdev/.IN/.TEMP/.NULL, size, format)
     or   DEFINE(?)

This command is used to establish a link between an Input/Output (I/O) channel and a particular file or output device. Its parameters are described in the following paragraphs. Note that the access method is not specified in the DEFINE command: it is determined from the way in which the program uses the specified I/O channel.

DEFINE(?) causes a list of the current channel-file definitions to be printed.

The DEFINE parameter: chan

This parameter must be specified. It gives the channel number of the definition. It must be a one or two digit integer in the range 1-80. Note that only one definition can exist at one time for a particular channel number. If a channel number specified in a DEFINE command has already been defined then the previous definition is lost.

The DEFINE parameter: file/outdev/.IN/.TEMP/.NULL

This parameter, which must be specified, is used to nominate the file or device to be used. In its simplest form it can be a filename:

          DEFINE(1,DIGDATA27B)

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(18,ERCC28.TRIAL)

Members of partitioned files can be used, but only for input:

          DEFINE(23,TESTINPUT_EXPER171279)

The various output devices available are described in Chapter 2 and in Appendix 5. If an output device is used it must be specified by the appropriate mnemonic:

          DEFINE(37,.LP)

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

          DEFINE(8,.IN)   input from interactive terminal associated with channel 8

          DEFINE(10,.OUT)   output on channel 10 associated with interactive terminal

Input files can be concatenated using "+" characters:

          DEFINE(18,FILE1+FILE2+FILE3)

The files used must all be of the same type and, in the case of data files, must all have the same record format (see below).

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. For example:

    DEFINE(27,ALPHABETA-MOD)

Note that if this facility is used to append output to an existing data file then the format used (see below) is taken from the existing file, not from the DEFINE command.

Temporary and dummy file definitions: .TEMP and .NULL

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 while its definition is valid - that is, until the command CLEAR is used (see below), or DEFINE is used again for the same channel number, or the session ends.

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

    DEFINE(3,.TEMP)
    RUN(CREATE)
        .
        .
    RUN(VALIDATE)
        .
        .
    DEFINE(3,STORECOP)
        .
        .
    RUN(PROG3)

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 to be STORECOP. 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 keyword ".NULL" can be used whilst testing programs. It has the following effects:

* On input, it 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, FORTRAN or ALGOL (Chapters 12, 13, and 14), or in the case of store mapping (Chapter 9).

The DEFINE parameter: size

This parameter can be used to control the size of an output file. It must be an integer in the range 1-1023 and defines the size of the file in units of 1 Kilobyte (1024 bytes). Its precise effect depends on the access method being used and on whether the file is new or old. Table 7.1 below summarises the effect.

Notes

* The default value for the size parameter is 255 (Kbyte).

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

    DEFINE(18,.LP,500)

Access Method	Effect of size parameter	
	New File	Old File
Stream (IMP or ALGOL)	determines maximum size of file for duration of current definition	determines maximum size of file for duration of current definition
Sequential (ALGOL, IMP or FORTRAN)	determines maximum size of file for duration of current definition	determines maximum size of file for duration of current definition
Direct access (ALGOL or IMP)	determines actual size of file	ignored
Direct access (FORTRAN)	ignored - size extracted from DEFINE FILE statement in FORTRAN program	ignored
Store map (IMP)	not allowed - file must exist (see Chapter 9)	ignored

Table 7.1: Effect of DEFINE "size" Parameter

The DEFINE parameter: format

The format parameter, which is only relevant for output files, is of the form C or rn, where:

C   specifies that a character file is to be written (FORTRAN programs only)

r   is the record format, F or V (i.e. Fixed or Variable length record); default V

n   is the record length in bytes; with V, the maximum number of bytes available for user data; with F, the actual number

Output access method	Record format information taken from DEFINE	Note
Stream	No	Character files do not have record format
Sequential	Yes	Existing file format used if file is old and being written to using "-MOD"
Direct access (IMP)	No	IMP DA files always have fixed 1024 byte records
Direct Access (FORTRAN)	No	Information taken from DEFINE FILE statement in FORTRAN program
Direct Access (ALGOL)	Yes	DEFINE information used only if file is new; parameter must be of the form Fn
Store map	No	Store map files do not have record format

Table 7.2: Effect of DEFINE "format" Parameter ("rn" Form)

Table 7.2 is only relevant when the "rn" form is used. This form is required when setting the record format and length for an output data file. It is only relevant for some access methods, and is ignored if an already existing file is being extended using the "-MOD" facility.

The "C" form relates only to output files, to be written by FORTRAN programs. If C is specified, the file created is a character file.

Examples

     F80   Fixed length records, each of 80 bytes

     V200 Variable length records, each containing a maximum of 200 bytes of user data

Note that the default record format for files is V1024. This can be used for almost all applications, and thus it is rarely necessary to specify the format parameter.

Parameter	Position	Default	Contents	Examples
channel	1	None	Channel number	3, 14, 80
file/outdev/.IN/ .TEMP/.NULL	2	None	filename member (input only) device concatenated files and/or members (for input only) temporary file dummy file	ERCC27.HELP15 ABCTE_MEM1 .CP, .GP29 ABC+PD_A+END .TEMP .NULL
size	3	255	file size in Kbytes	500
record format and length/C	4	V1024	record format code and record length C: code for character file from FORTRAN	F80, V137 C

Table 7.3: Summary of DEFINE Parameters

The command CLEAR

     CLEAR(chanlist)

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

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

* with a list of channel numbers, in which case the selected definitions are cleared

Examples:

     CLEAR
     CLEAR(71)
     CLEAR(1,27,36)

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

Also available as part of the Edinburgh Subsystem is the Edinburgh Compatible Context Editor, invoked by use of the command ECCE. ECCE is similar, from the user's point of view, to the Subsystem editor, but differs considerably in detail. It is described in the second part of this chapter, followed by sections describing the commands SHOW and RECAP; these serve the same purposes as LOOK and RECALL respectively, but are based on ECCE.

THE EDINBURGH SUBSYSTEM EDITOR

The EDIT command

        EDIT(file, file)

This command invokes the Edinburgh Subsystem editor, which 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" already exists. 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 itself be altered. "oldfile" can be a character member of a partitioned file. Editor commands used will alter the copy in "newversion". Note that if "newversion" does not exist it will be created, and if it does exist its current contents will be overwritten.

 * EDIT(oldfile, .NULL) - see the command LOOK, at the end of this description.

Method of editing

Editing is accomplished by moving a notional cursor through the file and inserting and removing text with respect to the current position of the cursor (hence the name "context" editor). On entry to the editor the cursor is positioned at the top (beginning) of the file. After each line of edit commands has been typed by the user, one or more lines of the file (depending on the commands given) will be output on the interactive terminal, followed by the prompt "Edit:". By this means the user can determine whether his commands have had the desired effect.

COMMAND STRUCTURE

All editor commands are single letters (upper or lower case). 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 single asterisk can be used instead of an integer to mean "very large positive integer".

* 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. For example:

        /ABC/   ?12*23(A/27)?   .IS
        THIS.   -/together/

If the delimiter character appears in the text of the string it must be typed twice in order to distinguish it from the closing delimiter. For example:

        .A=2..3*(B/PI).

* A filename enclosed in the characters "<" and ">". For example:

        <MYFILE>
        <ERCC06.EDITTEST>

Note that it is permissible to use this form wherever the text string form (above) is allowed, the complete contents of the file being then equivalent to the text string. However, where its usefulness is doubtful, the <filename> form is not mentioned in the descriptions of the relevant commands, below. It is included in the summary table (Table 8.1).

* The single character ' (quote). This is used to specify the same text string as that last given in a use of this editor command. Hence the sequence TM/%END/M1M' is equivalent to TM/%END/M1M/%END/ .

Table 8.1 (later in the chapter) indicates the valid parameter types for each command. Commands can be typed one to a line, or concatenated without separators on a line. For example:

        TP10M/%ENDOF/

Note that a failure in a command within the sequence of commands on a single line 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 P 10 M /%ENDOF/

Commands 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. When text at the beginning of the file is output on the user's terminal, the top of the file itself is represented by "*T*".

   B - Bottom. This command takes no parameter. It moves the cursor to the bottom (end) of the file. When text at the end of the file is output on the user's terminal, the bottom of the file itself is represented by "*B*".

   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 by 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" - 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. For example:

    M-/%BEGIN/

A - After. This command used with an integer parameter alters the position of the cursor by the number of characters specified by the value of the parameter. Hence A3 means move the cursor forward three characters 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, as appropriate.

The command A can also be used with a text string, in which case it has an effect similar to Move, the difference being 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 then 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; for example:

    Edit: M/%EMD/
    ↓*B*
    Edit: HM/%END/
    ↓%END

Command to insert text

I - Insert. This 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. The text string may be of any size and may contain newline characters.

Alternatively "I" can be used with a filename parameter, in which case the complete 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.

A member of a partitioned file may also be specified:

    I<PD1_MEMA>

Commands to delete 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 line and the two lines following
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 <u>and including</u> the first occurrence of the specified text. Hence in a textual file the sequence:

M/Here/D/./

would have the effect of deleting the whole of the first sentence beginning with "Here" after the present position of the cursor (strictly, from the next "Here" up to and including the first "." thereafter). As with Move and After, the delimited text can be preceded by "-" to cause the deletion of text to proceed backwards up the file from the present position.

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 in the file before or after the cursor (as appropriate) then the characters that do exist are removed.

The command 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/Here/R/./ would have the effect of removing only the "." following the first occurrence of "Here". Note carefully the distinction between D/./ and R/./ .

The form R-/text/ is also permitted. It causes removal of the first occurrence of the specified text when moving back up the file from the present position of the cursor.

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. The cursor is normally printed as ↟ or ^, depending on the user's terminal.

In addition the editor command P can be used:

P - Print. This command causes lines to be printed on the terminal. It can take an integer parameter, in which case the effect is to print the specified number of lines; thus 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 also 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.

The form P-/text/ is also permitted. It causes the printing to start at the first occurrence encountered of the specified text when moving back up the file from the

position of the cursor. In this case printing continues until the position of the cursor is reached.

Note that the cursor is <u>not moved</u> by the P command.

The cancel command

8-5

If an error is made in typing editor commands the normal rules apply for deleting characters or a whole line (see Chapter 3). Additionally, the editor command C can be used:

C - Cancel. This command cancels previous 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 - End. E is used with no parameter as the normal exit command. The effect is to return to normal Subsystem command level.

Q - Quit. 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.

Preserving editing done so far

W - Write. This command can be used at any time during editing to "Write" all the editing done so far to the output file. This is a useful way to avoid the risk of losing all one's editing if a System failure occurs before one completes an editing session. Note the following:

* The editing done thus far is consolidated into the output file.

* If Q (see above) is used subsequently, the output file will be as it was left by the W, <u>not</u> as it was before editing commenced.

* The cursor is left at the top of the file (i.e. at "*T*") by this command.

8-5

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:

* repeating 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 were 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". An asterisk, "*", can be used in such cases in place of the integer repetition count. It means "a large number of times" and will cause the command sequence to be executed until a failure occurs, e.g. "*B*" reached.

Another use of the command repetition facility might be to find out the names of the next 10 subroutines in a FORTRAN program. The command sequence to do this might 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 ten times the next line containing "SUBROUTINE". This 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

    S - Separate. S is used to set a separator before the present position of the cursor. It is output as "*S*" on the interactive terminal. 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 set by S:

    T - as before, this moves the cursor to the top of the file.

    B - as before, this moves the cursor to the bottom of the file.

    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, TM*, 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 separator can be removed from the file by use of the command K:

    K - Kill. K takes no parameter and leaves the cursor at the point previously occupied by the separator.

Note that the command string KS causes the cursor to be placed immediately after the separator, without changing the position of the separator.

Moving a section of text within a file

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

 * setting a separator at the <u>destination</u> of the text, 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 (see below)

 * removing the separator with the command K

The commands S and K are described above. The command U is used to move a piece of text within the file:

> U - Uproot. 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.

> The form U-/text/ is also permitted. In this case the text moved extends from the first occurrence encountered of the specified text when moving back up the file from the cursor position, down to the cursor position.

> 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 rest of program
%ENDOFPROGRAM
*B*
```

> Editing dialogue:

```
Edit:M/%ROUTINE A/S          /set separator
*S*
↑%ROUTINE A
Edit:M/%ROUTINE B/U/%END      /move text, up to and including %END
/:/                          /followed by a newline
↑ text of rest 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 rest of program
%ENDOFPROGRAM
```

Extracting part of a file

    F - File. The command F is used to copy part of a file into another file or to an
    output device. The parameter with F must be of the form

          <filename>
       or <output device>

    For example: F<EXTRACT>, F<.LP23>, F<OUT-MOD>

    The abbreviations used for output devices are given in Chapter 2. The text which is
    copied 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 contents of the file will be overwritten, unless the "file-MOD"
    form (see example above) is used, when the text is appended to the existing contents of
    the file. If the file does not exist, a new file will be created. In all cases a
    character file is produced.

F 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 B
     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 its
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 its 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 or W is executed, 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.

Notes:

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

  * Since there are pointers to the top and the bottom of the file, moving the cursor to
    these points with T and B is efficient. When relevant, 0 is also an efficient command.

  * Searching backwards for text using M-/text/ or M-n (or A-/text/ or A-n) is considerably
    slower than searching forwards.

COMMAND SUMMARY

Table 8.1 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.

Command	Use	Valid Parameters						
		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	Uproot		*	*	*	*	*	
W	Write	*						

Table 8.1: Summary of Subsystem Editor Commands

THE COMMAND LOOK

        LOOK(file)

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 T#LIST, the default compiler listing file (see Chapter 11).

A similar effect can be achieved by typing

        EDIT (filename,.NULL)

There are three differences between using LOOK and EDIT:

 * The editor commands D, G, I, R, U and W are not allowed.

 * The prompt at editor command level is "Look:" rather than "Edit:".

 * The parameter for LOOK has a default of T#LIST.

Otherwise the facilities available are identical.

THE COMMAND RECALL

RECALL

This command is used to interrogate the file containing a copy of all interactive terminal Input/Output operations for this user; see the parameters NORECALL, TEMPRECALL and PERMRECALL of command OPTION (Chapter 17). RECALL takes no parameter. There are three differences between RECALL and EDIT:

* The editor commands D, G, I, R, U and W are not allowed.

* 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:" rather than "Edit:".

THE EDINBURGH COMPATIBLE CONTEXT EDITOR

The Edinburgh Compatible Context Editor (ECCE) was developed by members of staff of the Department of Computer Science, University of Edinburgh, in particular Mr Hamish Dewar (now of Thistle Computers Ltd.). The following description is based on a document produced by the Department.

Calling the editor

The editor can be used to manipulate any Edinburgh Subsystem character file.
There are three ways in which the ECCE command can be used:

Command:ECCE(oldfile)   to amend an existing file

Command:ECCE(oldfile, newfile)   to create (or overwrite) "newfile" with the edited
                                     "oldfile", leaving "oldfile" unaltered

Command:ECCE(,newfile)   to create (or overwrite) "newfile"

All the files specified in the above forms of calling ECCE can be members of partitioned files; so also can the secondary input and secondary output files (see the sections "Secondary input" and "Secondary output", near the end of this chapter).

COMMANDS

After invoking ECCE, the user issues commands from his terminal, one to a line or many to a line, and in either upper or lower case. Spaces have no significance, except within text strings, and command lines are terminated either by a newline character or by a semi-colon (;). After reading the command line the editor checks it for syntactic correctness and then executes the individual commands in left to right order. A syntax error in a command line (such as an unmatched string delimiter) causes the whole command line to be ignored and an error report to be produced (see below: "Command failure").

To terminate the editing session and close all input and output files, the command %C is issued. In this case %C must be the only command in the command line.

The file pointer

Editing commands (except the "special" commands such as %C) operate on the source text at the "current position" in the text. In order to define the current position, the editor maintains a pointer, henceforth called the "file pointer". Conceptually this is always between two characters of the source text, or immediately before the first character of the file or immediately after the last character in the file. When printing out a line on the user's terminal the editor identifies the file pointer by typing an up-arrow or caret character in the appropriate position unless this is at the beginning of a line.

Example:

        This is an up-ar↑row or caret character.
        But here the file pointer is at the start of the line.

For convenience, the end of the file is treated as an extra line following the last line
in the file. This extra line is typed out as **END**. For example:

        This is the last line of my file.
        END

By default ECCE types out the current line (i.e. the line containing the file pointer)
after the execution of each command line, unless the line was typed out in response to the
immediately preceding command line and the file pointer was not moved (forwards or
backwards) across a line boundary by the latest command line. Thus, in all cases the last
line printed on the user's terminal is the line containing the file pointer.

A simple subset of commands

In this section a simple subset of ECCE's facilities will be described. This is
sufficient to perform most editing operations. More sophisticated commands and programmed
commands will be described later. The commands described allow the file pointer to be
moved forwards and backwards in the source text, a line of source text to be killed
(removed), a line of text to be inserted into the file, occurrences of a specified text
string to be found, lines of text to be printed, and the editing session closed.

All these commands (except %C) can be followed by a positive integer, zero (0), or *, to
specify how many times the command is to be repeated. Zero and * denote indefinite
repetition, i.e. the command is executed repeatedly until it fails or until a large number
of repetitions (currently 5000) have been performed.

%C   Close the editing session. The input and output files are closed and control
            returned to command level. %C must be the only command in the command line.

M   Move the file pointer to the start of the next line. This fails if the file
            pointer is already at the end of the file (that is, after the last line of the
            file).

M-   Move the file pointer back to the start of the previous line. This fails if
            the current line is the first line of the file.

K   Kill the current line. That is, remove the whole of the current line from the
            file. The file pointer is left at the beginning of the next line (as for M).
            This command fails if the file pointer is already at the end of the file.

G   Get a line of text from the user's terminal and insert it between the end of
            the previous line (if any) and the start of the current line. The file pointer
            is left at the start of the current line (that is, immediately after the new
            text). The user is prompted for input with a colon (:) and the command will
            fail if the first character of the input line is a colon, in which case the
            file pointer is not moved. The rest of the line after the colon (if present)
            is treated as a command line. The form G* can be used when typing in an
            indeterminate number of lines; see the example below.

F/TEXT/   Find TEXT. Search the rest of the file, starting at the file pointer, for the
            first occurrence of the string TEXT. A string is any sequence of characters,
            excluding newline, enclosed in quotes. Any symbol other than those which have
            a defined significance in the command language may be chosen to represent the
            quotation mark, for example / ' " # or .. The file pointer is left immediately
            before the first occurrence of TEXT if TEXT is found. Otherwise the command
            fails and the file pointer is left at the end of the file. If the command is
            repeated, then the occurrence of TEXT just found is ignored in the subsequent
            search for TEXT (see below: "Text location and manipulation commands").

P   Print the current line at the user's terminal. If the multiple form of P is
            used, for example P6 or P*, then a move (M) is performed after the first and
            subsequent P's (but not after the final one). In this way a sequence of lines
            may conveniently be printed at the user's terminal. The simple command P can
            never fail and does not move the file pointer. The multiple form of the Print
            command (for example P10) fails only if an implicit attempt is made to move

beyond the end of the file (for example, trying to print 10 lines when the current line is only 8 lines from the end of the file), and always leaves the file pointer at the start of the last line printed (or at the end of the file).

The P command identifies the file pointer by typing a caret or up-arrow character in the appropriate position, unless this is at the start of a line. The end of the file is identified as **END**.

It should be noted that the multiple form of P is not strictly repetition, for the effect of P followed by P is to print the current line twice, whereas the effect of P2 is to print the current line and the next line, and move the file pointer to the start of the next line. Generally this does not cause any problems or confusion.

An example of the use of ECCE

In this section ECCE is used interactively to create and update a file of text, which will contain part of a poem by Roger McGough called "Discretion". The complete dialogue between ECCE and the user is displayed. The user is prompted for command lines with a ">" character and for input with a ":". Starred lines are ECCE's normal (default) monitoring output (see "Monitoring commands", below). Note that "*" is not printed at the user's terminal, and that ">" and ":" are not inserted into the source text. Explanatory comments occur in the right hand half of the line starting with a "/" character.


```
        Command:ECCE(,POEM)             /invoke the editor to make a new file
        Edit
        >g*                             /get some lines of input
        :Discretion
        :Discretion is the better part of Valerie
        :(though all of her is nice)
        :lips as warm as strawberries
        :eyws as cold as ice
        :the very best of everything
        :only will suffice
        :not for her potatoes
        :and pudding made of rice
        :
        :Not for hwr potatoes
        :and puddings made of rice
        :she takes carbohydrates like God takes advice
        ::                              /end the input for now

        >m-9                            /move back 9 lines
      * eyws as cold as ice
        >k g                            /kill it and get a replacement
        :eyes as cold as ice
        >f/pudd/                        /missed the "s" off the end
      * and ↑pudding made of rice
        >k                              /delete the line
      *                                 /current line after deletion
        >g                              /get a replacement
        :and puddings made of rice
        >f/hwr/
      * not for ↑hwr potatoes
        >k g                            /kill it, and replace it
        :not for her potatoes
        >m*                             /move to end of file
      * **END**
        >g*                             /and continue the input ...
        : ...
```


Text location and manipulation commands

It is apparent from the above example that the use of only the very limited subset of commands introduced so far is clumsy, tedious, and error prone. The commands described below can be used to move the file pointer to immediately before or after a specified text string, and to insert, substitute, or delete specified strings of text, rather than whole lines.

In many of the commands introduced in this section a search for a text string is implied. This search always begins at the character immediately following the file pointer and fails, with the exception of Find, if the end of the line is encountered before the text is found. Find fails only at the end of the file.

In successive searches for a text string by Find or Uncover, the occurrence just found is ignored. By contrast, the Delete command will operate on the occurrence of text just found.

In all the following examples the "/" character is used as a string delimiter. Any character with no other significance to the editor could be used, for example ' or " or $ or . (or / itself).

Note the effect on the text location commands of %U and %L (described in "Upper and lower case control commands", below).

F/TEXT/   Find TEXT. Search the rest of the file, starting at the file pointer, for the
           first occurrence of TEXT. The file pointer is left immediately before the
           first occurrence of TEXT if it is found. Otherwise the command fails and the
           file pointer is left at the end of the file. If the previous command was a
           text location command (F, V, or U, see below) then the occurrence of TEXT just
           located is ignored when searching for TEXT. This applies particularly to the
           case of F's being repeated; for example, F/TEXT/5 finds the 5th occurrence of
           TEXT after the current position of the file pointer.

S/STR/   Substitute STR for TEXT. If the previous command was F/TEXT/, U/TEXT/ or
           V/TEXT/ (see below) then delete TEXT and insert STR. The file pointer is left
           immediately to the right of STR. The command fails if the previous command was
           not an F, U or V. Note that STR may be the null string, so that F/ABCD/S// ·
           deletes the first occurrence of ABCD.

T/TEXT/   Traverse TEXT. Search the current line, starting at the file pointer, for the
           first occurrence of TEXT and move the file pointer to immediately after it. If
           TEXT is not found on the current line the command fails. In this case the file
           pointer is not moved. Traverse is rather like Find except that the file
           pointer is left after the occurrence of TEXT, but note that it is not a text
           location command (Substitute cannot be used after it). Traverse is useful in
           many circumstances: for example, when adding an "s" (or anything else) to the
           end of a word.

D/TEXT/   Delete TEXT. Search the current line, starting at the file pointer, for the
           first occurrence of TEXT, and delete it. The command fails if TEXT is not
           found before the end of the current line. If the command succeeds then the
           file pointer is moved to the position previously occupied by TEXT; otherwise
           the file pointer is not moved.

I/TEXT/   Insert the specified TEXT immediately before the file pointer. The file
           pointer is left immediately after the inserted TEXT. Insert will fail if the
           line length is already greater than 160 characters. If Insert fails then the
           file pointer is not moved.

U/TEXT/   Uncover TEXT. Search the current line, starting at the file pointer, for the
           first occurrence of TEXT and remove all characters between the file pointer and
           the start of TEXT (TEXT itself is not removed). The file pointer is left
           immediately to the left of TEXT. If TEXT is not found on the current line then
           the command fails and the file pointer is not moved. If TEXT is found, it may
           then be replaced by using the S/STR/ (Substitute) command.

V/TEXT/   Verify that TEXT occurs immediately to the right of the file pointer. Fail
           otherwise. If successful, S/STR/ can be used to substitute STR for the TEXT
           just verified (see above for S). This command is of use in programmed commands
           (see below: "Programmed commands").

The following combinations of commands are often found to be useful:

        F/TEXT1/ S/TEXT2/   /applies to rest of file
        D/TEXT1/ I/TEXT2/   /applies to rest of line
        T/word/ I/ending/   /applies to rest of line
        U/./   /delete rest of sentence (strictly, up to next full stop)
        U/./S/,/   /delete rest of sentence and change full stop to comma

A further example of the use of ECCE

In the earlier example of the use of ECCE only a very basic subset of commands was used.
In this section the same example is reworked to show how the text location and
manipulation commands, just introduced, can be used to make the necessary changes more
easily.

The comment and monitoring conventions, and the command and input prompts, are exactly the
same as in the previous example.

```
     Command:ECCE(,POEM)            /invoke the editor to make a new file
     Edit
     >g*                            /get some lines of input
     :Discretion
     :Discretion is the better part of Valerie
     :(though all of her is nice)
     :lips as warm as strawberries
     :eyws as cold as ice
     :the very best of everything
     :only will suffice
     :not for her potatoes
     :and pudding made of rice
     :
     :Not for hwr potatoes
     :and puddings made of rice
     :she takes carbohydrates like God takes advice
     :a surfeit of ambition
     :is her particylar vice
     :Valerie fondles lovers
     :like a mousetrap fonles mice
     ::                             /end the input for now

     >m-0                           /move back to start
   * Discretion
     >f/eyws/ s/eyes/               /correct first mistake
   * eye↑s as cold as ice
     >f/pudding/t/g/                /missed the "s" off the end
   * and pudding↑ made of rice
     >i.s.                          /so put the "s" back in
     >f/hwr/ d/w/                   /remove the mistake
   * not for h↑r potatoes
     >i/e/                          /and correct it  .
     >f/y/                          /look for next blunder
   * she takes carboh↑ydrates like God takes advice
                                    /not there yet!
     >f/y/                          /so repeat the command
   * is her partic↑ylar vice
     >d/y/ i/u/                     /fix the error
                                    /N.B.  D is not a text location command
                                    /so "y" just Found is deleted

     >f/nle/ s/ndle/
   * like a mousetrap fondle↑s mice
     >%c                            /end the edit

     Command:                       /next command prompt from the Edinburgh Subsystem
```

Command failure

A command can fail in either of two ways. First it may be syntactically incorrect, in
which case the whole command line is ignored and an error report is produced. Secondly it
may fail in execution, in which case a failure report is produced, the current line (at
the time of the failure) is printed, and the rest of the command line is ignored.

In general the failure of a simple command leaves the file pointer unmoved; however if the
failing command is part of a command line or repeated command then the file pointer is
left positioned by the last successfully executed command.

Syntax errors include commands that are not recognised, mismatched string delimiters,
mismatched parentheses (see below: "Programmed commands"), command line size exceeded
(currently the limit is 40 simple commands), and so forth. For example:

```
      >w
      W?                             /unknown command
      >f/hello.
      TEXT FOR F?                    /mismatched string delimiters
      >mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm
      SIZE?                          /command too long
      >g/abc/
      G /?                           /wrong syntax entirely
but
      >wp2;m                         /W (and rest of command line) is ignored
                                     /but M (next command line) is executed
```

Examples of execution failure:

```
      **END**                        /at end of file
      >m                             /can't, so it will fail
      FAILURE: M
      **END**
and
      >m                             /move to next line
      How now brown cow.
      >s/horse/ m                    /meaningless - S can only follow F, U or V
      FAILURE: S'horse'
      How now brown cow.             /note, Move not executed
or
      >p                             /print current line
      How now brown cow.
      >t/now/2                       /note that "now" occurs only once
      FAILURE: T'now'
      How now↑ brown cow.            /second T/now/ not possible
or
      >g3                            /get 3 lines
      ::                             /didn't mean it, so get out of G
      FAILURE: G
      ... current line ...           /echo of current line on error
```

Note that indefinitely repeated commands such as g* or m* produce no failure reports, but that the failure condition is used to terminate the repetition.

Character manipulation commands

Of the commands met so far, M and M- move the file pointer past a whole line and K deletes a whole line. The next four commands operate on individual characters within a line.

R   Right shift the file pointer one character position. This fails if the file
           pointer is already at the end of the current line.

L   Left shift the file pointer one character position. This fails if the file
           pointer is already at the beginning of the current line.

E   Erase the character immediately to the right of the file pointer. This fails
           If the file pointer is already at the end of the current line.

E-   Erase the character immediately to the left of the file pointer. This fails if
           the file pointer is already at the beginning of the current line.

C   Causes Case inversion of the character to the right of the file pointer, then
           shifts the pointer one position right. Hence command C2 causes ↑Ab to become
           aB↑. If the character is not a letter it is left unaltered but the shift right
           still takes place. The C command fails if the file pointer is already at the
           end of the current line.

C-   Causes Case inversion of the character to the left of the file pointer, then
           shifts the pointer one position left. Hence command C-2 causes cD↑ to become
           ↑Cd. If the character is not a letter it is left unaltered but the shift left
           still takes place. The C- command fails if the file pointer is already at the
           start of the current line.

It is useful to note the effect of indefinite repetition (*) with these commands:

R*	/move the file pointer to the end of the current line.
L*	/move the file pointer to the start of the current line.
E*	/erase from the file pointer to the end of the current line.
E-*	/erase from the start of the current line
C*	/invert the case of the text from the file pointer to the end of the current line.
C-*	/invert the case of the text from the start of the current line to the file pointer.

Breaking and joining lines

The following commands are used to break a line into two parts, and to join two lines together.

B   Break the current line into two parts at the file pointer. That is, insert a newline character immediately to the left of the file pointer, so that the second part becomes the new current line. B never fails. Breaking a line at its beginning (that is, when the file pointer is at the start of the line) has the effect of inserting an empty line immediately before it (which may also be achieved by means of G, then replying carriage return to the G prompt).

J   Join the current line and the next line by removing the newline character at the end of the current line. This command fails if the current line already exceeds 120 characters in length. If the command succeeds then the file pointer is left at the join. Otherwise the file pointer is not moved.

The repetition command

A command line consisting solely of a number or * causes the previous command line to be executed the specified number of times. For this purpose repetition commands do not count as command lines, so that it is the most recent non-repetition-command command line that is repeated. Zero (0) or * causes the previous command line to be executed until failure. For example:

```
>M                      /move to the next line
>3                      /move 3 more times (if possible)
>2                      /move twice more (not 6 times)
>m-2                    /move back two lines
>4                      /now back 8 more (not 4)
>*                      /all the way to the start of the file
>P3                     /print the first 3 lines of the file
The cat
sat on
the mat. Clever cat.
>                       /ready for next command
```

Note that this command sequence was executed in quiet mode (see below: Monitoring commands) and that ">" is ECCE's command prompt.

Monitoring commands

There are three special commands that are used to change the amount of "echoing" that ECCE performs. These have no effect on the source text. Special commands must appear as the only command in the command line.

%M   Monitor normally. This is the default when ECCE is first invoked. The current Tine of the source text is typed at the user's terminal after the execution of each command line unless two conditions hold true. First the current line has just been typed out in response to the immediately preceding command line, and secondly the current command line did not cause the file pointer to be moved across a line boundary at any stage in its execution. The current line is not re-echoed if the last command in the latest command line was a Print command.

%F   Full monitoring is turned on. The current line is typed out after the
            execution of each command line, unless the last command executed was a print
            (P) command. Thus at least one line is output on the user's terminal following
            each command line.

%Q   Quiet mode. Turn off all monitoring. The current line is typed out only if
            explicitly requested by means of the print (P) command, or if the command line
            fails.

Upper and lower case control commands

There are two special commands which affect the operation of the text location and
verification commands (i.e. D, F, T, U and V). As before, a special command must appear
as the only command of the command line.

%U   Upper case. The user is considered to be at an upper case terminal, and so
            cannot distinguish upper case text from lower case text in the file being
            edited. For this reason, the case of the text scanned by a text location or
            verification command is not regarded as significant; thus, for example, F/ABC/
            and F/abc/ are treated as entirely equivalent, and either would match "abc", or
            "ABC", or "Abc", or "AbC", etc., in the file. %U is the default.

%L   Lower case. With %L selected, the case of the string being sought is
            significant and must be matched exactly. %L must be specified if required - %U
            is the default.

Context specification - D, F, T, and U revisited

So far D, F, T, and U have been presented with their default contexts only. However, the
number of lines to be searched in a text location command may be specified explicitly by
inserting a number between the command letter and the text. This makes it possible to
limit the scope of Find (by default the rest of the file), and extend the scope of Delete,
Traverse and Uncover (by default the rest of the current line). For example:

        D50/DOG/   /search 50 lines at most
        F10/HELP/   /search 10 lines at most
        T3/END/   /search 3 lines
        U*/./   /search rest of file

The effect of specifying a context is best explained with reference to a specific command,
for example, D50/DOG/. This operates in the following way:

    1) Search the current line for DOG.
    2) If found then Delete DOG and terminate the command successfully.
    3) Has step 1 been executed 50 times?
    4) If so then command fails.
    5) Otherwise Move to next line and go back to step 1.

If this command succeeds the effect is to delete the first occurrence of DOG within the
next 50 lines (current line + next 49 lines). If it fails the effect is M49 (not M50).

The other examples above have the following effects respectively: Find the first
occurrence of HELP within the next 10 lines (or M9 if HELP doesn't occur within the next
10 lines), Traverse the first occurrence of END within the next 3 lines (or M2 if END does
not occur within the next 3 lines), and more dangerously, Uncover all text up to the next
"." (or Kill the rest of the file if "." does not occur before the end of the file). Note
that this final command U*/./ is very dangerous, as all the text examined is Killed.

These commands may also be repeated. For example:

        F15/HELP/2

This command seeks the string HELP within the rest of the current line and the next
fourteen lines, and then, if successful, ignores the occurrence just found and seeks HELP
again within the rest of the new current line and the fourteen lines following it. The
command will fail if any repetition of F15/HELP/ fails, in which case the effect is either
M14 if the first search fails, or M(between 14 and 28) if the first succeeds and the
second fails. If the command succeeds the file pointer is left immediately before the
second occurrence of HELP.

Programmed commands

It will have been noticed that several commands may be put on one command line to create a command "program". Furthermore, command lines may be repeated a specified number of times, or until failure. This section describes how more general "programs" can be written.

( )   Bracketing. A string of commands bracketed together is treated as one command for purposes of failure and repetition. For example:

           (M I/   /)*

       inserts four blanks at the start of each line of the rest of the file.

?   Optional execution. Any command failure condition is ignored. For example:

           D/PHONE/?

       deletes the word PHONE from the current line if it occurs on the current line. Otherwise it does nothing; in particular it does not fail.

\   Inverted failure (success) condition. A command followed by \ instead of a repetition number has its failure condition inverted, i.e. it succeeds if and only if it fails! For example:

           (MV/%R/\)0

       This programmed command moves the file pointer to the start of the next line then repeats if V/%R/ fails. That is, the move is repeated until a line beginning with %R is found or until end of file is reached.

, (comma) Alternative command sequences. Sequences of commands separated by commas form alternatives. If the first alternative fails the next is tried, and so on. If an alternative succeeds then the remaining alternatives are ignored. This allows a generalised IF-THEN-ELSE construct to be programmed. For example:

           D/CAT/,D/DOG/,M

       This command either deletes CAT or DOG on the current line, or moves the file pointer to the start of the next line. (Try to Delete CAT. IF unsuccessful THEN try to Delete DOG. IF still unsuccessful THEN try to Move to next line.) Note too that command lines may be broken immediately after a comma and thus span two or more physical lines. For example:

           >D/CAT/,
           D/DOG/,M

       Note that one must be very careful when using commands with alternatives. For example:

           MD/CAT/,MD/DOG/

       would not have anything like the effect of the previous example. In this case we move to the next line, and then if D/CAT/ fails we move to the next line and try D/DOG/. If CAT and DOG occurred on consecutive lines and the command were started on a line containing CAT (next line contains DOG) then no occurrences of CAT or DOG would be deleted.

By judicious use of ( ) , ? \ and repetition, quite complex editing sequences can be programmed. For example:

           (d/cat/ i/chat/ 1* , d/dog/ i/chien/ 1* , m)*

This "program" translates the words "cat" and "dog" to their French equivalents (but note that "catalogue" gets translated to "chatalogue"!). Note also the following useful commands:

           (RM)0   /find the next empty line
           (RO(LD/ /)OM)0   /remove trailing spaces from all lines
           ((I/   /M)60B6)* /right shift all lines by four spaces
                              /and separate pages with 6 empty lines

This final command is useful for paginating a file before printing it on a line printer, but note the assumption of 66 lines per page, and no carriage return characters in the file.

Note that "*" and "0" (indefinite repetition) are entirely equivalent.

Macros

Three "macro" commands can be defined by the user, namely X, Y, and Z. When invoked, the effect of a macro is <u>exactly</u> as if the macro text had been typed instead of the macro letter. For example:

>%X=(RM)*   /note: %X=macro text, <u>not</u> X=macro text

This defines X to be the next-empty-line command (see above). The effect of X (or x) in a command line is then exactly the same as (RM)*. Macro text length is restricted to 63 characters. In addition, both macro texts and command lines may contain no more than 40 command units, where each comma, bracket, \, number, and simple command counts as one unit.

For example, if X were to be defined as MK, then

    X4   is exactly equivalent to

    MK4   and not to

    (MK)4   as might be imagined. But note that

    (X)4   <u>is</u> equivalent to

    (MK)4

A macro can be defined as several command lines separated by semicolons. For example:

%X=f1/TEXT1/;f1/TEXT2/

%X=%s;f/%ROUTINE/nf/%END/ma;%s

The latter example involves secondary input, which is described in the following section.

SECONDARY INPUT

Secondary input is a feature that allows parts of a second input file to be copied into the main file. The secondary input file is completely unchanged by any editing operations that might be performed upon it.

To enter secondary input mode the command

    %S=filename

is issued. This causes ECCE to prompt for commands with ">>" rather than ">", and the editing commands thenceforth operate on the file specified (the "secondary input file") rather than on the primary file.

To leave secondary input mode a %S command without the "=filename" part is issued and ECCE then prompts for commands with ">" again.

Subsequent %S commands cause a switch to the secondary input file, or back to the primary file, as appropriate. The prompt used by ECCE indicates which file is being operated on.

A subsequent %S=filename command, where the filename specified is different from the previous one, causes a change of secondary input file, and ECCE goes into (or remains in) secondary input mode.

Note that when ECCE is invoked the secondary file is defined by default to be the same as the primary file. Thus if in the first use of %S no filename were specified, the <u>original</u> version of the primary file (i.e. prior to the current edit session) would be used.

A secondary input file must:

    * exist

    * be a character file (or a character member of a partitioned file)

    * be accessible to the ECCE user

Otherwise a failure message will be output when the %S=filename command is issued.

A %S or %S=filename command must be the only command in the command line.

A %C command issued in secondary input mode has the same effect as in primary input mode, and ends the editing session in the same way.

While in secondary input mode ECCE maintains a second file pointer to the secondary input file, and editing commands cause this second file pointer to be moved. The main file pointer remains at the point it was at when the %S command was issued. If secondary input is re-entered then the secondary file pointer is where it was when secondary input mode was last left. Initially the secondary file pointer is at the beginning of the secondary input file.

Parts of the secondary input file can be transferred either to the main file or to a secondary output file (see "Secondary output", below). This is done by means of two commands: Note and Abstract, which can only be used in secondary input mode. The effect of Note is to note the current position of the (secondary) file pointer. This overrides any previously issued Note commands. The secondary file pointer can then be moved to the end of the section of the file which is required. The effect of an Abstract command is then to copy all text from just after the last Noted position to just before the current position of the (secondary) file pointer, either into the main file in a position just before the (main) file pointer, or to the end of the secondary output file if one is currently defined (see below).

The Abstract command may be repeated, in which case another piece of text is abstracted, starting at the same Noted position as the previous abstracted text; it follows that at least some of the text in the secondary file would be extracted twice by this operation.

Note that when an Abstract command is issued, the secondary file pointer must be <u>after</u> the last Noted position, otherwise it fails.

Only a subset of ECCE commands may be used in secondary input mode: in particular, commands which insert or delete text are prohibited. The simple commands allowed are F, L, M, M-, P, R, T, V, X, Y, and Z (plus A and N of course). Monitoring and macro commands and programmed command qualifiers are also allowed.

Note that in general the size of the main file may not be increased by more than 16384 characters in a single editing session. It is possible to reach this limit using the secondary input facility.

SECONDARY OUTPUT

A command of the following form can be given:

        %O=out

where "out" is a filename or output device. It is then defined to be the "secondary output file". If a file of the specified name already exists then its old contents are overwritten; otherwise it is created. It is <u>not</u> valid to specify a secondary output file as %O=file-MOD.

The effect of this command is that text selected from the secondary input file by the use of the N and A commands is copied to the secondary output file, not to the main file.

A subsequent %O=out command causes the current secondary output file to be closed and a new one to be defined.

The command %O (i.e. no secondary output file defined) causes the current secondary output file to be closed, and the main file to be reselected for receipt of text abstracted from the secondary input file.

Examples

```
        >%0=.LP23                      /define secondary output
        >%s=MISCIMPS                   /enter secondary input mode
        >>f/READ TAG/                  /move to position
        %ROUTINE ↑READ TAG(%INTEGER NAME)
        >>1*                           /to get to start of line
        >>n                            /note the position
        >>(v/%END/\m)0m                /move past end of routine
                                       /blank line following %END statement in file
        >>a                            /abstract the routine
        >>%s                           /and back to main source
        >                              /usual command prompt
```

The effect of this example would be to list the source of routine READ TAG on line printer 23. If the %0=.LP23 command at the start had been omitted, the source of the routine would instead have been copied into the main file, just before the main file pointer.

The dialogue of the above example might be continued as follows:

```
        >%0                            /close secondary output file
        >m-*                           /move to start of main file
        %BEGIN                         /first line of file
        >%S                            /switch to secondary input (file MISCIMPS)
                                       /current line (blank) in secondary input file
        >>m-*nf'!END'                  /Note start of file, then seeks text '!END'
        !END SPECS                     /Text found
        >>a                            /Abstract text. Note that the file pointer is at
                                       /the start of the line, so the current line is not
                                       /itself abstracted.
        >>%s                           /switch back to the main file.
        %BEGIN                         /current line in the main file
        > ...                          /continue editing main file
```


THE COMMAND SHOW

        SHOW(file)

The SHOW command is used to call the editor for the purpose of examining a file rather than altering it. It takes a single parameter - the name of the file to be examined, with a default of T#LIST, the default compiler listing file (see Chapter 11). For example:

        Command:SHOW(TEST)

A number of standard ECCE commands are precluded within SHOW, in particular those which attempt to alter the file. The following basic commands are allowed:

        A, F, L, M, M-, N, P, R, T, V, X, Y, Z

The monitoring and macro commands are allowed as well as programmed command qualifiers. As before, %C is used to close the session.

The %0=out and %0 commands are allowed, text being selected for the output file by the commands N and A applied to the file being examined, not to a secondary input file. %S is not allowed in SHOW.

THE COMMAND RECAP

        RECAP

This command is used to interrogate the file containing a copy of all interactive terminal Input/Output operations for this user; one of the parameters TEMPRECALL or PERMRECALL of command OPTION (Chapter 17) must have been used to specify that the terminal monitoring facility is required. RECAP is used without a parameter, and its operation is identical to that of SHOW.

SUMMARY OF ECCE COMMANDS

Special commands

%C	Close the editing session.
%F	Full monitoring.
%L	Lower case terminal.
%M	Monitor normally (default).
%O	Secondary Output mode
%Q	Quiet mode: no monitoring.
%S	Secondary input mode.
%U	Upper case terminal (default).
%X	Macro definition.
%Y	Macro definition.
%Z	Macro definition.

Programmed command qualifiers

( )	Bracket a group of commands for purposes of repetition and failure.
?	Optional execution of command.
\	Invert the failure condition (succeed if and only if the command fails).
,	Alternative execution (IF-THEN-ELSE...).

Simple commands

A	Abstract all text between the last Noted position and the file pointer.
B	Break. Insert a newline at the current position.
C	Cause Case inversion of the next character if a letter.
C-	Cause Case inversion of the previous character if a letter.
D/TEXT/	Delete the first occurrence of TEXT on the current line.
E	Erase the next character.
E-	Erase the previous character.
F/TEXT/	Find the first occurrence of TEXT.
G	Get a line of input.
I/TEXT/	Insert TEXT at the current position.
J	Join the current line to the next line.
K	Kill the current line.
L	Left shift the file pointer by one character position.
M	Move the file pointer to the next line.
M-	Move the file pointer to the previous line.

Simple commands (continued)

N   Note the current position.

P   Print the current line.

R   Right shift the file pointer by one character position.

S/STR/   Substitute STR for the TEXT just Found, Uncovered, or Verified.

T/TEXT/   Traverse TEXT. Move the file pointer past the next occurrence of TEXT on the current line.

U/TEXT/   Uncover TEXT. Remove all characters between the file pointer and the next occurrence of TEXT on the current line.

V/TEXT/   Verify that TEXT occurs immediately to the right of the file pointer.

X   Macro invocation.

Y   Macro invocation.

Z   Macro invocation.

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 5 and 6).

Principle of operation

As explained in Chapter 1, all files (except those held on magnetic tape) are accessed by connecting them at an address in a very large virtual memory. The languages available 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 2900. On the other hand, these routines are necessarily 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 data, and at the same time free the user from any constraints imposed by file formats; for example, the restriction that IMP direct access files must have a block size of 1024 bytes.

Direct mapping can be used most readily with IMP mapping facilities; the user is referred to the IMP manual (Reference 7) for further information. In particular, array pointer variables are used in the following examples.

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 their use requires a knowledge of the detailed structure of the file, which is liable to change.

* Data files with no record structure: these files are intended for direct mapping. They have no pre-defined structure, and can only be used for this type of file access.

Note that a file to be used for direct mapping must already exist - it will not be created by the Subsystem merely as a result of being DEFINEd and then referenced by a program (see below).

Creating a store map file - the command NEWSMFILE

          NEWSMFILE(newownfile, size)

The command NEWSMFILE is used to create a store map file (i.e. an unstructured data file). It takes two parameters, both obligatory:

newownfile   the name of the file to be created

size   the length of user data required (in bytes)

For example, if it were required to create a file to hold an array of 1000 variables, each of 8 bytes, one could give the command

          NEWSMFILE(STORE,8000)

Linking the file to a data structure within an IMP program

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

        DEFINE(27,STORE)

Note that .NULL must not be specified in the DEFINE command when direct mapping is to be used.

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 of the user's data in the file STORE (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 for finding the length of a file (see later example).

If the variables in the file STORE are of type longreal then they could be accessed using the longrealmap LONGREAL:

        LONGREAL(START) = PI*R**2
        J = LONGREAL(START+8)
        LONGREAL(START+16) = LONGREAL(START+24)
                •
                •

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:

        %INTEGER START, LEN
        %LONGREALARRAYFORMAT OUTAF(1:1000)
        %LONGREALARRAYNAME OUTA
                •
                •
        START = SMADDR(27,LEN)
                •
                •
        OUTA==ARRAY(START,OUTAF)

From this point on, the contents of the file can be accessed as OUTA(1) to OUTA(1000).

Note that SMADDR must not be called more than once in respect of a particular file, unless the file in question has been closed in the meantime (see below). Thus the information obtained from SMADDR - start address and length of user data - must be saved, as above, if required later in the program.

Effect of accessing the array

It should be understood that, in the above example, the array OUTA and the file onto which it has been mapped 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:

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

This change lasts not merely until the end of the program - it is permanently recorded in

the file, and any subsequent access to this file will find these elements cleared to zero. Thus, where a file is 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 or complex directories or tables.

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 program variables that have 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 area of virtual memory used for the mapped file might have been 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)

Examples of store mapping

In the first example a record array name is mapped onto an unstructured data file. Some of the record elements are accessed.

```
%ROUTINE PAYCHECK(%INTEGER CHANNEL, RECNO)
%INTEGER I, J, K, START, LENGTH
%STRING(11) NAME
%EXTERNALINTEGERFNSPEC SMADDR(%INTEGER AD, %INTEGERNAME L)
%EXTERNALROUTINESPEC CLOSESM(%INTEGER C)
%RECORD %FORMAT PAYF (%STRING(11) SURNAME, %INTEGER AGE, SEX, YEAR, %C
                                          %INTEGERARRAY SALARY (1:12))
! Each record thus formatted contains 72 bytes.
%RECORDARRAY %FORMAT PAYAF(1:RECNO) (PAYF)
%RECORDARRAY %NAME PAY (PAYF)
        .
        .
! Assume that a file was associated with channel CHANNEL via DEFINE,
! prior to the execution of the routine.

    START=SMADDR(CHANNEL,LENGTH); ! File now connected.
    %IF LENGTH < 72*RECNO %START
        PRINTSTRING("File too small:")
        WRITE(LENGTH,1); NEWLINE
        PRINTSTRING("Must be at least")
        WRITE(72*RECNO,1); NEWLINE
        %RETURN
    %FINISH
    PAY==ARRAY(START,PAYAF)
    ! Now %RECORDARRAY %NAME PAY has been mapped onto the file.
    ! Note that START was set by the SMADDR call.
        .
        .
    NAME=PAY(I)_SURNAME
    %IF PAY(I)_SALARY(J)>350 ....
        .
        .
    PAY(K)_YEAR=1978
        .
        .
    CLOSESM(CHANNEL)
%END; ! Of %ROUTINE PAYCHECK.
```

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 7. 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 set to number of bytes in the file.
                                 ! Channel CHAN must be DEFINEd before the
                                 ! program is executed.
    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 next example the program prints out the numerical values of the bytes in a given record in a file which has fixed length records, and which has been DEFINEd as channel 80:

```
         %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 length:")
             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
```

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

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 <u>before</u> a
  call of SMADDR, or <u>after</u> 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 must
  be 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
  information beyond the end of its new size is lost. Otherwise the previous contents of
  the file remain unchanged. If NEWSIZE is larger than the present size, each additional
  byte is set to zero.

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

Example:

In the following example the file STORE used in an earlier example (of NEWSMFILE) is
extended to allow the program to access 1500 longreal variables; it is assumed that the
command DEFINE(72,STORE) has already been given. Notice that the program does not itself
make explicit reference to STORE, only to the associated channel number.

```
%BEGIN
%EXTERNALROUTINESPEC CHANGESM(%INTEGER CHAN, NEWSIZE)
%EXTERNALINTEGERFNSPEC SMADDR(%INTEGER CHAN, %INTEGERNAME LEN)
%LONGREALARRAYFORMAT OUTAF(1:1500)
%LONGREALARRAYNAME OUTA
%INTEGER LEN, START
! File STORE is defined on channel 72.

    CHANGESM(72,12000); ! Extend to hold 1500 longreals (12000 bytes).
    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 2900. 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
implementations of IMP on ICL 2900 series computers using VME/B or VME/K make no provision
for direct addressing of files. On the other hand, for programs written specifically for
use on EMAS 2900 the use of direct mapping merits serious consideration.

Conclusion

One of the problems involved in using mapped files lies in overcoming the conceptual block
to the idea that a program can access a file without using read and write routines. This
difficulty may exist because programmers have grown accustomed to thinking in terms of
reading and writing data between the computer system and the "outside world", even when
that outside world is part of the computer system, such as a disc file. Store mapping
exploits the fact that, in EMAS 2900, a disc file can be equated with storage space used
by a running program for holding variables.

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: mapping 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 structured in a way that is convenient to the programmer.

The primary use of magnetic tapes in EMAS 2900 is by the archive and back-up components of the file system (described in Chapter 1). These functions are controlled by the System and the user need have no knowledge of tape formats, serial numbers and so on.

There are two other categories of magnetic tape use:

* Standard user magnetic tapes. These are managed by the user via high-level language programs; they must be in a standard format. This mode of access is described in the rest of this note.

* Non-standard tapes. These include tapes which are compatible with the hardware - see below - but which have one of a wide variety of non-standard formats, depending on the System on which they were written. A limited range of utility programs is available to access them. In the first instance intending users should contact their local Advisory Service.

Magnetic tape hardware

The ICL 2900 series computers which can run the EMAS 2900 operating system normally have tape decks which can only read and write magnetic tapes with the following characteristics:

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

(Some installations may also be able to handle 800 bpi, NRZI mode tapes.) Users having magnetic tapes destined for EMAS 2900 with characteristics which differ from these should contact their local Advisory Service in the first instance, where they can obtain information about converting them to a suitable form for input.

Tape labelling standard

User magnetic tapes read and written on EMAS 2900 must have IBM OS/370 labels. This standard was adopted because it is well documented (see Reference 11 for tape label formats, and Reference 12 for tape file formats), and it is likely that users transferring magnetic tapes to other installations will find it the most convenient standard.

F	Fixed	One fixed-length record per block.
FB	Fixed Blocked	An integral number of complete fixed-length records per block.
V	Variable	Variable-length records, one per block.
VB	Variable Blocked	Variable-length records, one or more per block.
VS	Variable Spanned	Variable-length records, possibly spanned over more than one block.
VBS	Variable Blocked Spanned	Variable length records, one or more per block, with each record possibly spanned over more than one block.
U	Undefined	Undefined format - one variable-length record per block.

Table 10.1: Summary of Magnetic Tape Record Formats

Full details of the available record formats are contained in the references given above. They are summarised in Table 10.1.

Notes

* The letter A following any of the formats given in Table 10.1 implies that the first character of each record is to be used as a format effector. This is only likely to be of use for files being sent to IBM installations.

* A record is the unit of information with which the user's program operates.

* A block is the unit of information with which the System operates. A file is held on magnetic tape as a series of blocks, each usually (but not always) consisting of several records. When information is stored on a magnetic tape with only one record per block, the tape file is said to be unblocked; otherwise it is blocked.

* When a record of a magnetic tape file starts in one block and finishes in a subsequent block of the file, the record is said to be spanned across the relevant blocks.

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

　　　　DEFINEMT(channel, file, vol, label, record format and length, blocksize)

This command takes up to six parameters, the first three of which must be specified.

channel	An integer with value in the range 1-80. This specifies the channel number used by the program to gain access to the file.
	Note that only one file per tape can be selected by the user's program for reading or writing at one time, although more than one can be defined.
file	The name of the tape file to be read from or written to.
	When naming a tape file the user can specify up to 17 characters, in any format. For IBM compatibility, the form required is:
	name1.name2.name3 ...
	where name1 must be given and each of name1, name2, etc. consists of not more than 8 upper case letters or digits, starting with a letter. As few or as many names as required, separated by dots, may be specified, subject to the overall maximum of 17 characters. In particular, the name does not have to be in the standard EMAS 2900 format. This feature may be needed when reading tapes on EMAS 2900 which were written at other installations.
vol	The 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	A positive integer which specifies the position of the file on the tape (default = 1).
	Note that writing to a tape file with label n, say, automatically destroys all files on the tape with labels greater than n. Thus, if one writes a file with label 2 on a tape which previously contained 5 files, at labels 1 to 5, the effect is that the label 1 file is unchanged, the label 2 file is overwritten, and the files at labels 3, 4 and 5 are all lost.
record format and length	Of the form "rn", where

> r   is the record format: one of FB, V, VBS, etc., as listed above
>
> n   when the record format is "variable", the <u>maximum</u> number of bytes per record available for user data; when the record format is "fixed", the <u>actual</u> number of bytes available

This parameter is similar, but not identical, to the IBM parameter "LRECL": when the record format is "variable", the "n" part of the IBM parameter specifies the maximum number of bytes available for user data <u>plus</u> 4 bytes for red tape at the start of the record.

This parameter is ignored when reading; in this case the information is extracted from the tape label (default when writing = VB1024).

blocksize	A positive integer in the range 18-32760. It is used when writing files with fixed-length records to specify the length of each block, and when writing files with variable length records to specify the maximum length of each block.
	Note that if a fixed record length is used, the blocksize must be an exact multiple of the record size; if a variable record length is used, the blocksize must be at least 8 bytes greater than the record size. As with the previous parameter, this parameter is ignored when reading. (Default when writing = 4096, or a value close to 4096 compatible with the "record format and length" parameter)

Examples of valid calls of DEFINEMT

     DEFINEMT(7,ERCC27.TESTTAPE,AS1273)

     DEFINEMT(18,EJNN33.DATA2704,EA1777*,7,FB100,8000)

     DEFINEMT(5,ERNN.JXTB15.17SST,X13621)

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 2900 internal code (see Appendix 2), from or to EBCDIC on magnetic tape. Thus a tape written using IBM 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

It may be necessary to restrict access to user magnetic tapes. This is because the number of tape decks available for users at an EMAS 2900 installation is likely to be much lower than the number of simultaneous users. Further information should be sought from your local Advisory Service.

CHAPTER 11
COMPILERS, OBJECT FILES AND PROGRAM LOADING

The Edinburgh Subsystem includes compilers for the programming languages IMP, FORTRAN and
ALGOL 60. The relevant language manuals (References 7, 4 & 8 and 5 & 9 respectively)
contain full details of the languages. The environments provided by the Subsystem for
programs written in these languages are described in Chapters 12, 13 and 14 respectively.

This chapter first describes the commands that invoke the compilers, and the associated
commands PARM, LINK and RUN. There then follows a description of the loading of programs
prior to their execution.

The commands IMP, FORTE and ALGOL

      IMP(<u>source</u>, <u>object/.NULL</u>, clist/.NULL, out)

      FORTE(<u>source</u>, <u>object/.NULL</u>, clist/.NULL, out)

      ALGOL(<u>source</u>, <u>object/.NULL</u>, clist/.NULL, out)

These commands are used to compile IMP, FORTRAN and ALGOL 60 programs, respectively. In
each case the four 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 DECLARS+B+C.

2. The name of the object file to be generated by the compiler. Object files are
    described in detail below, under "Program loading". If a file of the given 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 compilation time.
    Note that this parameter cannot specify a member of a partitioned file. An object
    file can, however, be copied into a partitioned file and used from there (see
    below).
    The filetype is "object" if the compilation succeeds, and "corrupt object" if it
    does not.

3. The name of a listing file or device. If this parameter is omitted a file created
    by the Subsystem with the name T#LIST will be used. Valid alternatives include
    output devices (see Chapter 2). Again, .NULL can be used if no listing is
    required. Note that T#LIST is destroyed when the user logs off, and is overwritten
    if used in another compilation in the current session.

4. A supplementary output file or device, for error messages only. This is normally
    used to provide a list of error messages on the interactive terminal (device code
    .OUT). If omitted no separate list of faults is produced unless a default "out"
    has been defined by use of the OPTION parameter CFAULTS (see Chapter 17).

Examples of calling compilers

      FORTE(FORTP4,P4Y,.LP)

This would compile the FORTRAN source file FORTP4 into an object file P4Y and would output
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 T#LIST. Note that if T#LIST contains a listing from a previous compilation
in the current session it will be overwritten.

      IMP(ERCC77.BASE+MINE,ABCOBJ,.LP14,.OUT)

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

PARM(parmlist/?)

Various compile time and loader options can be selected; this is done by means of a call of the command PARM. A call of PARM takes effect for all future compilations until the user calls PARM again or logs off, whichever is sooner. PARM has the effect of resetting all values to the default settings, and then setting the ones selected (cf. OPTION). Hence PARM with no parameters merely resets the defaults. See also OPTION(INITPARMS), described in Chapter 17.

Table 11.1 gives details of how the IMP and ALGOL compilers use the PARMs, and which are selected by default in most installations. (Exceptions to the choice of default PARM options are given in Appendix 5.) Table 11.2 gives the same information with respect to the FORTRAN compiler.

The form PARM(?) can also be used; it causes the currently selected options differing from the default selection to be printed.

Notes

* The options must be specified before compilation, as they affect the compilation process.

* The existing options are liable to change, and new ones may be introduced. Such changes will be advertised in the usual way; see Chapter 4, "Subsystem information", for details.

PARM option pairs (Default underlined)	Notes
NOLINE LINE	LINE causes code to be included to keep a record of the number of the line being executed, for diagnostics.
DYNAMIC STATIC	DYNAMIC marks all references for satisfying dynamically. In addition, in IMP, %EXTERNALROUTINEs are loaded dynamically if they are specified in the calling program as, e.g., %DYNAMICROUTINESPEC. See also "Dynamic loading", below.
FREE FIXED	FREE causes no source line length limit to be applied. Applicable to both IMP and ALGOL.
DEBUG NODEBUG	Related to the IMP "debug" facility. See Reference 13, available from your local Advisory Service.
MAP NOMAP	Not used by compilers. MAP causes command LINK (see below) to output a map of linked file.
OPT NOOPT	OPT - optimise code - gives rather more than NOCHECK, NOLINE, NODIAG (e.g. stack overflow not checked).
CODE NOCODE	CODE causes a listing of the object code generated by the compiler to be produced.
LET NOLET	At run-time, LET allows loaded program with unresolved external references to start execution. ("High-risk" option.)
PROFILE NOPROFILE	IMP: PROFILE causes extra code to be inserted to keep a count of how often each line is executed. Call of routine PPROFILE then gives summary of information and resets counts. PARM LINE is forced on. ALGOL: n/a.

Table 11.1: IMP and ALGOL PARM Options
(continued on next page)

PARM option pairs (Default underlined)		Notes
NOTRACE	TRACE	TRACE causes routine traceback information to be kept, for diagnostics.
NOARRAY	ARRAY	ARRAY causes an array bound check to be carried out on all array accesses. Note that some checking on multidimensional array bounds will be carried out even if NOARRAY is selected.
NOCHECK	. CHECK	CHECK causes a check to be made that each variable whose value is about to be used has been assigned.
STACK	NOSTACK	IMP: STACK causes arrays to be put on the user stack, rather than on an auxiliary stack; liable to fail with "stack overflow". For special cases only. (See "The use of the stack", below.) ALGOL: n/a.
NODIAG	DIAG	DIAG causes symbol tables to be retained at run-time, for the production of values in local variables in diagnostics.
NOLIST	LIST	LIST causes a program listing to be generated at compile-time. NOLIST causes output of a heading and any error messages only.
QUOTES	PERCENT	IMP: n/a. ALGOL: Specifies keyword notation.
MAXDICT	NORMALDICT	MAXDICT causes the compiler tables to be set to the maximum possible size.

Table 11.1: IMP and ALGOL PARM Options

PARM option pairs (Default underlined)		Notes
MISMATCH	NOMISMATCH	MISMATCH: the size of a subprogram argument is determined by the size of the corresponding dummy argument. NOMISMATCH: the sizes of the corresponding dummy and actual arguments must be the same. NOMISMATCH results in faster execution.
EBCDIC	ISO	Specifies code used for internal character data and Hollerith constants.
DYNAMIC	STATIC	DYNAMIC causes all references to be marked for satisfying dynamically. See also "Dynamic loading", below.
DEBUG	NODEBUG	n/a at present (but is planned).
MAP	NOMAP	Not used by compiler. MAP causes command LINK to output a map of linked file.
OPT	NOOPT	OPT is equivalent to NOCHECK and NODIAG.
ATTR	NOATTR	At end of each program unit, ATTR causes list of attributes (type, size, whether scalar or array, etc.) to be output.

Table 11.2: FORTRAN PARM Options
(continued on next page)

PARM option pairs (Default underlined)		Notes
CODE	NOCODE	CODE causes a listing of the object code generated by the compiler to be produced.
LET	NOLET	(Run time option.) LET allows loaded program with unresolved external references to start execution. Also, suppresses warning messages if there is an inconsistency between lengths of references to the same data entry.
LABELS	NOLABELS	LABELS causes code to be included to maintain a list of the most recent labels, routine calls and returns executed, for diagnostic purposes. Currently 32 entries are kept (multiples, e.g. in DO loops, count as 1). Subroutine LABELS can also be called to get information output (see Reference 6 for details).
XREF	NOXREF	XREF causes output of references to variables, with line numbers of their occurrences.
ZERO	NOZERO	ZERO causes COMMON areas not initialised by a BLOCKDATA to be filled with zeros; otherwise they are filled with the unassigned pattern.
INHIBIOF	ALLOWIOF	INHIBIOF inhibits the occurrence of an integer overflow interrupt. If not set, the least significant 32 bits are used when an integer overflow occurs. INHIBIOF may be withdrawn later.
PROFILE	NOPROFILE	n/a (to be introduced later).
NOARRAY	ARRAY	ARRAY causes a hardware check that all array subscripts lie within bounds at run-time. With NOCHECK (see below) selected, ARRAY checks upper bounds but not all lower bounds. NOARRAY causes hardware array bound checking to be omitted.
NOCHECK	CHECK	CHECK causes a check to be made that all variables whose values are used have been assigned. It also causes a check that the number of parameters passed to a subprogram is correct, and that array element references are in the total bounds of the array. NOCHECK omits all this checking, but also causes all variables established by the compiler (i.e. excluding COMMON blocks other than those appearing in BLOCK DATA) to be initialised to zero.
NODIAG	DIAG	DIAG causes symbol tables to be retained at run-time, for diagnostics.
NOLIST	LIST	LIST causes a program listing to be generated at run-time. NOLIST causes only erroneous statements to be listed.
MINSTACK	NORMALSTACK	MINSTACK causes array dope vectors to be stored off the stack, thus reducing the stack size.
MAXDICT	NORMALDICT	MAXDICT causes the compiler tables to be set to the maximum possible size.
R8	R4	No. of bytes allocated to REAL variables.
L8	L4	No. of bytes allocated to LOGICAL variables.
I8	I4	No. of bytes allocated to INTEGER variables.

Table 11.2: FORTRAN PARM Options

The command LINK

        LINK

This command can be used to link (up to 256) object files to produce a compound object file. It can be used to reduce the number of separate files in the user's file index. 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 6). It prompts for input files using the prompt "Link:". The list must 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 input filenames and the output filename must all be different.

The following example shows how the command can be used:

        Command:LINK
        Link:P4Y
        Link:ABCOBJ
        Link:PDFILE_MEM1
        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.

Note that it is not possible to extract a constituent object file from a linked file.

The command RUN

        RUN(object)

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. An object file member of a partitioned file can also be specified.

Examples:

        RUN(MYPROG)

        RUN(EGNP99.SUMXOB_GO22)

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 7). A call of CPULIMIT may also be required to set an appropriate time limit (see Chapter 17).

When a program belonging to another user is to be run, the object file must be permitted to this user in READ mode.

PROGRAM LOADING

An important part of the Subsystem is the "program loader". This is used to modify the contents of the virtual memory in preparation for the execution of a program or a command. The program loader is called automatically by the following:

 * The Subsystem command interpreter (Chapters 15 and 16), when the user types a command, or a command is specified in the job stream, which is not already loaded.

 * The Subsystem job interpreter (Chapter 16), when a command not already loaded is specified in the job stream.

 * The command RUN, described above.

 * The IMP routine CALL (Chapter 15).

 * The FORTRAN routine EMASFC (Chapter 13).

The design of the program loader is such that most users will not find it necessary to understand the details of its operation. Much of this section is primarily directed to those who, for interest or because of the size or complexity of their programs, wish to know more about the process of program loading.

Object files

Object files are generated by the various language compilers in a standard format. One result of this is that it is possible, subject to the limitations of the parameter passing facilities of each language, for a procedure written in one language to call a procedure written in another. Thus an IMP program could call a FORTRAN subroutine.

An object file can contain some or all of the areas listed below. The precise ways in which the compilers utilise these areas and the names they give to them vary, but one important distinction is made in every case: that between shared and unshared information.

* code	This is the main part of the file and contains the instructions which are to be executed by the computer. This area is shareable.
* constants	These are values which are used by the program but are not altered during its execution; e.g. IMP %CONSTINTEGERs. This area is shareable.
* initialisation values	These are values which are used to initialise variables which may be modified during program execution; the values specified in, for example, DATA statements in FORTRAN programs and %OWN variable declarations in IMP programs. This area is used only at load time, when the values are copied into an unshared area (not in the object file).
* diagnostic tables	These are used to provide run-time diagnostic information in the event of a failure or an explicit call on the diagnostic package. This area is shareable.
* load data	This contains the names of entry points (procedure names) in this object file and the references made by this file to other object files. This information can be listed using the command ANALYSE (Chapter 6). The load data also contains information which is used by the program loader. It is used only at load time, when the information in it is copied into an unshared area (not in the object file).

Sharing object files

In order to avoid the cost of copying, and to take advantage of the file sharing facilities (Chapter 1), the format of an object file is such that most of the information in it can be accessed directly from the file at run-time.

However, while a program is being executed it is necessary for the user to have space in his virtual memory for variables and linkage information. Clearly the space for variables must be unique to each user, and so it is not possible to locate it in the object file itself. Further, linkage information has to be unique because it contains pointers to other object files, which may have different virtual addresses in different users' virtual memories. As explained above, the object file contains the initial values to be assigned to initialised variables, e.g. %OWN variables in IMP or variables initialised by DATA statements in FORTRAN. Part of the process of loading an object file involves copying the initial values from the object file into the particular user's work area.

Because unique copies of the unshareable parts of an object file are made at load time, it follows that if several users are executing the same program simultaneously (often true of utilities such as the Subsystem editor), then they can all use the same copy of the object file. As a result there is a reduction in paging and in use of the main store. It is important to stress that the production of shareable object files is the compiler writer's responsibility and does not impose any restrictions on the user.

Loading a single object file

The process of loading a single object file involves the following actions:

* Connecting the file in the virtual memory (see Appendix 1). The address at which the file is connected is not important - it can be at different virtual addresses in different virtual memories.

* Copying the initialisation values from the object file to the user's work area.

* Establishing links, as necessary, to other procedures that are already loaded.

* Adding names to the list of procedures still to be loaded.

* Relocating addresses in the program's work area, to reflect the addresses at which the object file and the work areas are connected.

Dynamic loading

Normally all the external references made by a program must be resolved (by the loading of appropriate object files) before the program starts execution. This can mean that large programs or packages take a relatively long time to start execution, because of the complexity of the loading process. With such programs a number of entry points might not be used during a specific run, so that the time taken to locate and load the relevant object files has effectively been wasted.

It is possible, however, to arrange that a routine (i.e. the object file containing it) be loaded only when it is required, and not automatically prior to the start of execution of the program. This is known as dynamic loading, since it takes place when the program is executing.

It is important to appreciate that for a routine to be loaded dynamically it is not necessary to modify the routine in any way. It is the program or procedure that references the routine that must be marked to inform the loader (a component of the Subsystem) that the reference should be satisfied dynamically. This must be done at compile time, thus:

ALGOL and FORTRAN: compile with PARM(DYNAMIC) - all references are then marked for satisfying dynamically.

IMP: use %DYNAMICROUTINESPEC in place of %EXTERNALROUTINESPEC (see Chapter 12) for all routines and procedures which are to be loaded dynamically. It is also possible to use PARM(DYNAMIC), causing all references to be marked for satisfying dynamically.

To obtain information about the references in an object file, use

ANALYSE(objectfile, R)

Running a program with dynamic references

For IMP and ALGOL programs there is no further special action required. The program is run or command called just as for a program without dynamic references. When the file containing the dynamic references is loaded, any references to procedures already loaded are satisfied immediately. All others are left. The program then starts executing until a call is made on a procedure which is not loaded. At this point a jump is made to the loader and an attempt made to load the required procedure. If this fails then a message of the form

Failed to load NAME dynamically. NAME not found.

where NAME is the name of the routine in question, is output, followed by diagnostics, and program termination. If it is successful then the reference is filled in the program's GLA (general linkage area) and the routine is called. Subsequent calls behave exactly as for calls to routines loaded prior to execution.

Where the routines being loaded dynamically include FORTRAN code there is a complication. FORTRAN uses part of the user stack (see the end of this chapter) for its scalar variables, but unlike IMP and ALGOL this space is allocated all the time the file is loaded - not just during the execution of the routine. This so called "initialised stack"

must be put at the bottom of the user stack, and for this to be possible a space must be allocated. With dynamic loading the loader does not know precisely how much space to allocate because, until execution begins and the additional routines are loaded, it has no information about how much space is required. It therefore leaves a space which is large enough to be adequate in most cases. The size of the space is given by the INITSTACKSIZE parameter of command OPTION (see Chapter 17 and below).

However, for large FORTRAN programs or packages it may be necessary to increase this parameter from its current value, which can be determined from the command OPTION(?). Currently its maximum possible value is 100 (Kbyte).

Locating a procedure

A required procedure is located by one of two methods. When the command RUN is used the parameter is the name of an object file or member which contains a compiled program. Thus the object file is explicitly named. In the case of a Subsystem or user-written command, however, or a reference from another object file, the loader is only told the name of the required procedure, not the name of the object file containing it. The directory structure is used to locate the required object file, given a procedure name.

Directory files

A directory file contains two types of information:

* a mapping between procedure entry names and the names of object files (and members) which contain the procedures

* a mapping between alias names and the entry names to which they refer

Creating directory files

Directory files are normally created automatically when certain parameters of the command OPTION are used (see Chapter 17), but they can be created explicitly by use of the command NEWDIRECTORY (see below). At any time a user has a current active directory, which must be his own file; by default this has the name SS#DIR. The name of the active directory can be changed by use of the command OPTION (see Chapter 17). The current active directory is the directory which is modified by explicit or automatic calls of the commands INSERT, REMOVE and ALIAS (see below). It is also the first directory to be searched for the names of required procedures which are not already loaded. If a call is made on one of the commands INSERT, REMOVE or ALIAS and the active directory does not exist then one is created automatically, with default size.

The command NEWDIRECTORY

         NEWDIRECTORY(newownfile, num1, num2)

This command can be used to create a directory file. Note however that it is rarely necessary to use it - only when a directory larger than the default size is required. When the default size is satisfactory, a directory file may be created by use of the command OPTION (Chapter 17).

The first parameter of NEWDIRECTORY is the name to be given to the directory file. A file of that name must not already exist.

The second and third parameters are optional and can be used to specify the sizes of the two tables in the file. The second parameter specifies the number of procedure entry names and aliases that can be stored in the directory. When this number is calculated an allowance should be made for extra space in the table: because of the method used to locate a name, the efficiency of the search decreases significantly when the table is more than about 80% full. Thus, for an application involving 200 entry names, the second parameter should be about 250.

The third parameter is used to set the length of the "PLIST". This is an area in the file used for holding the following information:

* Names of object files containing the entries in the directory.

* Aliased names, i.e. the names to which the aliases have been added.

* Extensions to entry names and aliases. This only refers to names which are longer than 10 characters.

It is difficult to calculate the precise length required for the PLIST; a value of about five times the number of entries is of the right order. It may have to be increased if the directory is to contain many long entry names, or refers to a large number of object files which each contain few entry names.

The default values for the second and third parameters are 160 and 856.

The command ANALYSE can be used to obtain information about the size of a directory file and the extent to which its two tables are filled.

The command TIDYDIR

        TIDYDIR(directory, percentage)

TIDYDIR is used to tidy a directory file. It takes two parameters, both optional:

directory   The directory file to be tidied; if no parameter is given, the current ACTIVEDIR (see OPTION, Chapter 17) is used.

percentage   A number, between 10 and 100, which indicates how full (as a percentage) the directory is to be after tidying. The default value is 70(%).

TIDYDIR performs the following operations on the directory file:

* It removes any INSERTed files which are no longer available.

* It re-uses wasted space in the PLIST area of the directory.

* It increases or decreases the size of the directory, depending on how full it is.

* It adjusts the relative sizes of the procedure entry table and PLIST area within the directory, in proportion to their use.

Note that for directories with a calculated new size of less than an epage, only the REMOVE and "PLIST re-use" operations are performed, and the default sizes for the procedure entry table and PLIST area are used. See the description of NEWDIRECTORY (above) for further information.

The structure of a directory file can be found by using the command ANALYSE in the following form:

        Command:ANALYSE(directory,S)

The command INSERT

        INSERT(objectfilelist)

For the directory search system to operate it is necessary for the relevant directory or directories to contain information about the object files which a user wishes to access. The command INSERT is used to put information about one or more object files into the current active directory.

Apart from the explicit call of INSERT the use of any of the standard compilers has the effect of re-INSERTing an object file after successful recompilation. Thus if a user inserts the object file OBJECT23 into his current active directory once, and subsequently recompiles it several times, the information relating to OBJECT23 will be kept up to date in the active directory. For example, it will include any change to the names of procedures in the file. If a subsequent compilation fails, the name OBJECT23 (with no entries associated with it) will be left in the directory, so that it gets properly re-inserted when the user eventually does compile it successfully.

If INSERT is used in respect of an object file which is currently inserted in the active directory, the effect is to REMOVE the existing information and then to re-INSERT it, thus ensuring that the information reflects the current state of the object file.

The command REMOVE

> REMOVE(objectfilelist)

The command REMOVE is used to remove information about one or more object files from the current active directory. Apart from explicit calls, REMOVE is invoked automatically when necessary by INSERT (see above), and when an object file is destroyed.

The command ALIAS

> ALIAS(command, newname)

ALIAS is another command used to put information into the current active directory. The command enables a user to specify an alias for a particular procedure (command) name. When an alias is being added to a name the form is

> ALIAS(name, alias)

For example, if a user wishes to refer to the command ALGOL with the alias A, he should type:

> ALIAS(ALGOL,A)

To remove all the aliases to a given name, the second parameter should be omitted; e.g.

> ALIAS(ALGOL)

Notes

* The aliases which have been specified are checked by the program loader, both when loading a command and when satisfying references from other procedures. Thus in the example above the user could reference the ALGOL command from within a program using the name A.

* A directory file containing alias information need not contain a reference to the actual procedure aliased. If it does not, the search mechanism described below is used to locate it.

* If an attempt to invoke a procedure by means of an alias fails (because the procedure could not be found), the resulting error message specifies the name of the procedure rather than the alias used in the attempt to invoke it.

Searching for a procedure

The following directories are searched to locate a procedure:

* The session directory: this is a directory containing information about all the procedures which are currently loaded. At the start of the session this comprises the procedures in the Subsystem "basefile" only; these include most of the commonly used commands. Later during the session, as other Subsystem components are used (such as the compilers or the mathematical routines), they are added to the session directory.

* Next, the user's active directory is searched (see below).

* Next, any directories which have been specified by the SEARCHDIR parameter to the OPTION command (see Chapter 17). Note that OPTION(?) can be used to obtain a list of these directories.

* Finally, a directory of Subsystem procedures is searched. This contains references to object files containing all the Subsystem procedures not in the basefile.

There are a number of implications of this search order:

* Since the session directory always contains references to the procedures in the Subsystem basefile, these cannot be replaced by references to procedures of the same name belonging to a user.

* Subsystem procedures other than those in the basefile can be overridden only if they have not already been loaded during this session.

* The number of directories to be searched affects the speed of the loader. It is advisable to check the list periodically, using OPTION(?), and to remove any redundant entries. In addition, the list should be ordered so that the most frequently used procedure names are in the active directory or the SEARCHDIR directory nearest the top of the list.

* When a command that is already loaded is accessed via an alias, the alias will be noted in the session directory and will thus remain effective for the rest of the session, even if it is later removed from the active directory. Thus in the following sequence the command M will access the METER command for the duration of the session, even though it has been removed from the active directory.

Command:ALIAS(METER,M)

Command:M

27/10/78 14.04.33 CPU= 1.52 Secs CT= 0 Mins PT= 152 CH= 67p

.
.
.

Command:ALIAS(METER)

Command:M

27/10/78 14.05.20 CPU= 2.03 Secs CT= 1 Mins PT= 282 CH= 93p

Example of setting up a directory file

Suppose that it is required to add, to a directory file, references to a number of compiled programs, and make it available to other users. In addition, the user doing this wishes to add the directory to his own list of directories to be searched (the "SEARCHDIR" list). The command OPTION, which is used, is described in Chapter 17.

```
OPTION(ACTIVEDIR=GENERLIB)
INSERT(OBJ1,OBJ2,GENOBJ_A1,GENOBJ_A2,ABCD01.SUNDRY)
REMOVE(GENOBJ_A1)
ALIAS(SMITH,JONES)
OPTION(SEARCHDIR=GENERLIB,ACTIVEDIR=SS#DIR)
PERMIT(GENERLIB)
PERMIT(OBJ1)
PERMIT(OBJ2)
PERMIT(GENOBJ)
```


Notes

* The first OPTION command causes GENERLIB to be the user's active directory; in addition, GENERLIB is created (default size) if it does not already exist.

* The INSERT command causes references to the object files specified to be added to the active directory (i.e. to GENERLIB). Note that object file members of partitioned files can be inserted, as can object files belonging to other users.

* The REMOVE command causes the reference to GENOBJ_A1, just inserted in the active directory, to be removed (presumably because the user realised that it was not to be included).

* The ALIAS command causes JONES to be an alias of SMITH. Thus when JONES is specified, the procedure SMITH will be sought. The alias information is added to the current active directory (in this case GENERLIB), and any user including the directory in his SEARCHDIR list will be able to use JONES as an alias for SMITH.

* The second OPTION command causes GENERLIB to be put at the head of this user's SEARCHDIR list; in addition SS#DIR is made his active directory henceforth. Note that it is permissible for the current active directory to be in the SEARCHDIR list, although when encountered there it is ignored, having already been searched as the active directory.

* The various PERMIT commands give all users read access permission to GENERLIB and to the object files inserted into it in this example, apart from ABCD01.SUNDRY, for which this user cannot set access permissions (since it is not his own file). Giving access to a directory file does not automatically give access to object files referenced by it.

The use of the stack

The stack is an area of data storage which operates on a "last-in first-out" basis. One of the hardware restrictions imposed by 2900 series machines is a limitation on the size of the hardware stack. Despite supporting a very large virtual memory the hardware only allows for stacks of up to 255 Kbyte. For many users this should present no problems but for users of large or complex programs or packages this prove awkward. The compilers attempt to minimise the effects of this restriction in different ways. The IMP and ALGOL compilers normally store arrays in an auxiliary stack which is a standard file and to which the 255 Kbyte limitation is not applied. The hardware stack is then used only for local scalar variables and the control information needed for each routine call. Only programs that use very large numbers of local scalar variables or much recursion will be affected by the stack size limit.

In the case of the current FORTRAN compiler the solution is somewhat different, and the USERGLA area (see below) is used for holding arrays. Furthermore, in order to exploit the shorter, faster instructions available for accessing variables on the hardware stack, the compiler uses the stack very heavily. One effect of this is that very large FORTRAN programs may not load. In this case the OPTION parameter INITSTACKSIZE (see below) may need to be increased. Contact your local Advisory Service if loading a FORTRAN program or package proves troublesome.

Changing the sizes of work areas

The command OPTION (see Chapter 17) can be used to change the sizes of various files related to loading. The form OPTION(?) can be used to determine their current sizes. The relevant parameters to OPTION are:

USERSTACKSIZE=n   This is the stack used when running all programs and procedures other than those in the Subsystem basefile. The maximum value of n is 252 (Kbyte).

AUXSTACKSIZE=n   This is the stack used for IMP and ALGOL local (not %OWN) arrays.

INITSTACKSIZE=n   This is the area at the bottom of the user stack, used by the FORTRAN compiler to store all the scalar variables in a program. The maximum value of n is 100 (Kbyte). As explained in "Dynamic loading" (above), sufficient space must be left for the scalar variables associated with the routines that are loaded either prior to or during execution.

For most users the default sizes of these areas will be adequate; they will only need to be increased for exceptional programs.

Most of the EMAS 2900 operating system and Edinburgh Subsystem is written in the high
level programming language IMP. Consequently EMAS 2900 provides an ideal environment for
running programs written in this language. The language is described in Reference 7.
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 11. 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 normally specified by %EXTERNALROUTINESPEC statements, etc. If a
separately compiled routine or function is to be dynamically loaded, the word %DYNAMIC
replaces the word %EXTERNAL in its specification in the calling program. (Dynamic loading
is described in Chapter 11.)

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 ANALYSE
(Chapter 6), 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

As well as writing complete IMP programs one can write 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 15.

Note the following characteristics of entry points:

* The first 31 characters of the name of the routine or function are used for external
  linking. Names for external entities should therefore differ in their first 31
  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 could be

        %EXTERNALINTEGERFNSPEC WFILE(%INTEGER C, B)

The source file must contain one or more %EXTERNAL routines, each terminated by %END, and
the whole file must be terminated by %ENDOFFILE. Note the following about %EXTERNAL
routines, and about the source files containing them:

* %EXTERNAL routines cannot be nested; on the other hand each routine can contain routines and blocks.

* A file of %EXTERNAL routines can contain routines and functions which are global to the whole file, though not themselves %EXTERNAL.

* A file of %EXTERNAL routines 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;  ! OF %ROUTINE SETUP

%EXTERNALROUTINE FILE(%INTEGER IN, START)
   .
   .
   SETUP(IN)
   .
   .
%END;  ! OF %EXTERNALROUTINE FILE

%EXTERNALINTEGERFN WFILE(%INTEGER OUT, START)
%EXTERNALROUTINESPEC WRITESQ(%INTEGER C, %NAME B,E)

   %ROUTINE CHECK
      .
      .
   %END;  ! OF %ROUTINE CHECK

   SETUP(OUT)
   CHECK
      .
      .
%END;  ! OF %EXTERNALINTEGERFN WFILE

%ENDOFFILE
```

If the object file produced by compiling the above source file were analysed using the command ANALYSE (Chapter 6), it would be found to contain entry points FILE and WFILE. Note that the routine SETUP is not accessible 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 held 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 must 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(4)=244
   .
   .
%END;  ! OF %EXTERNALROUTINE START
%ENDOFFILE
```

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

Notes

* As with routine entries, only the first 31 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 11.

External routines called from programs can be loaded dynamically; the significance of this is explained in Chapter 11. Note that it is the program calling the routine which specifies that the routine is to be dynamically loaded; the routine itself is not changed in any way.

Unsatisfied References

If, during loading, a procedure cannot be located to match an %EXTERNAL specification then an appropriate message is output. Execution of the program is then inhibited unless the PARM option LET has been specified (see Chapter 11). In that case the program is allowed to continue, until a call is made on the procedure. If this occurs an appropriate message is output, followed by an execution of %MONITOR; %STOP.

If an %EXTERNAL data entry is not found to satisfy an %EXTRINSIC reference then a failure message is printed and execution is inhibited.

Debugging IMP programs

The PARM option DEBUG can be used to facilitate the debugging of IMP programs and external routines. A description of the facilities available is given in Reference 13.

LIBRARY ROUTINES

IMP System Library

Standard IMP library routines are available in the System Library, whose directory is searched automatically following a search through any directory files nominated by the user (see Chapter 11 and the SEARCHDIR parameter of OPTION, Chapter 17). The library routines available are described in Reference 6.

Graphics and other libraries

There are libraries of routines for using graphics devices and for other specialist purposes. In order to access such routines it is necessary to use the SEARCHDIR parameter of the OPTION command, as explained in Chapter 17. Details of the routines available can be obtained from Reference 3 or from the Advisory Service.

ACCESSING EMAS 2900 FOREGROUND COMMANDS FROM IMP PROGRAMS

It is possible for an IMP program to access directly any Edinburgh Subsystem foreground command. As explained in Chapter 15, the Subsystem comprises a large number of IMP routines and, in particular, there is one for each foreground command. Any foreground command routine to be called directly must be specified explicitly and must have one string parameter of length 255. For example, if a program is required to call the command DEFINE, it would have to include

        %EXTERNALROUTINESPEC DEFINE(%STRING(255) S)

The string parameter must be specified as of length 255, as shown, even for commands which take no parameters (such as USERS). 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 were required to associate channel 23 with a line printer, then one would type the command

        DEFINE(23,.LP)

If the same command were called from within a program the routine call would be

        DEFINE("23,.LP")

Notes

* Since within a program the command parameter is a string, it must be enclosed in quotes (as above) or be a string expression.

* The Subsystem removes spaces from the parameters of commands typed on the interactive terminal. This is not done in the case of commands called from programs, and therefore the string parameter should not contain spaces.

* Instead of calling foreground commands directly, it is possible to use the routine CALL. This takes two string parameters: the name of the command to be invoked, and the parameter to that command. CALL is described in detail in Chapter 15.

Chapter 15 gives further details of the calling of commands from within programs, and describes the use of the following related IMP routines and functions:

        CALL

        RETURN CODE

        SET RETURN CODE

        SSFMESSAGE

        SSFOFF

        SSFON

OTHER IMP ROUTINES SPECIFIC TO EMAS 2900

There are a number of external routines available to the IMP programmer which are specific to EMAS 2900. 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 command SETMODE is available to set Terminal Control Processor (TCP) characteristics, and to find out what the current settings are. The routine PROMPT and the function INTERRUPT are used to exploit features of the interactive terminal.

SETMODE

The Subsystem command SETMODE (described in Chapter 17) enables the TCP Setmode options (see Chapter 3) to be set, not directly from the interactive terminal, but from the Subsystem. By calling this command from within a program, as described in detail in Chapter 15, it is then possible for a user to have TCP options set from a program running interactively in his process.

PROMPT

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)

When a user's programs are running 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(18) INFILE, OUTFILE
%INTEGER RECORD
     PROMPT("INPUT FILE:")
     READSTRING(INFILE)
     PROMPT("OUTPUT FILE:")
     READSTRING(OUTFILE)
     PROMPT("RECORDS:")
     READ(RECORD)
     .
     .
```

Assuming that the program was compiled into an object file named TRANSFORM, when it is run from an interactive terminal the dialogue might have the following appearance:

```
Command:RUN(TRANSFORM)
INPUT FILE:"RES506"
OUTPUT FILE:"OUTPUT6"
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 specified no prompt is output.

INTERRUPT

The string function INTERRUPT is used to enable a program to respond to interrupts typed at the interactive terminal. As explained in Chapter 3, 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 capital letter followed by CR - these single character interrupts are interpreted by the Subsystem (see Chapter 4).

* A reply of 2-15 characters, terminated by CR - the occurrence of such an interrupt can be detected by the currently executing user program, via INTERRUPT.

INTERRUPT must be explicitly specified:

        %EXTERNALSTRINGFNSPEC INTERRUPT

It returns and clears the last multi-character interrupt from the interactive terminal. Note that there is no queue of such interrupts; thus if a multi-character interrupt has not been read (via INTERRUPT) by the time that another multi-character interrupt is sent then it will be lost. INTERRUPT returns a null string if there is no interrupt outstanding.

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(31) S)

The parameter must be a string or string expression containing the name of the file. The form "file_member" is also permitted. 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(11) FILE
        %EXTERNALINTEGERFNSPEC EXIST(%STRING(31) S)
            READSTRING(USER)
            READSTRING(FILE)
            %IF EXIST(USER.".".FILE)#0 %START
                PRINTSTRING(FILE." exists")
                NEWLINE
            %FINISH

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

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

Channel number / file correspondence

The IMP routine DEFINFO provides an executing program with information on the current links between channel numbers and files. Given a channel number, the routine indicates whether the channel is defined, and the associated filename or device name if it is. DEFINFO must be explicitly specified in the calling program:

        %EXTERNALROUTINESPEC DEFINFO(%INTEGER CHAN, %STRINGNAME FILENAME, %C
                            %INTEGERNAME STATUS)

where:

CHAN	is the channel number about which information is required; the value specified must be in the range 1-80
FILENAME	(set by DEFINFO) is the filename associated with channel CHAN
STATUS	(set by DEFINFO) indicates the status of channel CHAN:

        0   not defined (FILENAME is set to the null string in this case)

    Otherwise, each bit set in STATUS has the following significance:

        $2**0$ defined
        $2**1$ open

Notes

* The bit settings in STATUS may be extended in future to provide additional information; consequently a mask should be used to check STATUS. For example, use STATUS&3=3 rather than STATUS=3 to check if a channel is open.

* FILENAME should always be allowed a length of 255 to cope with the case where a concatenated list is defined on a channel. In this case the string returned is just as it would appear in the corresponding DEFINE command, i.e. with "+" symbols between the concatenated files.

* Where a magnetic tape file is defined (see Chapter 10), FILENAME has the form "filename(vol)", where vol is the tape serial number.

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 UINFI and the external string function UINFS can be used to obtain such information. The functions must be specified:

        %EXTERNALINTEGERFNSPEC UINFI(%INTEGER TYPE)

        %EXTERNALSTRINGFNSPEC UINFS(%INTEGER TYPE)

The result obtained by calling either of these functions is a single item of information, specified by the value of TYPE. The values of TYPE which can be specified with these functions, and the meanings of the results obtained, are given in Table 12.1. Further TYPE values may be defined later, but the meanings given below will not be changed.

TYPE	UINFI result	UINFS result
1	File System number	Username
2	1=Foreground, 2=Background 3=Foreground OBEY	Delivery information
3	Current number of users	Start time of session
4	-	Current PROMPT string
5	Current CPU limit for a command (in seconds)	Current ACTIVEDIR
6	Maximum file size for DEFINE (in Kilobytes)	Subsystem version no
7	-	User's surname
15	Current setting of OPTION ITWIDTH	
16	Zero if OPTION NOBRACKETS is in effect, otherwise non-zero	-

Table 12.1: UINFI and UINFS Summary

Example

In the following program the user's name and delivery information are printed out, along with the number of user processes currently active on the System.

        %EXTERNALINTEGERFNSPEC UINFI(%INTEGER TYPE)
        %EXTERNALSTRINGFNSPEC UINFS(%INTEGER TYPE)
            •
            •
        PRINTSTRING("User: ".UINFS(1)); NEWLINE
        PRINTSTRING("Delivery: ".UINFS(2)); NEWLINE
        PRINTSTRING("Current no of users:")
        WRITE(UINFI(3),1); NEWLINE
            •
            •

Calling FORTRAN

The object files produced by the EMAS 2900 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 6.

Four file access methods are provided for IMP:

* Streams, used for character input/output: use routines READSYMBOL, WRITE, etc.

* Sequential binary file handling: use routines READSQ, WRITESQ, etc.

* Direct access binary file handling: use routines READDA, WRITEDA, etc.

* Store mapping: see Chapter 9.

The routines and functions provided by the IMP language implementation are described in Reference 7. The System-dependent aspects of these facilities are described below.

Linking channel numbers to files and devices

The command DEFINE (Chapter 7) is used to establish a link between a channel number used in a program and a particular file or output device. Note that the channel number must be a 1 or 2 digit number in the range 1-80.

As explained in Chapter 7, the access method to a file or device depends on the way in which the associated channel number is used in the program. In the case of IMP programs, the access method depends upon which I/O routines are used with the channel number defined. Thus any conflict between a file and its access method (e.g. an attempt to use a data file, rather than a character file, for stream input) will only be faulted when the program is executed, not when the DEFINE command is given.

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 ".IN").

* A character file (see Chapter 7).

Note that a data file cannot be used for character input to an IMP program: it must first be converted to a character file (see CONVERT, Chapter 7).

For output the following can be used:

* The interactive terminal (defined if necessary as ".OUT").

* 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(1,.IN)
        DEFINE(22,.OUT)
        DEFINE(17,FILEIN)
        DEFINE(80,.PP)

Default stream definitions

Table 12.2 shows the definitions that are established by default, in foreground and background mode:

Channel	Input/Output	Device Foreground	Background
0	Input	.IN (user's terminal)	.IN (Job file)
98	Input	.IN (user's terminal)	.IN (Job file)
0	Output	.OUT (user's terminal)	.OUT (.LP)
99	Output	.OUT (user's terminal)	.OUT (.LP)
95	Output	.PP	.PP
96	Output	.GP	.GP
97	Output	.CP	.CP

Table 12.2: IMP Default Channel Definitions

As indicated in Table 12.2, .IN, the primary input device, stands for the user's terminal in foreground mode, and for the job file in background mode; .OUT, the primary output device, stands for the user's terminal in foreground mode, and for the line printer appropriate to the source of the job in background mode (the local line printer, .LP, for background jobs initiated from foreground).

Line length limits are not imposed by EMAS 2900, but are in some cases by the devices themselves (e.g. line printers and card punches), or by the TCP (in the case of the user's interactive terminal). In the latter case, the limit can be varied by use of the TCP setmode facility (see Chapter 3 and above) to any value between 15 and 160 inclusive.

No line limit is imposed when a file is read from or written to using character I/O.

Note that stream 0 is exceptional in that it can be used for input and output simultaneously.

Size of stream output 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 both to files and output devices. The size is passed as the third parameter to DEFINE, and sets the size in Kilobytes (1024 bytes). The default is 255 Kbyte, the maximum currently 1023 Kbyte.

Examples:

        DEFINE(10,FILEA,500)
        DEFINE(11,.LP,100)

Notes

* The size required must be specified in Kilobytes. It is rounded up to 1 Kilobyte less than the next multiple of 4 Kilobytes.

* See Chapter 7 for further information about DEFINE.

* EMAS 2900 imposes no upper limit on the size of a file being sent to a remote device, although the communications network used might do. Contact your local Advisory Service for details of any such limits.

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 7). For binary 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
Binary Paper Tape Punch	.BPP	80
Graph Plotter	.GP	80
Matrix Plotter	.MP	300

Table 12.3: IMP Access to Binary Output Devices

Note that "spanning" across records is not used, so that a failure occurs if an attempt is made to write a record longer than the maximum record size for the output file.

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 for which a new file has been DEFINEd, a file is created and each byte of it is filled with a bit pattern which can be interpreted as meaning "unassigned". The size of the file is extracted from the third parameter passed to DEFINE. For example, if it were required to create a file of 10 records (each of 1024 bytes), the following DEFINE could be used:

        DEFINE(70,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 file (if new) and open it

If the size parameter of DEFINE is omitted a size of 255 Kbyte is assumed for a new direct access file. This will not later be reduced to the "used" size by the Subsystem (which is what happens in the case of character files), with a consequent waste of disc space. See also the section on the command DEFINE, in Chapter 7.

STORE MAP FILES

Accessing files by mapping them onto variables and arrays in a program is described in Chapter 9.

EFFICIENCY OF IMP WHEN USED WITH EMAS 2900

When writing IMP programs specifically for EMAS 2900 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(1:200,1:100)

  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 systematically, an attempt should be made to access the elements in the order in which they occur in store. For example, when such an array is cleared to zero the following should be used:

        %INTEGER I,J
        %CYCLE I=1,1,100
            %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.

This Chapter describes the environment in which a FORTRAN program runs on EMAS 2900. A compiler is provided for the FORTRAN IV language, which is fully described in References 4 and 8. The FORTRAN user, as against the IMP user, is at a disadvantage in exploiting some of the more sophisticated features of EMAS 2900, 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 11. When preparing such a source file using an interactive terminal the Tab control function (HT or CTRL+I - see Chapter 3) can be useful. The command PARM is used to set compile time options. The file may contain one or more program units (subprograms), of which at most one can be a main program. If after successful compilation the object file is analysed using ANALYSE, 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 the first 31 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. COMMON blocks in different subprograms are automatically equivalenced when a program is loaded. Each named COMMON block is equated with any other COMMON block of the same name, and is 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 must be the name of an object file which includes a main program. Before starting execution the object file must be loaded, as described in Chapter 11. 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 only allowed to proceed if the PARM LET has been specified (see Chapter 11). If it has been, processing will continue up to the first call of a routine which was not found. At this point an appropriate message is printed, followed by diagnostics and execution of a STOP.

System library

A FORTRAN system library is automatically searched for the intrinsic and mathematical function subprograms. Details of these subprograms are contained in Reference 6, and a summary table is included in References 4 and 8.

Graphics and other libraries

There are libraries of graphics routines and other specialist routines available on EMAS 2900 (Reference 3). Additionally the Numerical Algorithms Group (NAG) library is available to the FORTRAN programmer. Further information can be obtained from your local Advisory Service.

Accessing EMAS 2900 foreground commands

EMAS 2900 foreground commands can be accessed from within FORTRAN programs. Chapter 15 discusses some of the wider implications of doing this, but the mechanism is described here. EMAS 2900 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

 * 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; e.g.

        CALL EMASFC('DEFINE',6,'10,.LP23',8)

The first parameter is a literal constant which contains the name of the command to be 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(10,.LP23)

on the interactive terminal at Subsystem 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)

Note also that single quotes are used round text strings in FORTRAN (cf. IMP string delimiters).

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 there is a further 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 VALUES')
        CALL FPRMPT('No. of values:',14)
        READ(5,101) NOV
101     FORMAT(I2)
        CALL FPRMPT('Value:',6)
        DO 1 I=1,NOV
1       READ(5,101) VALUE(I)
          .
          .
```

The dialogue resulting on the interactive terminal might have the following appearance:

```
NOW TYPE IN CONTROL VALUES
No. of values: 3
Value:27
Value:10
Value:99
```

The FORTRAN implementation includes a "list-directed" READ facility which simplifies the operation of reading data from the interactive terminal. This is fully described in References 4 and 8.

Calling IMP

The object files produced by the FORTRAN and IMP compilers are compatible and it is possible, within the limits imposed by the parameter passing facilities of the two languages, 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 2900. The mechanisms for making calls from FORTRAN to IMP are described in Reference 6.

INPUT/OUTPUT

Access Methods

EMAS 2900 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

Table 13.1 indicates which file types and devices can be used for each access method. The file types are described in Chapter 7, the output devices in Chapter 2.

| | Sequential | | Direct Access | |
	With FORMAT	Without FORMAT	With FORMAT	Without FORMAT
Input	Terminal (.IN) Character File Data File	Data File	Data File	Data file
Output	Terminal (.OUT) Line Printer (.LP) Card Punch (.CP) Paper Tape Punch (.PP) Data File Character File	Binary Paper Tape Punch (.BPP) Graph Plotter (.GP)* Matrix Plotter (.MP)* Data File *routines provided for access	Data File	Data File

Table 13.1: Summary of FORTRAN I/O Access Methods

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 command CLEAR, are described fully in Chapter 7. When used with FORTRAN the normal form of the command should be

```
    DEFINE(n,filename)
or  DEFINE(n,device)
```

For example:

```
    DEFINE(23,DATA107)
    DEFINE(1,.LP)
```

Sequential input

It is possible to use the interactive terminal (.IN), a character file or a data file for input (see Table 13.1 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
```

(Note the carriage control character in the FORMAT statement used for output.) When run, this would appear on the interactive terminal (assuming default channel definitions) as:

```
    TITLE:FIRST EXPERIMENT
    FIRST EXPERIMENT
```

When a data file is being read, 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 Table 13.1 above). It is important to understand the effect of the carriage control character ("format effector") which is often put at the start of the output record. If the output is directed (by command DEFINE) to a device that can interpret this - for example, the interactive terminal or a line printer - then it is interpreted as explained in Reference 4. 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 command, the presence of carriage control characters must be indicated to LIST, so that the first character of the line is not printed but is used to control the line spacing of the printer. This is done by specifying the fifth parameter of LIST as FE (see Chapter 6). For example:

```
        .
        .
        .
100     FORMAT('1HEADING 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(8,.LP)

were typed before running this program the output would be printed on the line printer,
correctly interpreting the carriage control characters for "newpage" and "two new lines".
If however it were required to put the output in a file and then to list the file later,
it could be defined thus:

        DEFINE(8,OUTLP,,C)

Note that this would cause a character file to be created; if the C parameter were omitted
a data file would be created. LIST can handle either. If later the file were to be
listed using

        LIST(OUTLP,.LP,,,FE)

then the carriage control characters would be correctly interpreted. If, on the other
hand, the FE parameter were omitted from the LIST command, the listing would contain the
carriage 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 re-reading by another FORTRAN program, one need not
include a carriage 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.

* .NULL (the dummy file specification) cannot be used.

* The size and record length of the file are not taken from the DEFINE command: they are
  taken from the DEFINE FILE statement in the FORTRAN program 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 255 Kbyte.

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 Foreground	Background
5 6 7	Input Output Output	.IN (user's terminal) .OUT (user's terminal) .CP	.IN (Job file) .OUT (.LP) .CP

Table 13.2: FORTRAN Default Channel Definitions

Note that the meanings of .IN and .OUT depend upon whether the program is running in
foreground or background mode. Note also that the effect of channel 5 being defined 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 16). For example, the following program, compiled
into object file FORT73, could be run using the Job file shown:

```
        Program                        Job file

          .                              .
          .                              .
100   FORMAT(2I4)                    RUN(FORT73)
      READ(5,100) NUM,TOP             276 389
          .                              .
          .                              .
```

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 2900 routine
CLOSEF is available. Its parameter must be an INTEGER*4 constant or variable containing
the logical channel number of the file to be closed; for example:

        CALL CLOSEF(27)

This chapter describes the way in which an ALGOL 60 program can be compiled and run on
EMAS 2900. The implementation of ALGOL is described in References 5 and 9. Examples of
ALGOL in this chapter use the "ECMA" hardware representation, which for ALGOL is the
default on EMAS 2900 (corresponding to the PARM option QUOTES). Note that Reference 5
uses the "%" symbol to denote keywords, as do IMP examples in this manual.

Compilation

An ALGOL source file can be compiled using the command ALGOL, as described in Chapter 11.
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 procedures not contained in
a 'BEGIN' - 'END' block. Such procedures are not intended to be executed independently,
but are for calling from other ALGOL programs or procedures. This topic is covered fully
in Reference 5.

Running an ALGOL program

Before execution, an ALGOL program is loaded as described in Chapter 11. If there are any
references to separately compiled procedures that cannot be satisfied, then their names
are listed and execution is inhibited unless the PARM LET has been specified (see Chapter
11); otherwise execution commences. If a call is made on an unsatisfied reference the run
terminates with an appropriate failure message, followed by diagnostic information.

Accessing EMAS 2900 Subsystem commands

Currently there is no facility for allowing an ALGOL program to make a call on an EMAS
2900 Subsystem command. The restriction is due to 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 the Subsystem command DEFINE using the same
parameter.

```
%EXTERNALROUTINE ADEFINE(%STRINGNAME AS)
%EXTERNALROUTINESPEC DEFINE(%STRING (255) S)
DEFINE(AS);! Call on DEFINE using parameter 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('('3,LP')')
    .
    .
'END'
```

Chapter 15 includes further information about calling EMAS 2900 commands.

Library routines

The ALGOL "standard functions" are described in References 5 and 9. Other mathematical procedures available are described in Reference 10. Note also that the Numerical Algorithms Group (NAG) library may be available to the Algol programmer. For further details consult the local Advisory Service.

Calling other IMP and FORTRAN routines

Reference 5 describes the mechanism by which ALGOL programs can access IMP and FORTRAN procedures.

ALGOL INPUT/OUTPUT

Stream, sequential access and direct access Input/Output are all available with EMAS 2900 ALGOL. The DEFINE command must be used to establish a link between a logical channel and a file or output device (see Chapter 7). As explained in Chapter 7, there is no command provided for creating data or character files: if a new filename is specified in a DEFINE command, the file will be created by the Subsystem when it is opened or written to by a program.

Streams

The ALGOL stream-handling facilities are similar to those of IMP, described in Chapter 12. The I/O procedures available are described fully in References 5 and 9. The following points should be noted about them:

* Many of the standard character I/O procedures provided have the same names as IMP routines. However their use is not always identical to that of the IMP equivalents (e.g. the READ procedure).

* In addition to the standard character I/O procedures, EMAS 2900 ALGOL allows for the use of a selection of the IFIP character I/O procedures proposed for ALGOL.

* In addition to the standard I/O procedures, EMAS 2900 ALGOL allows for the use of ICL 1900 Series ALGOL procedures, for compatibility with 1900 ALGOL programs.

* Strings are only provided in ALGOL for annotating output. They cannot be read in, or manipulated as in IMP, although they can be passed as parameters (see example above). In ALGOL programs, all space and newline characters are ignored, and so the characters "_" and "¬" are used in strings to represent space and newline respectively.

The following channel definitions for stream I/O are set up by default:

Channel	Input/Output	Device Foreground	Background
1	Output	.OUT (user's terminal)	.OUT (.LP)
2	Input	.IN (user's terminal)	.IN (Job file)

Table 14.1: ALGOL Default Channel Definitions

Note that limitations on the number of characters per line (i.e. between newline characters) are not imposed by EMAS 2900 but by the output devices themselves, or by the Terminal Control Processor (in the case of the user's interactive terminal). In the latter case, the limit can be varied by use of the TCP setmode facility (see Chapter 3) to any value between 15 and 160 inclusive.

No line limit is imposed when a file is used for input or output.

Sequential access

The binary sequential access procedures OPENSQ, PUTSQ, GETSQ, CLOSESQ are provided. These are equivalent to the IMP routines OPENSQ, WRITESQ, READSQ, CLOSESQ respectively. See Chapter 7 for details of the DEFINE command required to set up the correspondence between a channel number and a file; note that the defaults provided for parameters other than the channel number and filename are usually satisfactory.

Direct Access

The binary direct access procedures OPENDA, PUTDA, GETDA, CLOSEDA are provided. These are equivalent to the IMP routines OPENDA, WRITEDA, READDA, CLOSEDA respectively. See Chapter 7 for details of the DEFINE command required to set up the correspondence between a channel number and a file.

ALGOL direct access files must have fixed-length records, but the actual size of the record can be chosen by the programmer. (Cf. IMP, where the record size can only be 1024.) When new ALGOL direct access files are being DEFINEd, all four DEFINE parameters must be specified. For example:

        DEFINE(3,ALGOLDAFXYZ,4,F500)

The third parameter specifies the size of the file, in Kilobytes; its default value is 255. In the example above the size requested is 4 Kbyte, which will permit 8 records of 500 bytes each.

The record format (fourth parameter) must be specified, in the form Fn, where n is the size in bytes of each record.

A fundamental design intention of the Edinburgh Subsystem is that all the facilities 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 primarily for the IMP user; Chapters 13 and 14 explain how EMAS 2900 commands can be accessed from FORTRAN and ALGOL.

The section "The command interpreter", below, explains how each Subsystem commmand is in fact an IMP external routine with a single string parameter. The implications of this are as follows:

* IMP programs written by users can call external routines. Thus user programs can call Subsystem commands. The details of doing this are given in the sections "Calling commands from within programs" and "Use of the routine CALL".

* Users can write IMP external routines. Thus users can write commands. This is expanded in "Writing one's own commands".

* User-written commands are external routines and so can call other commands, as these are also external routines. This is also discussed in "Writing one's own commands".

* It is necessary for programs which call commands to be able to detect whether they have worked or not, and whether any error messages have been generated. The ways of doing this are explained in "Detecting errors in commands called from within programs".

Other sections of the chapter explain some consequences of calling commands from within programs - those relating to Input/Output, protection of files and protection of file definitions - and some restrictions imposed.

The command interpreter

The command interpreter is that part of the Edinburgh Subsystem which reads the text typed by an interactive user in response to a "Command:" prompt. It converts the text into two parts:

* a string of up to 31 characters, which constitutes the name of the command being called (truncated if necessary)

* a string of up to 255 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 11) and called with the parameter string provided passed to it. Thus for each command in the Subsystem there is a routine with the following specification:

        %EXTERNALROUTINESPEC command(%STRING(255) PARAMETER)

Calling commands from within programs

There are two methods of calling a command from within a program. The command in question may be a Subsystem or a user-written command (see "Writing one's own commands", below): in either case a call of an external routine is involved.

The first method of calling a command from within a program involves making a call direct to the external routine (i.e. the command), as follows:

a. Specify the routine with a %SPEC statement of the form given above: for example:

        %EXTERNALROUTINESPEC DEFINE(%STRING(255) S)

b. 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, part of a program is given which reads a file name from the interactive terminal and then calls DEFINE to link the file to channel 1.

```
%EXTERNALROUTINESPEC DEFINE(%STRING(255) S)
%STRING(11) FILE
    .
    .
READSTRING(FILE)
DEFINE("1,".FILE)
    .
    .
```

Note that some commands do not require a parameter, for example METER. In this case the command must still be specified with a string parameter and called with a null string as its actual parameter; otherwise the call will fail with the message "Wrong number of parameters".

Example:

```
%EXTERNALROUTINESPEC METER(%STRING(255) S)
    .
    .
METER("")
    .
    .
```

Use of the routine CALL

The second method of calling a command from within a program involves use of the IMP routine CALL. This must be specified as follows:

```
%EXTERNALROUTINESPEC CALL(%STRING(31) COMMAND, %STRING(255) PARAMETER)
```

CALL can be used to make a call to any of the Edinburgh Subsystem commands, and to any other external routines which take a single parameter of type %STRING (255). The effect of a call on this routine is to load (see Chapter 11) the specified routine and then to call it. The examples of calls on DEFINE and METER given earlier are thus equivalent to:

```
CALL("DEFINE", "1,".FILE)

CALL("METER", "")
```

There is an overhead involved in making the routine call in this way but this may be outweighed by the following advantages:

* It is only necessary to include a specification for CALL, not for all the target routines called. This is useful in situations where it is not known until run time which command is going to be called.

* If the routine being called is a user-provided one then it will be loaded by the CALL routine, called, and then unloaded. It could then be altered (by re-compilation) for a subsequent call.

The CALL routine is potentially a useful tool for users who wish to develop their own interface to the Edinburgh Subsystem.

Writing one's own commands

It is a straightforward matter to write one's own commands, since, as explained above, a command is merely an external routine with a single parameter, of type %STRING (255). The structure of IMP external files is described in Chapter 12 and in Reference 7. Commands can, but need not, call Subsystem commands or other user-written commands.

When a user-written command is called from an interactive terminal, its string parameter contains all the characters typed in brackets after the command name, with any spaces and newlines removed.

Since user-written commands are identical in form to Subsystem commands, they can be called from within a program as well as from an interactive terminal. The two methods of calling commands from within programs, viz. directly or using routine CALL, are both available.

Example:

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 could type:

      Command:I(PROG27)

The routine required could be as follows:

```
%EXTERNALROUTINE I(%STRING(255) FILE)
%EXTERNALROUTINESPEC IMP(%STRING(255) S)
    %UNLESS 0 < LENGTH(FILE) <= 10 %START; ! CHECK LENGTH OF FILE NAME
        PRINTSTRING("Invalid parameter to I")
        NEWLINE
        %RETURN
    %FINISH
    ! NOW CALL IMP COMPILER WITH NAMES FILE."S" AND FILE."Y"
    IMP(FILE."S,".FILE."Y,,.OUT"); ! FAULT OUTPUT TO .OUT
%END; ! OF ROUTINE I
```

Detecting errors in commands called from within programs

The routines and functions described in this section are specified as follows:

      %EXTERNALINTEGERFNSPEC RETURN CODE

      %EXTERNALROUTINESPEC SET RETURN CODE(%INTEGER N)

      %EXTERNALSTRINGFNSPEC SSFMESSAGE

      %EXTERNALROUTINESPEC SSFOFF

      %EXTERNALROUTINESPEC SSFON

When Subsystem commands typed on the interactive terminal fail they output appropriate error messages on the terminal. When called from within a program they output the same error messages on the currently selected output stream; in this case, however, the output of the messages can be suppressed by making a call on the external routine SSFOFF. In addition, whether SSFOFF has been called or not, any error message generated by a Subsystem command can be detected within the program calling the command by use of the string function SSFMESSAGE.

SSFOFF and SSFMESSAGE are applicable to Subsystem commands only. By contrast, a "return code" can be set by any program or routine (including a user-written command), and the final value of the return code set during the execution of the program or routine can then be examined by the calling program. The return code can be set in the following ways:

* By a standard Subsystem command. Each Subsystem command sets the return code to 0 if it has worked properly, and to a value greater than 0 if it has not worked properly. Note that if the Subsystem command involved a number of separate actions (e.g. deletion of a list of files) then the value of the return code would reflect the success or failure of the last action.

* By a call of the routine SET RETURN CODE. User-written programs or routines can call this routine; it takes a single integer parameter which must be in the range 0-4095.

* By the use of "STOP n" in FORTRAN. If "n" is omitted the return code is set to zero.

However the return code was set, by using the function RETURN CODE a program can determine what its value is and thus whether a program or routine which it has called has set it as expected.

Examples:

The first example shows how SSFOFF, SSFMESSAGE and RETURN CODE can be used:

```
%EXTERNALROUTINESPEC DEFINE(%STRING(255) S)
%EXTERNALROUTINESPEC SSFOFF
%EXTERNALINTEGERFNSPEC RETURN CODE
%EXTERNALSTRINGFNSPEC SSFMESSAGE
%EXTERNALROUTINESPEC PROMPT(%STRING(15) S)
%STRING(25) OUTPUT FILE

        SSFOFF; ! SUPPRESS STANDARD FAILURE MESSAGES
          .
          .
        PROMPT("OUTPUT:")
GOUT:   READSTRING(OUTPUT FILE)
        DEFINE("80,".OUTPUT FILE)
        %IF RETURN CODE#0 %START; ! SOME FAILURE IN DEFINE
            PRINTSTRING(SSFMESSAGE." TRY AGAIN.")
            NEWLINE
            -> GOUT
        %FINISH
          .
          .
```

Notes

* SSFOFF remains in effect until the return is made to command level, or until the converse routine SSFON is used.

* RETURN CODE and SSFMESSAGE should be interrogated immediately after the command to be checked has been called.

The next example gives a user-written command TESTIMP which compiles an IMP program and, if the compilation fails, sends the listing file to the line printer:

```
%EXTERNALINTEGERFNSPEC RETURN CODE
%EXTERNALROUTINESPEC IMP(%STRING(255) S)
%EXTERNALROUTINESPEC SEND(%STRING(255) S)

%EXTERNALROUTINE TESTIMP(%STRING(255) S)
! THE PARAMETER S SHOULD CONTAIN ONLY THE
! NAMES OF THE SOURCE AND OBJECT FILES.
    IMP(S); ! CALL THE COMPILER WITH THE SUPPLIED PARAMETERS.
    %IF RETURNCODE # 0 %THEN SEND("T#LIST")
%END; ! OF TESTIMP
```

In the next example a user-written command GO is given. In GO a program which sets the return code is compiled with all checks on and run with a small set of data. If it ends in order (i.e. calls SET RETURN CODE(0)) then it is re-compiled with PARM(OPT) and run again with a full set of data.

```
%EXTERNALROUTINESPEC DEFINE(%STRING(255) S)
%EXTERNALROUTINESPEC IMP(%STRING(255) S)
%EXTERNALROUTINESPEC PARM(%STRING(255) S)
%EXTERNALROUTINESPEC RUN(%STRING(255) S)
%EXTERNALROUTINESPEC SEND(%STRING(255) S)
%EXTERNALINTEGERFNSPEC RETURN CODE

%EXTERNALROUTINE GO(%STRING(255) S); ! THE PARAMETER S IS NOT USED.
    PARM (""); ! SET DEFAULT PARMS
    IMP("PROGS,PROGY")
    %IF RETURNCODE # 0 %THEN SEND("") %AND %RETURN
    ! I.E. COMPILER LISTING SENT TO LINE PRINTER IF COMPILATION FAILED.
    DEFINE ("1,TESTDATA"); ! USE THE TEST DATA
    RUN("PROGY"); ! THE TEST RUN
    %IF RETURNCODE # 0 %THEN %RETURN; ! THE PROGRAM FAILED
    PARM("OPT")
    IMP("PROGS,PROGY")
    DEFINE("1,FULLDATA")
    RUN("PROGY")
%END; ! OF GO
```

The source of the program being tested by GO might be as follows:

```
%BEGIN
%EXTERNALROUTINESPEC SET RETURN CODE(%INTEGER N)
%INTEGER I, M
    .
    .
    SETRETURNCODE(4)
    SELECTINPUT(1)
    READ(M)
    %CYCLE I=1,1,M
    .
    .
    SETRETURNCODE(0)
%ENDOFPROGRAM
```

Note that if close control of a sequence of commands is required, use of the job control language described in Chapter 16 might be preferable to writing a "one-off" IMP program.

Commands which read input or generate output

Many commands generate output and some, for example TELL, require input apart from their parameters. The Edinburgh 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.

* Output is normally sent to the currently selected output stream. However, if an explicit output file is specified as a parameter to a command then the file is used for any output generated by the command. The user's output stream is then re-selected before the return is made to his program. For example:

```
    SELECTOUTPUT(8)
    PRINTSTRING("About to call FILES."); NEWLINE
    FILES(",,OUT")
    PRINTSTRING("FILES called."); NEWLINE
```

The two PRINTSTRING messages are sent to stream 8. The FILES output is sent to file OUT.

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("80,OUT")
    SELECTOUTPUT(80)
    .
    .
    DESTROY("OUT")
```

In this situation the operation would fail with the message:

```
    DESTROY fails - inconsistent use of file OUT
```

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); ! Primary output selected
    CLOSESTREAM(80)
```

The same failure message would occur if any of the following types of activity were attempted in respect of a currently open data file:

* DISCONNECT

* RENAME

* OFFER

* any command that would write to the file; for example, FILES(",,OUT")

* any command that would alter a partitioned file one or more of whose members were currently open for input or were being executed

Apart from files which are currently in use for Input/Output operations, files which are currently loaded (see Chapter 11) are also protected from corruption. For example, if the object file RPROGY were to contain the statement

        IMP("RPROGS,RPROGY,.LP")

then this call of IMP would fail because an attempt would be 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 operation would fail as described above.

Protection of file definitions

File definitions set up by DEFINE are protected if the channel they reference is currently open. For example, the second DEFINE in the following:

        DEFINE("3,FIRST")
        SELECTINPUT(3)
            .
            .
        DEFINE("3,SECOND")
            .
            .

would fail with the message:

        DEFINE fails - attempt to re-define open channel

Again, use of CLOSESTREAM will solve this problem.

Restrictions in calling commands

* The command OBEY cannot be called from within a user program.

* Some of the "SIZE" parameters of the command OPTION (Chapter 17) relate to the environment of a running program. If any of these parameters is specified when OPTION is called from within a program, its effect is deferred until execution of the program has terminated.

Conclusion

The facilities described above, together with those described in Chapter 12 (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, 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.

Although the most usual way of using EMAS 2900 is through an interactive terminal (foreground mode), one can also run work on the System without an interactive terminal. This is known as running work in background mode. Whilst running in foreground mode the user can prepare a job file and put it into the queue of jobs waiting to be run in background mode. It is also possible to submit a background job on cards or paper tape. This is described below (see "BACKGROUND JOBS - JOB INTERPRETER").

One of the tasks of the SPOOLR System process is to schedule the running of this work, according to criteria such as optimum loading and time of day.

A background job can consist of a sequence of foreground commands, plus data as appropriate; it would then normally be processed by the same component of the Edinburgh Subsystem as deals with commands typed by an interactive user, viz. the Command Interpreter, whose operation is described at the start of Chapter 15. The preparation and submission of this type of job is described below in "BACKGROUND JOBS - COMMAND INTERPRETER".

However, a background job can instead be processed by the Job Interpreter. This is also part of the Edinburgh Subsystem and operates in a similar way to the command interpreter; but it has certain additional features designed to facilitate the control of background jobs. It is described below, in "BACKGROUND JOBS - JOB INTERPRETER".

BACKGROUND JOBS - COMMAND INTERPRETER

Preparing the job file

The job file can be read in on cards or paper tape (see Chapter 2), or created using a text editor (see Chapter 8). It should contain a sequence of commands and data, as might be typed during 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(2,.IN)
RUN(TESTBY)
"TEST RUN" 4 7 8 8 -99
```

Notes

* The format of the file is identical to that used for the command OBEY (see Chapter 17).

* The user does not need to provide any job identification information. This is added by the Subsystem, and printed out - for reference - by command DETACH (see below) when the job is submitted to the background job queue.

* It is not necessary to terminate the job with the command STOP or QUIT as one would terminate a foreground session. The job is terminated automatically when end-of-file is reached, or if the command STOP or QUIT is included.

* In order to read data from the file itself the relevant stream must be defined to be .IN (see the DEFINE in the example above), since the file is a direct replacement for the user typing at his terminal.

* As in interactive use of the System, the failure of a command in the job does not cause the job to terminate; cf. the behaviour of the job interpreter (described below).

Running the job

        DETACH(<u>file</u>, cputime, parameter file)

The command DETACH can be used to put a job to be processed by the command interpreter
into the background job queue. The timing of the execution of the job depends upon the
scheduling parameters specified (see below) and on the job scheduling rules operated by
the Service Manager; see "Job scheduling", below, and Appendix 5.

file

    The filename of the job file: this is the file prepared as above and must be specified.
    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)

cputime

    A CPU time limit for the <u>whole</u> job. This information is used for scheduling purposes,
    as well as imposing a limit on the time used for the job. Refer to Appendix 5 for
    details of the default and maximum values for this parameter and the units used on
    various EMAS 2900 services. Note that if any of the individual commands in the job is
    likely to take more than the default command time limit it will be necessary to include
    a call of the command CPULIMIT (see Chapter 17) in the job file, as well as giving the
    estimated total job time as this parameter.

parameter file

    The name of a character file which contains document scheduling parameter assignments,
    one to a line, and terminated by ".END" on a line by itself. The available scheduling
    parameters are described in Table 2.1, Chapter 2.

    Alternatively, the parameter file can be specified as ".IN"; the user will then be
    prompted on his interactive terminal for scheduling parameter assignments. As before,
    this is terminated by replying ".END".

    There is no default for this parameter: if it is not specified the default values for
    the document scheduling parameters are assumed.

Example:
        Command:DETACH(CF,10,.IN)
        DOC param:AFTER=22.00.00
        DOC param:NAME=MAIN RUN
        DOC param:.END

If the job is detached successfully a confirmatory message is output which also informs
the user what identifier (a number in the range 1-1000) has been given to the job. This
is also the identifier given in output from the command DOCUMENTS (see Chapter 17), which
can be used to locate a job document in the background job queue.

Failures using DETACH

See Appendix 3: Edinburgh Subsystem Error Messages. In addition, failures will occur when
DETACH is invoked if a time limit outwith the permitted range is used. See Appendix 5 for
defined limits on various EMAS 2900 services.

Removing a job from the job queue

        DELETEDOC(<u>identifier</u>)

This command allows the user to delete a job which is currently queued. It takes a single
parameter, a number in the range 1-1000 which uniquely identifies the job document. This
identifier is output when a job is DETACHed, and can be determined for any particular job
document by use of the DOCUMENTS command (see Chapter 17).

Job scheduling

The precise rules for running background work are liable to change depending on the loading of the machine. The broad aim is normally to give priority to interactive use, although this is at the System Manager's discretion. It is, for example, possible to modify the balance between interactive and batch use of the System, and to take account of interactive users whose programs are in fact behaving like batch jobs. It is also possible to run background and foreground work concurrently for one user; however this again is at the discretion of the System Manager.

Controlling background jobs

The recommended method of controlling background jobs in EMAS 2900, when using the command interpreter, is to write an IMP or FORTRAN program, or command. The program could call Subsystem commands or user-written commands, as necessary, and could use the function RETURN CODE to determine the success or otherwise of commands called. Further details and some examples are given in Chapter 15.

If these techniques are not appropriate then use of the job interpreter (described below) should be considered.

BACKGROUND JOBS - JOB INTERPRETER

Access to the job control language

As explained at the start of Chapter 15 and above, commands and command parameters typed by an interactive user of EMAS 2900 are analysed by the command interpreter, a component of the Edinburgh Subsystem. It analyses the command line typed, generates an error message if the syntax is faulty, or otherwise causes the specified command to be sought, loaded (if necessary) and entered. When the command execution is complete, a return is made to the command interpreter and the "Command:" prompt appears - the user is again "at command level".

However, a more conventional "batch" job control language has also been implemented, by means of a separate job interpreter.

When a user logs on to the System he will initially be using the command interpreter. If he uses OBEY to cause a file of commands to be obeyed in foreground mode, then the command interpreter will be used during the OBEY. If he uses DETACH to detach a file of commands to the background job queue, then again the command interpreter will be used.

If, however, a background job is submitted to the System as a deck of cards or paper tape then the job interpreter will be used. If a file of commands is detached from an interactive session by means of the command DETACHJOB (described below), then the job interpreter will be used. If a file of commands is obeyed during an interactive session by use of the command OBEYJOB (described below), then again the job interpreter will be used.

Further, if during an interactive session the user gives the command OBEYJOB without a parameter (i.e. no file of job control statements specified), then the effect is to cause a switch to the job interpreter for the rest of the interactive session, or until a switch back is caused by use of the job control language command .ENDJOB.

Note that the job interpreter is so called because it accepts job control language statements, not because it is accessible only in background mode. Indeed, as explained above, both interpreters are accessible from foreground and background modes of operation.

Subsystem commands relating to the job interpreter

DETACHJOB(<u>file</u>, cputime, parameter file)

This command is identical to DETACH (described above) except that the job is processed by the job interpreter, not the command interpreter.

```
        OBEYJOB(file, out)
```

This command is identical to OBEY (described in Chapter 17), except that:

* It causes the job interpreter, not the command interpreter, to be used.

* The parameter (the file of job control statements) is optional. If omitted, the effect of the command is to cause a switch from the command interpreter to the job interpreter. This is explained more fully below.

Submitting a card job

The form of the card deck should be as follows:

```
        //DOC parameter=value, parameter=value, ...

            job deck

        //DOC
```

This deck should be preceded by an appropriate EMAS 2900 Input Card.

As explained in Chapter 2, the first and last cards both have //DOC in their first 5 columns; these cards delimit the job deck. Note that there are no spaces between // and DOC. A number of parameters can be specified on the first card. They are described in Table 2.1 (Chapter 2).

Note that the USER, PASS and DEST parameters must be given when a job deck is submitted.

Summary of access to interpreters

Logging on	causes use of	command interpreter while logged on
OBEY	causes use of	command interpreter while file is obeyed
DETACH	causes use of	command interpreter while job is executed
OBEYJOB	causes use of	job interpreter while file is obeyed, or until further notice if no file is given
DETACHJOB	causes use of	job interpreter while job is executed
//DOC ...,DEST=BATCH,.. (any job in the background queue which has not been DETACHed; e.g. a card job)	causes use of	job interpreter while job is executed
.ENDJOB	causes use of	command interpreter while in foreground mode or causes background job to terminate

The access arrangements have the following implications:

* Users who do not require the facilities provided by the job control language need not be aware of its existence.

* Users may try out a file of job control language commands during an interactive session, by use of the command OBEYJOB. Once the file has been validated it can then be detached to the background job queue by use of the command DETACHJOB.

* The job interpreter can be used interactively, if the command OBEYJOB is first given without a parameter. Thus the user can test his understanding of the job control language by typing in various constructions and observing the results. In particular, he can declare and use macros (described below).

Note however that for interactive operation the command interpreter is more efficient than the job interpreter, and should be used if the additional control facilities of the job control language are not required.

16-4

The job control language

The EMAS 2900 job control language provides facilities for controlling background jobs.
The most important of these facilities are concerned with:

* making the progress of a job dependent on the results of previous steps

* allowing data to be included simply in a job

* defining "macros" to replace sequences of commands which are frequently repeated

Example:

```
.WHENEVER CMNDFAIL .GOTO NOGOOD
.WHENEVER JOBFAIL .GOTO NOGOOD
@
DEFINE (10,TESTFILEA)              @ A test file of data
DEFINE (20,TESTOUT)
RUN (ERYZ25.PROG1)
.IF RESULT>0 .GOTO TESTFAILED
@ (Presumably RESULT was set by use of SETRETURNCODE in IMP,
@  or STOP n in FORTRAN.)
@
DESTROY (TESTOUT)
DEFINE (10,BIGFILE)                @ The big file of data
DEFINE (20,BIGOUT)
RUN (ERYZ25.PROG1)
@
.IF RESULT=0 .THEN SEND (BIGOUT)   @ Big run was successful
.ELSE MESSOUT (Big_run_failed) ·
@ MESSOUT is not a Subsystem command - perhaps it was written by this user.
.ENDJOB
@
TESTFAILED: MESSOUT (Test_run_failed)
SEND (TESTOUT)
.ENDJOB
@
NOGOOD: MESSOUT (Subsystem_command_failed,_or_job_syntax_error)
.ENDJOB
```

A number of points arise from this example:

* Edinburgh Subsystem commands are, in most cases, valid statements in the job control
  language. Thus, for example, a valid file of commands intended to be OBEYed is usually
  also a valid set of job control language statements. The equivalence between the
  interpreters only breaks down in cases where symbols defined to have special meanings
  to the job interpreter are involved; e.g. the @ symbol.

  If all the commands work successfully then the effect should be the same under both
  interpreters. If, however, a command fails, the actions of the two interpreters may
  well be different. This is explained further below.

* Certain capitalised words preceded by a period have special meanings in the language.
  Since Edinburgh Subsystem command names must start with a letter, there is no
  ambiguity.

* Comments are permitted in the language.

* Certain "conditionals" are provided which are set on the occurrence of various types of
  error. In the above example, CMNDFAIL and JOBFAIL are conditionals.

Summary of the job control language

In this language:

* "newline" normally marks the end of a statement, but .C at the end of a line indicates
  that the statement is continued on the next line.

* "space" separates the elements of a statement.

* Special control words are marked by an initial period; e.g. .IF .

* The user of the Edinburgh Subsystem can choose to give his commands with or without brackets round the parameters. This is done by specifying BRACKETS or NOBRACKETS, as appropriate, to the command OPTION (see Chapter 17). The method chosen by the user applies to commands in the job control language, and also to macro calls. (Macros are described below.)

* @ is the "comment symbol", marking the start of a comment. The comment is terminated by the next @ symbol or newline symbol, whichever comes first. A comment can appear in a statement wherever a space symbol is allowed.

* .DATA and .ED are the opening and closing data delimiters.

* .A at the end of a line causes the whole of the <u>statement</u> to be ignored ("abandoned").

These are the permissible constructions in the language:

    (a)   call

    (b)   .GOTO   label

    (c)   .IF   condition   .GOTO label

    (d)   .IF   condition   .THEN call

    (e)   .IF   condition   .THEN call
           .ELSE   call

    (f)   .IF   condition   .START
                     call
                     call
                     .
                     .
           .FINISH

    (g)   .IF   condition   .START
                     call
                     call
                     .
                     .
           .ELSE
                     call
                     call
                     .
                     .
           .FINISH

    (h)   .WHENEVER condition   .GOTO label

    (i)   .WHENEVER condition   .THEN call

    (j)   .WHENEVER condition   .START
                    call
                    call
                    .
                    .
          .FINISH

    (k)   null statement

Notes

* A "call" may be a routine call (which includes all standard Edinburgh Subsystem commands - indeed, all commands which may be called in foreground mode), one of the job control language special commands (see "Special commands provided", below), or a macro call.

* The first or only line of a construction may be labelled, with no more than one label. A label consists of up to 11 alphanumeric characters of which the first must be a letter. A null statement (i.e. nothing at all) may be labelled, thus enabling a label to appear on a line by itself. For example:

    label:
        .IF condition .THEN call
        .ELSE call

The colon must follow the label immediately - no intervening spaces are allowed.

* Within a .START .. .FINISH block, or a .START .. .ELSE .. .FINISH block, only calls are permitted. In particular, .START .. .FINISH blocks cannot be nested.

* A condition can be a variable name followed by a comparator followed by an integer; e.g. RESULT=4. Variables permitted are RESULT, RESULTA, RESULTB, RESULTC, RESULTD. Neither compound conditions nor arithmetic expressions are allowed. See "RESULT variables", below.

* A condition can also be one of the System-provided conditionals, e.g. PROGFAIL. These are described below.

* "Backward .GOTOs" are permitted only within macros (described below).

.WHENEVER

The job interpreter tests automatically for certain failures whenever they can occur. The purpose of ".WHENEVER" is to allow the user to override the interpreter's normal action when such a failure condition is detected. ".WHENEVER" cannot be used to test for other conditions.

Note that if a .WHENEVER statement is not used to specify what should be done when a particular failure occurs, the occurrence of that failure will cause diagnostics to be generated followed by termination of the job; i.e. the default is

        .WHENEVER ANYFAIL .THEN .ENDJOB

(ANYFAIL and .ENDJOB are described below.) A .WHENEVER statement can be used to suppress this termination (but not the generation of diagnostics). The failure conditions tested for are described in "Conditionals", below.

When a .WHENEVER statement is activated, the effect is as though the associated call, .GOTO or .START .. .FINISH block has been planted after the statement which caused the activation. The .WHENEVER statement must have been encountered by the job interpreter prior to the occurrence of the failure condition it is intended to handle; its placement is otherwise unimportant.

A ".WHENEVER condition .GOTO label" statement is only used once - further occurrences of the same fault are treated as if no ".WHENEVER" had been set up.

A job may have several .WHENEVERs for the same condition. The latest .WHENEVER processed by the interpreter is the one used when a fault occurs.

Conditionals

The job interpreter allows you to test the success of each step of a job, and to control the progress of the job depending on the results of previous steps.

As explained above, the interpreter automatically tests for certain failure conditions after each step and sets a number of System-provided "conditionals" accordingly. You may specify what is to be done when one of these conditions arises, by using ".WHENEVER ....".

The conditions tested automatically, and the System-provided conditionals set as a result, are as follows:

PROGFAIL   E.g., array bounds exceeded

CMNDFAIL   Non-zero return code from a <u>Subsystem</u> command
             (not from a user-written command)

LOADFAIL   Loader unable to find or load a routine called from the job control

JOBFAIL   Syntax errors in the job control

ANYFAIL   Set as a result of <u>any</u> of the above failures occurring

In addition, there is the following conditional:

ONLINE   Set in accordance with the way in which the job interpreter was
             invoked

Unlike the other conditionals, ONLINE being "true" does not count as a failure.

A conditional is set to "false" (i.e. no failure) just prior to any action which might
cause the relevant failure to occur. If the failure does occur then the conditional is
set to "true". It is not reset until another such failure could occur. Thus, for
example, once CMNDFAIL has been set to "true" (because of a Subsystem command failure) it
will not be reset to "false" until just before the execution of another Subsystem command.

JOBFAIL is reset prior to the interpreter's analysis of <u>each</u> line of the job file. Thus
it is pointless to test JOBFAIL with an .IF statement, since it would be reset prior to
the analysis of the .IF statement. It can be tested, however, by use of a .WHENEVER
statement.

To control the handling of error conditions, you may use .WHENEVER in three ways:

    1) with .GOTO:

            .WHENEVER JOBFAIL .GOTO FBX

    2) with a simple call:

            .WHENEVER CMNDFAIL .THEN LIST (BASE)

  or 3) with a sequence of calls:

            .WHENEVER PROGFAIL .START
               LIST (WORKFILE, .LP)
               COPY (BASE, WORKFILE)
            .FINISH

If you want to have errors ignored (apart from a diagnostic report) you
can specify .CONTINUE as a simple call:

            .WHENEVER CMNDFAIL .THEN .CONTINUE

If you want to nominate the same action for all failures, you can specify (for instance)

            .WHENEVER ANYFAIL .GOTO FBX

A failure condition arising from an individual statement can also be tested, by placing an
.IF statement after the relevant call. Note, however, that the normal action of the
interpreter is to terminate the job on the occurrence of a failure, so that in order to
enable the .IF statement to be obeyed, an appropriate .WHENEVER statement must already
have been given. For example:

            .WHENEVER CMNDFAIL .THEN .CONTINUE
                .
                .
                .
            IMP (PADG, PADC, PADL)
            .IF CMNDFAIL .GOTO NORUN

(but, as explained above, you <u>cannot</u> use .IF to test for JOBFAIL)

RESULT variables

The System-provided integer variable RESULT can be set by a user program, by use of the external routine SET RETURN CODE (IMP), or by use of the "STOP n" statement (FORTRAN). It is not set by Subsystem commands; their success can be tested by use of CMNDFAIL.

The value of RESULT can be tested. For example:

```
RUN (POTTO)
.IF RESULT=1 .THEN DEFINE (7, WHEEL)
.ELSE DEFINE (7, ROPE)
```

In these tests, the permissible comparators are <, <=, =, >=, > and <> (meaning "not equals"). The variable name must appear to the left, and an integer to the right, of the comparator. Only one such comparison can be specified - compound conditions are not permitted.

If you do not want to test RESULT immediately you can

```
.SAVERESULT (RESULTA)
```

and test RESULTA later in the job, using ".IF". The variables RESULTB, RESULTC and RESULTD are also available for the same purpose.

Notes on the use of RESULT

* RESULT can be set by user programs, user-provided commands, and packages.

* RESULT non-zero is not treated as a failure condition.

* RESULT cannot be tested by .WHENEVER.

* You do not need to have specified a .WHENEVER statement before you can try to test RESULT with an .IF statement; this is different from the way that the conditionals (e.g. PROGFAIL) work.

Special commands provided

1)   .SAVERESULT (RESULTx)

    where x is A, B, C or D. As explained above, this permits the value of RESULT to be stored, for later use.

2)   .CONTINUE

    A null statement; useful in cases such as

    .WHENEVER ANYFAIL .THEN .CONTINUE

3)   .ENDJOB

    Causes the job to terminate (or, if ONLINE, control to be handed back to the command interpreter; see "Access to the job control language", below).

4)   .INPUT

    Enables a file to be created from in-line data; described below.

In-line data

With the command interpreter, in-line data can be introduced by the use of .IN, thus:

    DEFINE(4,.IN)

This means that channel 4 is associated with the interactive terminal, and a program reading from channel 4 would take its input from the terminal (or from a file which was being OBEYed). The same mechanism applies to a job detached to the background job queue by means of DETACH, the job file taking the place of the interactive terminal.

There are two deficiencies in this scheme, which affect particularly the execution of a job file:

1) The placement of the appropriate data within the job file is critical - it cannot be given earlier, since the program will read whatever appears next in the job file at the time it executes. If two channels were associated with .IN, which is permissible if somewhat eccentric, then the user would have to intersperse the data to be read via each channel exactly in accordance with the way in which the program made use of the two channels.

2) There is no way of delimiting the data. Thus if the program read more or fewer lines from the job file than the user expected, either some subsequent command lines would be treated as data, or excess data lines would be treated as subsequent commands.

To remove these deficiencies, another way of specifying in-line data has been provided. It is available only with the job interpreter. Consider the following:

    DEFINE (4,*)
    .DATA
      .
      .
    (lines of data)
      .
      .
    .ED

The effect of such a sequence appearing in the job file is that the lines of data delimited by .DATA and .ED are read by the job interpreter and treated as a (nameless) file associated with channel 4. Thereafter, any program invoked within the job which read from channel 4 would access the given data.

Notes:

* The position in the job file of the specification of the data is not critical; the DEFINE must merely have been encountered by the job interpreter before any attempt is made by a program to use the channel DEFINEd.

* If a program attempts to read past the .ED, the message "INPUT ENDED" (as for a normal file) will be given. If the program does not read all the data given between .DATA and .ED, then the rest is ignored - it does not cause subsequent confusion.

* A set of data specified as above can be re-used without respecification. Channel 4 will refer to the given data until it is reDEFINEd.

* Several such input "files" can be defined (with respect to different channels) and used simultaneously by subsequent programs.

* The .DATA and .ED delimiters must appear on lines by themselves, without preceding spaces.

* The commands invoking the compilers, ALGOL, FORTE and IMP, can have the source file parameter specified as *, meaning that the source immediately follows, delimited as before by .DATA and .ED.

.INPUT

In addition to the new method of specifying in-line data described above, the job
interpreter also provides the command .INPUT, which enables a file to be created from
in-line data. The form of its use is as follows:

```
.INPUT (filename)
.DATA
    .
    .
    .
(lines of data)
    .
    .
    .
.ED
```

Notes

* A character file is the only type which can be created using .INPUT.

* As before, the .DATA and .ED delimiters must appear on lines by themselves, without
  preceding spaces.

* A file of the given name <u>must not</u> already exist.

* It is not possible to create a file in another user's process by use of .INPUT.

Example: use of '*'

This gives an example of the use of '*' to input a source IMP program, which is compiled.
The input text is supplied between the delimiters .DATA and .ED.

Thereafter, some in-line data is associated with channel 5, and the program is executed.

```
IMP (*,WRITEY)
.DATA
    %BEGIN
    %EXTERNAL %ROUTINE %SPEC SET RETURN CODE (%INTEGER N)
    %INTEGER LINES, LIM
    %INTEGER C
    %ON %EVENT 9 %START
        %STOP
    %FINISH
    SELECT INPUT (5)
    READ (LIM)
    READ SYMBOL (C) %UNTIL C=NL
    LINES = 0
    %WHILE LINES<LIM %CYCLE
        READ SYMBOL (C); PRINT SYMBOL (C)
        %IF C=NL %THEN LINES = LINES + 1
    %REPEAT
    SET RETURN CODE (LIM)
    %END %OF %PROGRAM
.ED
@
@ NOW AN EXAMPLE USING '*' WITH THE 'DEFINE' COMMAND.
DEFINE (5,*)
.DATA
6
THE LAMB AND FLAG
THE CHEQUERS
THE ANCHOR
THE RISING SUN
THE GEORGE AND DRAGON
THE EAGLE AND CHILD
.ED
@
@ NOW WE RUN A USER PROGRAM USING THE DATA 'DEFINE'D ABOVE:
RUN (WRITEY)
@
@ TO TERMINATE THE JOB:
.ENDJOB
```

The line printer output produced as a result of the execution of this job would be as follows:

```
        IMP (*,WRITEY)
         18 Statements compiled
        @
        @ NOW AN EXAMPLE USING '*' WITH THE 'DEFINE' COMMAND.
        DEFINE (5,*)
        @
        @ NOW WE RUN A USER PROGRAM USING THE DATA 'DEFINE'D ABOVE:
        RUN (WRITEY)
        THE LAMB AND FLAG
        THE CHEQUERS
        THE ANCHOR
        THE RISING SUN
        THE GEORGE AND DRAGON
        THE EAGLE AND CHILD
        **** Return code =            6
        @
        @ TO TERMINATE THE JOB:
        .ENDJOB
```

Macros

A macro is a named piece of text, which can be made accessible to the job interpreter. When a macro call is "obeyed", the job interpreter stops reading text from the current position and takes further input instead from the macro text. When the macro text has been exhausted, the interpreter takes up again from where it left off in the original text.

A macro consists of a macro header line, which is a single line, followed by the macro text, which is any text at all. For example,

```
        TRYPASCAL                                macro header line
        PASCAL (MYSOURCE, MYOBJ, MYLIST) )
        DEFINE (8, MYDATA)                 |     macro text
        RUN (MYOBJ)                        |
        LIST (MYLIST)                      )
```

If the job interpreter were processing a job control file which contained the text

```
        TRYPASCAL
        DESTROY (MYOBJ)
```

then it would process the lines

```
        PASCAL (MYSOURCE, MYOBJ, MYLIST)
          .

          .
        LIST (MYLIST)
```

from the macro text, before returning to the control file and processing the line

```
        DESTROY (MYOBJ)
```

A macro can exist in a separate file or it may be declared in a job. To be declared in a job, it must be presented between .MACRO and .MACEND, thus:

```
        .MACRO
        TRYPASCAL
        PASCAL (MYSOURCE, MYOBJ, MYLIST)
        DEFINE (8, MYDATA)
        RUN (MYOBJ)
        LIST (MYLIST)
        .MACEND
```

The delimiter .MACEND must not be preceded by any spaces.

Once the macro definition has been encountered by the job interpreter, it can be called, as described above.

A macro can be redefined.

A macro can also be stored in a character file or character member of a partitioned file. Any of the methods available for creating or modifying character files (e.g. text editor, .INPUT) can be used. The following points should be noted:

* Only one macro can be held in one file or member of a partitioned file.

* The .MACRO and .MACEND lines must not be present in the file.

* The first text line of the file must have the form of a macro header line. This is the only requirement regarding the contents of the file.

* For the macro to be accessible to the job interpreter, the file containing it must be inserted into the user's active directory by use of the command INSERTMACRO, described below.

* The command REMOVEMACRO, described below, is used to remove a macro from the active directory.

Subsystem commands relating to macros

        INSERTMACRO(filelist)

As explained above, a single macro can be held in a character file or character member of a partitioned file. The command INSERTMACRO is used to insert one or a number of such character files into the user's active directory. The list of filenames must be separated by commas.

        REMOVEMACRO(filelist)

This command is used to remove one or a number of files containing macros from the user's active directory. The list of filenames must be separated by commas.

Macro parameters

If part of the macro text is not known at the time the macro is declared (or filed), that part may be represented as "%%keyword". "keyword" is any string of one to eleven upper case letters or digits, of which the first must be a letter. The macro header must include the same keyword, without the %%:

        TRYPASCAL (SOURCE)
        PASCAL (%%SOURCE, MYOBJ, MYLIST)
        DEFINE (8, MYDATA)
        RUN (MYOBJ)
        LIST (MYLIST)

Several keywords may be used to represent different parts of the macro text, and the same keyword may be used more than once in the text (but not in the header).

        TRYPASCAL (SOURCE, OBJECT, DATA)
        PASCAL (%%SOURCE, %%OBJECT, MYLIST)
        DEFINE (8, %%DATA)
        RUN (%%OBJECT)
        LIST (MYLIST)

In the macro call, the appropriate bits of text can then be presented like this:

        TRYPASCAL (OBJECT=PASSY, SOURCE=EL42, DATA=OBAN)

If the interpreter obeyed this call, it would then process the following text:

```
PASCAL (EL42, PASSY, MYLIST)
DEFINE (8, OBAN)
RUN (PASSY)
LIST (MYLIST)
```

Note that, in the call, the parameters were not specified in the same order as in the
macro header. This is possible because the keywords identify which parameter is which.
To reduce the amount of typing involved, it is possible to leave out the keywords, but
then the parameters must be specified in the same order in the call as in the header.
Thus

```
TRYPASCAL (EL42, PASSY, OBAN)
```

would have exactly the same effect as

```
TRYPASCAL (OBJECT=PASSY, SOURCE=EL42, DATA=OBAN)
```

You need not specify values for all the parameters in a macro call. Those which you do
not specify will take "default values". These default values are defined in the macro
header, which can take a form such as

```
TRYPASCAL (SOURCE=PLUM, OBJECT=FIRST, DATA)
```

Here the default value for the keyword SOURCE is PLUM, and the default value for keyword
OBJECT is FIRST. No default is explicitly provided for DATA, so a null string will be
assumed.

Then if one called

```
TRYPASCAL (,FAL)
```

the values used for each keyword would be

```
SOURCE      PLUM
OBJECT      FAL
DATA        null string
```

If the macro text contained

```
PASCAL (%%SOURCE, %%OBJECT, MYLIST)
DEFINE (8, %%DATA)
```

then the job interpreter would actually process

```
PASCAL (PLUM, FAL, MYLIST)
DEFINE (8,)
```

and this last line would cause a failure, since DEFINE needs more than one parameter.

When a macro has a fairly large number of parameters, it is normal practice to arrange
that those which must always be specified appear first in the parameter list. A call on
the macro will typically specify all of those first parameters and one or two of the
others. It is convenient to be able to specify the first parameters "by position" and a
few of the remainder "by keyword".

This is possible in the job control language; for example, given a macro with header

```
TESTCODE (SOURCE, OBJECT, LIST, PARMS, CPULIM, TRAPLABEL)
```

one might call

```
TESTCODE (PASRCE, PAOBJ, TRAPLABEL=CANCEL, PARMS=NOARRAY)
```

If a macro text includes the characters %%, but the characters following are not one of
the declared keywords, no textual substitution takes place, and the characters %% will be
processed by the interpreter as they stand.

Long keywords may be abbreviated to (not less than) the first three characters in a call
of the macro (but nowhere else). The names of the keywords of one macro must therefore
all differ in their first three characters.

Example: defining a macro

```
.MACRO
STDCOMP (NAME=PROG, LANG=IMP)
%%LANG (%%NAMESRC, %%NAMEOBJ, %%NAMELIST, %%NAMEERR)
.IF CMNDFAIL .THEN LIST(%%NAMEERR)
.ELSE LIST(%%NAMELIST,.LP)
.MACEND

@ A simple call on the macro:
STDCOMP (PACK)

@ Equivalent to:
IMP (PACKSRCE, PACKOBJ, PACKLIST, PACKERR)
.IF CMNDFAIL .THEN LIST(PACKERR)
.ELSE LIST(PACKLIST,.LP)

@ Another call:
STDCOMP (LANG=FORTE, NAME=XOVR)
```

Note that it would be necessary to have included the statement

```
.WHENEVER CMNDFAIL .THEN .CONTINUE
```

earlier in the job.

Backward .GOTOs

These can appear only in macros.

The destination label in a backward .GOTO must be prefixed by a minus sign, e.g.

```
THERE:
    .
    .
    .
.GOTO -THERE
```

A backward .GOTO causes the job interpreter to start searching for the specified label from the beginning of the macro which it is currently obeying, instead of searching on from the current position.

BRACKETS and NOBRACKETS

The usual Edinburgh Subsystem options BRACKETS and NOBRACKETS apply to calls on macros and on commands in this language. Within a macro, the bracket option to be used in interpreting the macro text is determined by the form of the macro header. This means that a macro without parameters but using the bracket option must have a header of the following form:

```
MYMAC ()
```

On exit from the macro the option reverts to whatever it was before the call. Thus you can use a macro without knowing whether its text uses brackets or not. In your call on the macro, you use brackets or no brackets according to your current setting of the option, and regardless of the macro's internal use of brackets.

As explained above, when you define your own macros you need to indicate whether the macro text uses brackets or not. You do this by using brackets (or no brackets) in the macro header.

```
MYMAC PARM          <- header ->   MYMAC (PARM)
LIST %%PARM, .LP )                 ( LIST (%%PARM, .LP)
CHERISH %%PARM     }    text       { CHERISH (%%PARM)
FILES              )               ( FILES
```

Job control language examples

Several examples of the use of the job control language are given below. In each case the
contents of a job file are followed by the line printer output generated by the job
interpreter in executing the job. The //DOC statements required when a job is input on
cards or paper tape are not included below. However, see the examples in Chapter 2.

Example:

By default, a job will automatically terminate on any fault condition. If you want to
define any other action, you may override the automatic stop by means of the first
statement in the following example.

Thereafter, there is an attempt to compile a program with a deliberate mistake, to
demonstrate how to test for the result of a Subsystem command. In this command, the file
WRITEL will get the co:mpilation listing, and ERF will get any error messages.

```
.WHENEVER ANYFAIL .THEN .CONTINUE
@
IMP (*,WRITEY,WRITEL,ERF)
.DATA
    %BEGIN
    %EXTERNAL %ROUTINE %SPEC SET RETURN CODE (%INTEGER N)
    %INTEGER LINGES, LIM; ! THIS IS THE 'ERROR' - 'LINES' IS MIS-SPELT.
    %INTEGER C
    %ON %EVENT 9 %START
        %STOP
    %FINISH
    SELECT INPUT (5)
    READ (LIM)
    READ SYMBOL (C) %UNTIL C=NL
    LINES = 0
    %WHILE LINES<LIM %CYCLE
        READ SYMBOL (C); PRINT SYMBOL (C)
        %IF C=NL %THEN LINES = LINES + 1
    %REPEAT
    SET RETURN CODE (LIM)
    %END %OF %PROGRAM
.ED
@
@ NOW TO TEST FOR SUCCESS OR FAILURE:
.IF CMNDFAIL .START
    LIST (ERF) @ TO COPY THE ERROR LISTING INTO THE JOURNAL.
    .ENDJOB    @ TO STOP THE JOB IF THE COMPILATION FAILED.
.FINISH
@
@ CARRY ON HERE IF THE COMPILATION WAS SUCCESSFULL:
LIST (WRITEL,.LP)
@
@ SUPPLY DATA AND RUN THE PROGRAM:
DEFINE (5,*)
.DATA
4
THE WYNNSTAY ARMS
THE KING LUD
THE MASTER GUNNER
THE TRAFALGAR
.ED
RUN (WRITEY)
@
@ ANOTHER RESULT TEST, THIS TIME FOR THE RESULT OF A
@ USER PROGRAM:
.IF RESULT>0 .GOTO NONZERO
@
@ THAT MAY CAUSE THE NEXT COMMANDS TO BE SKIPPED.
    DESTROY (WRITEL)
    DESTROY (WRITEY)
NONZERO: FILES   @ TO SHOW WHETHER THOSE FILES WERE DELETED.
@
@ TO TERMINATE THE JOB:
.ENDJOB
```

Resulting line printer output:

```
- .WHENEVER ANYFAIL .THEN .CONTINUE
@
IMP (*,WRITEY,WRITEL,ERF)
 1 Fault in compilation
**** Fault trap: call  .CONTINUE
@
@ NOW TO TEST FOR SUCCESS OR FAILURE:
.IF CMNDFAIL .START
++++ Evaluating: " CMNDFAIL " yields TRUE
     LIST (ERF) @ TO COPY THE ERROR LISTING INTO THE JOURNAL.

*   11   FAULT 16 (NAME NOT SET)  LINES
     .ENDJOB    @ TO STOP THE JOB IF THE COMPILATION FAILED.
```

Example:

This time the .WHENEVER statement has been left out, so that the job will stop
automatically on any failure. Note however that a non-zero return code from a user
program or package is not regarded as a failure.

Note also that the ".IF CMNDFAIL .." statement is <u>pointless,</u> since a failure in the
compilation will terminate the job.

Finally, note that if no listing file had been specified, the default file T#LIST would
have been used - and would have been destroyed at the end of the job.

```
IMP (*,WRITEY,WRITEL,ERF)
.DATA
    %BEGIN
    %EXTERNAL %ROUTINE %SPEC SET RETURN CODE (%INTEGER N)
    %INTEGER LINGES, LIM; ! THIS IS THE 'ERROR' - 'LINES' IS MIS-SPELT.
    %INTEGER C
    %ON %EVENT 9 %START
        %STOP
    %FINISH
    SELECT INPUT (5)
    READ (LIM)
    READ SYMBOL (C) %UNTIL C=NL
    LINES = 0
    %WHILE LINES<LIM %CYCLE
        READ SYMBOL (C); PRINT SYMBOL (C)
        %IF C=NL %THEN LINES = LINES + 1
    %REPEAT
    SET RETURN CODE (LIM)
    %END %OF %PROGRAM
.ED
@
@ NOW TO TEST FOR SUCCESS OR FAILURE.
@ (HOWEVER THIS PART CANNOT BE REACHED, SINCE THE JOB WOULD ALREADY
@ HAVE BEEN TERMINATED IF A FAILURE HAD OCCURRED.)
.IF CMNDFAIL .START
    LIST (ERF) @ TO COPY THE ERROR LISTING INTO THE JOURNAL.
    .ENDJOB    @ TO STOP THE JOB IF THE COMPILATION FAILED.
.FINISH
@
@ CARRY ON HERE IF THE COMPILATION WAS SUCCESSFULL:
LIST (WRITEL,.LP)
@
@ SUPPLY DATA AND RUN THE PROGRAM:
DEFINE (5,*)
.DATA
4
THE WYNNSTAY ARMS
THE KING LUD
THE MASTER GUNNER
THE TRAFALGAR
.ED
RUN (WRITEY)
```

```
@
@ ANOTHER RESULT TEST, THIS TIME FOR THE RESULT OF A
@ USER PROGRAM:
.IF RESULT>0 .GOTO NONZERO
@
@ THAT MAY CAUSE THE NEXT COMMANDS TO BE SKIPPED.
     DESTROY (WRITEL)                    .
     DESTROY (WRITEY)
NONZERO:  FILES    @ TO SHOW WHETHER THOSE FILES WERE DELETED.
@
@ TO TERMINATE THE JOB:
.ENDJOB
```

Resulting line printer output:

```
IMP (*,WRITEY,WRITEL,ERF)
 1 Fault in compilation
**** Job terminated
```

Example:

This time the .WHENEVER statement is included. A FORTRAN program is compiled, and in this case the source text is correct.

```
.WHENEVER ANYFAIL .THEN .CONTINUE
FORTE (*,WRITEZ,WRITEL)
.DATA
       INTEGER B(20), N
 100   FORMAT (20A4)
 101   FORMAT (1X,20A4)
 102   FORMAT (1X,I3,' NAMES')
       N = 0
 1     READ (10,100,END=2) B
       N = N + 1
       WRITE (6,101) B
       GOTO 1
 2     IF (N.EQ.0) STOP 0
       WRITE (6,102) N
       STOP 1
       END
.ED
@
@ NOW TO TEST FOR SUCCESS OR FAILURE:
.IF CMNDFAIL .START
     LIST (WRITEL) @ TO COPY THE ERROR LISTING INTO THE JOURNAL.
     .ENDJOB     @ TO STOP THE JOB IF THE COMPILATION FAILED.
.FINISH
@
@ CARRY ON HERE IF THE COMPILATION WAS SUCCESSFULL:
LIST (WRITEL,.LP)
@
@ SUPPLY DATA AND RUN THE PROGRAM:
DEFINE (10,*)
.DATA
THE BAG OF NAILS
THE WYNNSTAY ARMS
THE KING LUD
THE MASTER GUNNER
THE TRAFALGAR
.ED
RUN (WRITEZ)
@
@ ANOTHER RESULT TEST, THIS TIME FOR THE RESULT OF A
@ USER PROGRAM:
.IF RESULT>0 .GOTO NONZERO
@
@ THAT MAY CAUSE THE NEXT COMMANDS TO BE SKIPPED.
     DESTROY (WRITEL)
     DESTROY (WRITEZ)
```

```
NONZERO: FILES    @ TO SHOW WHETHER THOSE FILES WERE DELETED.
@
@ TO TERMINATE THE JOB:
.ENDJOB
```

Resulting line printer output:

```
.WHENEVER ANYFAIL .THEN .CONTINUE
FORTE (*,WRITEZ,WRITEL)
 13 Statements compiled
@
@ NOW TO TEST FOR SUCCESS OR FAILURE:
.IF CMNDFAIL .START
++++ Evaluating: " CMNDFAIL " yields FALSE
     LIST (WRITEL) @ TO COPY THE ERROR LISTING INTO THE JOURNAL.
     .ENDJOB    @ TO STOP THE JOB IF THE COMPILATION FAILED.
.FINISH
@
@ CARRY ON HERE IF THE COMPILATION WAS SUCCESSFULL:
LIST (WRITEL,.LP)
@
@ SUPPLY DATA AND RUN THE PROGRAM:
DEFINE (10,*)
RUN (WRITEZ)
THE BAG OF NAILS
THE WYNNSTAY ARMS
THE KING LUD
THE MASTER GUNNER
THE TRAFALGAR
   5 NAMES

STOP  1

++++ Result code =   1
@
@ ANOTHER RESULT TEST, THIS TIME FOR THE RESULT OF A
@ USER PROGRAM:
.IF RESULT>0 .GOTO NONZERO
++++ Evaluating: " RESULT>0 " yields TRUE
@
@ THAT MAY CAUSE THE NEXT COMMANDS TO BE SKIPPED.
     DESTROY (WRITEL)
     DESTROY (WRITEZ)
NONZERO: FILES    @ TO SHOW WHETHER THOSE FILES WERE DELETED.
```

```
* APE          APY         * CLI         * CLISRCES      CLL
  CLY          COMP7         DB1DATA        DCUENTF       EIGHT
  ERF        * EXAMPS        F1          * INSERTM01Y  * JOBBASE
* LINKSS     * MACROUTS      MACY           NOTICE      * OLDFILES
  POLLY        SS          * SS#DIR         SS#OPT        SSD
  SST          SSZ           TCB         * TEXTLETS      WRITEL
  WRITEY       WRITEZ
```

```
@
@ TO TERMINATE THE JOB:
.ENDJOB
```

Example:

This job does not cause any failures, and it behaves just like the previous one even
though the ".WHENEVER" has been omitted. Towards the end of the job, the user program
gives a non-zero result code, but this is not treated as a failure and is not affected by
the presence or absence of a ".WHENEVER" statement.

```
FORTE (*,WRITEZ,WRITEL)
.DATA
        INTEGER B(20), N
100  FORMAT (20A4)
101  FORMAT (1X,20A4)
102  FORMAT (1X,I3,' NAMES')
```

```
              N = 0
      1       READ (10,100,END=2) B
              N = N + 1
              WRITE (6,101) B
              GOTO 1
      2       IF (N.EQ.0) STOP 0
              WRITE (6,102) N
              STOP 1
              END
      .ED
      @
      @ CARRY ON HERE IF THE COMPILATION WAS SUCCESSFUL:
      LIST (WRITEL,.LP)
      @
      @ SUPPLY DATA AND RUN THE PROGRAM:
      DEFINE (10,*)
      .DATA
      THE BAG OF NAILS
      THE WYNNSTAY ARMS
      THE KING LUD
      THE MASTER GUNNER
      THE TRAFALGAR
      .ED
      RUN (WRITEZ)
      @
      @ ANOTHER RESULT TEST, THIS TIME FOR THE RESULT OF A
      @ USER PROGRAM:
      .IF RESULT>0 .GOTO NONZERO
      @
      @ THAT MAY CAUSE THE NEXT COMMANDS TO BE SKIPPED.
          DESTROY (WRITEL)
          DESTROY (WRITEZ)
      NONZERO: FILES   @ TO SHOW WHETHER THOSE FILES WERE DELETED.
      @
      @ TO TERMINATE THE JOB:
      .ENDJOB
```

Resulting line printer output:

```
      FORTE (*,WRITEZ,WRITEL)
       13 Statements compiled
      @
      @
      @ CARRY ON HERE IF THE COMPILATION WAS SUCCESSFUL:
      LIST (WRITEL,.LP)
      @
      @ SUPPLY DATA AND RUN THE PROGRAM:
      DEFINE (10,*)
      RUN (WRITEZ)
      THE BAG OF NAILS
      THE WYNNSTAY ARMS
      THE KING LUD
      THE MASTER GUNNER
      THE TRAFALGAR
        5 NAMES

      STOP  1

      ++++ Result code =    1
      @
      @ ANOTHER RESULT TEST, THIS TIME FOR THE RESULT OF A
      @ USER PROGRAM:
      .IF RESULT>0 .GOTO NONZERO
      ++++ Evaluating: " RESULT>0 " yields TRUE
      @
      @ THAT MAY CAUSE THE NEXT COMMANDS TO BE SKIPPED.
          DESTROY (WRITEL)
          DESTROY (WRITEZ)
      NONZERO: FILES   @ TO SHOW WHETHER THOSE FILES WERE DELETED.
```

```
 * APE          APY         C2JOB        * CLI        * CLISRCES
   CLY          COMP7       DB1DATA        DCUENTF      ERF
 * EXAMPS       F1        * INSERTM01Y   * JOBBASE    * LINKSS
 * MACROUTS     MACY        NOTICE       * OLDFILES     SS
 * SS#DIR       SS#OPT      SSD            SST          SSZ
   TCB        * TEXTLETS    WRITEL         WRITEY       WRITEZ

@
@ TO TERMINATE THE JOB:
.ENDJOB
```

DETACHING JOBS TO OTHER COMPUTERS

Note that the commands DETACH and DETACHJOB, described above, are used only to put jobs
into the EMAS 2900 background queue. Some facilities outwith the Edinburgh Subsystem are
provided for sending job files to other computer systems. Contact your local Advisory
Service for details.

The first part of this chapter describes the method of charging users for their use of EMAS 2900 resources, the procedure to follow to obtain access to the System, and the related command PASSWORD. There is then a description of two information commands METER and USERS.

Finally there is a description of the ancillary commands available in the Subsystem. These are commands which do not readily fit into any of the categories covered by earlier chapters, or, as in the case of OPTION, commands which relate to several of the categories.

GAINING ACCESS TO THE SYSTEM

Before using EMAS 2900 it is necessary to obtain authorisation. This is in two parts:

* Obtaining authority to use the services of the computer installation. Consult your local Computing Service for details of the procedure to follow.

* Obtaining accredited EMAS 2900 user status. This is dealt with by the appropriate EMAS 2900 System Manager.

The end product is a 6-character user name and two four-character passwords. The first password is used for logging on via an interactive terminal, for foreground access to the System; the second is used for card and paper tape input, etc. (see Chapter 2). 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 consist of <u>any</u> four printable characters other than comma. The command takes 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

The use of resources is metered and charged for, and invoices are sent to the appropriate funding body. Consult your local Computing Service for details of the accounting mechanism outwith EMAS 2900 itself. There are two types of charges:

* charges for file space

* charges for computing resources used

File space charging

There are normally three rates for charging for file space:

* Files held in the disc store are charged at two rates: the charge for files which are "cherished" is higher than for those which are not. This is in order to recover the cost of backing up the cherished files and replacing them on disc in the event of a System or hardware failure.

* Files held in the archive store (on magnetic tape) are charged for at a much lower
  rate. This reflects the lower cost of keeping material on magnetic tape rather than on
  disc, and the fact that the archive store is more easily extended than the disc store.

The current charges in some EMAS 2900 services are given in Appendix 5.

Charges for computing

There are four metered 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 EMAS 2900 System.

* Page turns (pt) - this is a count of the number of pages brought into main store or
  written back to the disc for the user process. See Appendix 1 for a description of
  paging.

* Connect time (ct) - this is the time, in seconds, during which an interactive terminal
  is logged on to the System.

* I/O unit records (U) - this is a count of the number of unit records handled; for
  example, lines printed or cards punched.

The charging formulae used at different EMAS 2900 installations make use of these elements
in different ways. Consult Appendix 5 or your local Computing Service for details.

The command METER

        METER

This command is used to obtain usage information and an indication of the amount charged
thus far in the current background or foreground session. The command takes no parameter.
Output is as shown in the following example:

        21/09/78 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

        USERS

This command, which takes no parameter, is used to print out the number of currently
active user processes on the EMAS 2900 System. Note that the figure given does not
include any of the paged System processes (see Chapter 1). It provides an indication of
the loading on the System and the response that can be expected.

ANCILLARY COMMANDS

The following commands do not readily fit into any of the categories covered by earlier
chapters.

Command	Purpose
CPULIMIT	To set time limit for each subsequent command
DELIVER	To specify delivery information, to be printed on output files
DOCUMENTS	To print information about files awaiting output and jobs awaiting execution
MESSAGES	To suppress or enable operator message output on interactive terminal
OBEY	To execute a sequence of commands
OPTION	To set a number of optional characteristics of the Subsystem
QUIT	To terminate foreground session
SETMODE	To select Terminal Control Processor (TCP) options
STOP	To terminate foreground session
SUGGESTION	To send suggestion to the System Manager
TELL	To send message to another user

Table 17.1: Ancillary Subsystem Commands

The command CPULIMIT

CPULIMIT(time/?)

This command is used to set the amount of central processor unit (CPU) time allowed for each subsequent command. It takes two parameters, a number of minutes and a number of seconds; at least one of these must be specified. For example:

Command	Time set
CPULIMIT (3)	3 minutes
CPULIMIT (,10)	10 seconds
CPULIMIT (1,30)	1 minute and 30 seconds

The form CPULIMIT(?) can also be used. It causes the CPU limit currently in force to be printed out.

Notes

* The default and maximum CPU limit values are set by the System Manager.

* The CPULIMIT command can be useful for testing programs that contain faults which may result in infinite loops. By using CPULIMIT with a low value - 5 seconds or so - the elapsed time taken to reach the "Time exceeded" failure is considerably reduced.

* The current CPU time limit affects separately each command called by OBEY (see below): it is not applied as though OBEY were a single command.

The command DELIVER

DELIVER(address/?)

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 consist of suitable text with a maximum length of 31 characters. No spaces should be used, as they are discarded; the underline character is a suitable substitute. For example:

DELIVER(59_GEORGE_SQUARE)

DELIVER(CHEMISTRY_K.B.)

Note that the user's surname is automatically printed - there is no need to include it in the DELIVER string.

The form DELIVER(?) can be used to determine the current delivery information. It is printed in reply on the primary output device (usually the user's interactive terminal).

Note that the command takes effect immediately and remains in effect until another call of the DELIVER command.

The command DOCUMENTS

DOCUMENTS(queue/ident, out)

All documents input to EMAS 2900 and output from EMAS 2900 are handled by the paged System process SPOOLR, which maintains queues for each category of input and output document. The DOCUMENTS command gives the user information on these queued files and jobs, and in addition gives information on documents which have recently been removed from a queue and processed.

The command may take various parameters:

DOCUMENTS	With no parameter, information is given on all documents currently queued for this user.
DOCUMENTS(queue)	Gives the same information but only in respect to the specified queue; e.g. .LP for local line printer files or .BATCH for jobs.
DOCUMENTS(ident)	This describes a single document in greater detail. Note that in this case the document specified need not currently be in any queue. Information on the document remains available for a short period even after it has been taken from the queue and processed. The value of "ident" is included in the information produced by the other forms of the DOCUMENTS command, and by the DETACH command (Chapter 16) when a job is submitted for background execution.

The following information is printed for each queued file:

name	The name of the document in the form known to the user. If the document has been HELD by the operator for any reason, the name is preceded by an asterisk.
ident	A number in the range 1-1000 which uniquely identifies the document. The DOCUMENTS command will accept "ident" as a parameter and produce additional information on the file. Note also that "ident" is the parameter required by command DELETEDOC (which deletes a queued document.)
date time	The date and time when the document was received.
priority	One of the values Vlow, Low, Std, High, Vhigh.
size	The size (in Kbyte) for a file, or in seconds for a batch job.
ahead	An indication of the position of the document in its queue. For a file it gives the total Kbyte scheduled to be processed before it, and for a job it gives the total seconds of CPU time requested by preceding jobs. The value zero indicates that the document is at the head of the queue.

An optional second parameter (default .OUT) allows output to be directed to a file or device. For example:

    DOCUMENTS(.BATCH,TEMP)

The command MESSAGES

        MESSAGES(ON/OFF)

This command allows the user to inhibit operator and other messages which might otherwise interfere with the layout intended for some terminal output; e.g. document or graphics output. The command is used as follows:

        MESSAGES(OFF)   inhibits messages

        MESSAGES(ON)   re-enables messages

While the output of messages is inhibited, such messages are queued, and printed when messages are re-enabled. The types of message affected by this command include closedown warnings issued by operators, messages from other users (see command TELL, below), and file restoration messages issued by the paged System process VOLUMS.

The command OBEY

        OBEY(file, out)

This command is used to execute a sequence of commands (processed by the command interpreter). The required commands and any data they would normally read from the interactive terminal should be put in a file, using a text editor or some other means. The format should be identical to that which would be used when typing commands on the interactive terminal. OBEY 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 primary output device (in foreground mode the interactive terminal). For example:

    OBEY(CF26)
    OBEY(NRJOB,.LP15)

Calls of OBEY are not permitted among the commands in a file to be OBEYed; i.e. OBEY cannot be nested. It is, however, permissible to use OBEY in background mode (Chapter 16). The primary output device is the line printer appropriate to the source of the background job (the local line printer, .LP, when the job was initiated from foreground).

To have a file of job control language statements processed by the job interpreter, the equivalent command OBEYJOB should be used. This is described in detail in Chapter 16.

The extent of the echoing generated by OBEY can be controlled by use of the options NOECHO, PARTECHO and FULLECHO (see OPTION, below).

The command OPTION

        OPTION(optionlist/?)

This Guide describes the characteristics of the Edinburgh Subsystem. To a certain extent a user can modify the characteristics of the Subsystem, by use of the OPTION command. The changes specified take effect immediately (except in very rare circumstances; see Chapter 15 for details). They remain in force until changed again by a subsequent use of OPTION - they are not lost when the user logs off.

OPTION takes one or more keyword parameters, which are listed below with a description of their effect. The standard settings are described thereafter.

The parameter "?" can be given instead of an option list: OPTION(?) causes details of the currently effective options to be printed.

Option	Effect

ACTIVEDIR=directory

The nominated directory file, which must be the user's own, is to be modified by the commands INSERT, REMOVE and ALIAS, and examined before the SEARCHDIR directories (if any) when an entry point not currently loaded is sought.

SEARCHDIR=directory

The nominated directory file is to be inserted at the head of the list of directory files searched during the program loading process. If the nominated directory file is already in the list, it is moved to the head of the list.

REMOVEDIR=directory

The nominated directory file is to be removed from the SEARCHDIR list (see above).

The above three parameters are discussed further in Chapter 11.

BRACKETS

Parameters to Subsystem commands are to be included in brackets after the command name. All the examples in this Guide assume that this option is in force.

NOBRACKETS

Parameters to Subsystem commands are not to be enclosed in brackets. Instead the name of the command is to be separated from its parameters by one or more spaces, and the command name itself will not contain any embedded spaces. Spaces within parameters are ignored, as before.

For example, the following two sets of commands are equivalent:

```
        BRACKETS                   NOBRACKETS

Command:LIST(A,.LP)         Command:LIST A,.LP
Command:CPU LIMIT(1, 20)    Command:CPULIMIT 1, 20
Command:METER               Command:METER
```

NORECALL

The interactive terminal input/output dialogue is not to be recorded.

TEMPRECALL

The interactive terminal input/output dialogue is to be recorded for possible use by RECALL or RECAP (see Chapter 8) later in the current session.

PERMRECALL

The interactive terminal input/output dialogue is to be recorded for possible use by RECALL or RECAP (see Chapter 8), and is to be preserved between interactive sessions.

Only one of these options can be effective at any time. If PERMRECALL or TEMPRECALL is effective, then the interactive terminal dialogue is recorded in a file named SS#JOURNAL or T#JOURNAL respectively (created by the Subsystem). When this file is full the earliest dialogue recorded in it is overwritten by subsequent dialogue. Thus it is in effect a circular buffer.

ITWIDTH=n

The interactive terminal line width is to be assumed to be n characters by relevant Subsystem commands (e.g. FILES, ANALYSE). Note that this is not related to the TCP setmode 'W' option (see Chapter 3 and below).

NOBLANKLINES

Any blank lines to be output on the interactive terminal, however generated, are to be suppressed by the Subsystem.

BLANKLINES

Blank lines intended for output on the interactive terminal are not to be suppressed by the Subsystem.

NOECHO   Nothing is to be echoed during the execution of a file of commands
                        by OBEY (see above).

PARTECHO   Commands are to be echoed during execution of an OBEY.

FULLECHO   Commands and data are to be echoed during execution of an OBEY.

FSTARTFILE=file   The nominated file is to be OBEYed (see OBEY, above) automatically
                        whenever the user process is started in foreground mode (i.e. when
                        the user logs on). Note that it is not possible to nominate an
                        output file for the OBEY: any output will be printed on the user's
                        interactive terminal.

NOFSTARTFILE   No file is to be automatically OBEYed when the user process is
                        started in foreground mode.

BSTARTFILE=file   The nominated file is to be OBEYed automatically when the user
                        process is started in background (i.e. non-interactive) mode. Note
                        that it is not possible to nominate an output file for the OBEY:
                        any output will be sent to the line printer appropriate to the
                        source of the background job (the local line printer, .LP, if the
                        job was initiated from foreground).

NOBSTARTFILE   No file is to be OBEYed automatically when the user process is
                        started in background mode.

Only one of FSTARTFILE, NOFSTARTFILE, and only one of BSTARTFILE, NOBSTARTFILE can be
effective at any time.

ARRAYDIAG=n   Whenever diagnostics are to be generated during or following the
                        execution of a program, the values of the first n elements of each
                        array involved in the diagnostics are to be output.

CFAULTS=out   The specified device or file will be used by default as the fourth
                        parameter for compiler-calling commands (see Chapter 11); that is,
                        to contain a list of compile time fault messages. The setting of
                        CFAULTS can be overridden by an explicit specification of the
                        fourth parameter.

INITPARMS   The current settings of the PARM options (Chapter 11) are recorded,
                        and automatically selected at the start of every session.
                        N.B. This does not affect the operation of the PARM command, which
                        still resets the options to the site-defined defaults before
                        setting selected values.

The remaining parameters all relate to sizes of files. They each take a numerical value
which represents a number of Kilobytes. Most users will not need to change the initial
settings of these parameters.

ITINSIZE=n   Change the size of the interactive terminal input buffer to
                        n Kbyte.

ITOUTSIZE=n   Change the size of the interactive terminal output buffer to
                        n Kbyte.

INITSTACKSIZE=n   The size of the area at the bottom of the "user stack" (see below)
                        used by the FORTRAN compiler to store all the scalar variables in a
                        program. See Chapter 11, and particularly the section "Dynamic
                        loading", for details.

INITWORKSIZE=n   The initial size of any workfiles used by Subsystem components
                        (such as compilers and text editors) is to be n Kbyte.

USERSTACKSIZE=n   The stack (a type of data storage area), used by ALGOL, FORTRAN and
                        IMP programs for storing run-time housekeeping information and
                        local scalar variables, is to be n Kbyte in size.

AUXSTACKSIZE=n   The stack used by ALGOL and IMP programs for storing local arrays
                        at run-time is to be n Kbyte in size.

The "SIZE" parameters are discussed further in Chapters 11 and 15.

———

Note that the options available are liable to change. Up-to-date information can be
obtained by use of the HELP command (Chapter 4). In addition, the command OPTION(?) gives
a list of the current settings, and therefore of the available parameters.

Initial settings

The Subsystem parameters which can be modified by use of OPTION have the following
settings for a new user process. Note that the only way of reverting to them once any of
the values has been changed is by using OPTION to change the modified parameters back
again.

        ACTIVEDIR=SS#DIR

        Directories to be searched (SEARCHDIR/REMOVEDIR)
                        This depends on the installation; use the OPTION(?) form of the
                        command to determine which directories are currently searched.

        BRACKETS

        NORECALL

        NOFSTARTFILE

        NOBSTARTFILE

        PARTECHO

        ARRAYDIAG=0

        INITSTACKSIZE ⎫
                      ⎪
        INITWORKSIZE ⎬ The initial values for these parameters are liable to
                      ⎪ change. Use the OPTION(?) form of the command to
        USERSTACKSIZE ⎬ determine their current settings.
                      ⎪
        AUXSTACKSIZE ⎭

Examples

        OPTION(ACTIVEDIR=DIRA, SEARCHDIR=ERCC23.D237, ITWIDTH=85, PERMRECALL)

        OPTION(?)

        OPTION(NOBRACKETS,BSTARTFILE=BACKSTART)

The command QUIT

        QUIT

This command, which takes no parameters, is identical to STOP (see below).

The command SETMODE

        SETMODE(commandlist)

As described in Chapter 3, the user can change the operational characteristics of the
Terminal Control Processor (TCP) through which his interactive terminal is connected to
the EMAS 2900 mainframe. This is done by pressing the SOH (or CTRL+A) key on his
terminal, thus putting the TCP into "set" mode. Thereafter particular options can be
selected. The EMAS 2900 mainframe is not involved in this process in any way - the user
is communicating only with the TCP.

However, it is also possible to set the TCP's characteristics from the Subsystem. This is
done by use of the command SETMODE. The parameter to SETMODE consists of a single setmode
command or a list of setmode commands separated by commas. For example:

        SETMODE(LOWER)
        SETMODE(W=120,H=66)
        SETMODE(VIDEO=ON,TABS=5:10:15:*,GRAPH,DELETE=63)
        SETMODE

A number of points arise from these examples:

* Either the full setmode command name or the name abbreviated to its initial letter may
  be specified.

* Where a setmode command parameter is required it follows an equals sign (=).

* Where a setmode command parameter is to be a character, the ISO code numeric value of
  the character must be given; thus, DELETE=63 rather then DELETE=?. This enables
  non-printing characters to be specified. See also Note 1, below.

* SETMODE with no parameter causes all TCP functions to be reset to their default values.

* Each TCP command in the list is treated as a distinct call on SETMODE, so that if one
  fails an attempt is nevertheless made to execute the remainder.

Setmode commands

The TCP setmode commands available are those described in Chapter 3, with one addition:

    E   Echo   Disable echoing of input characters. (This is automatically used
                        when the password is entered during the log-on sequence.)

The following table summarises the setmode commands available, with their parameters and
default values.

Command	Parameter	Default
C Cancel	Numeric code (Note 1)	24 (i.e. CAN)
D Delete	Numeric code (Note 1)	127 (i.e. DEL)
E Echo	ON or OFF	ON
G Graph	-	OFF (Note 2)
H Height	Number (min=5, max=255)	0
I INT	Numeric code (Note 1)	27 (i.e. ESC)

Table 17.2: Setmode Commands Available via Subsystem Command SETMODE
(continued on next page)

Command	Parameter	Default
L Lower	-	OFF (Note 2)
P Pads	Number (min=0, max=100)	0
R Return	Numeric code (Note 1)	13 (i.e. CR)
T Tabs	List of numbers, separated by colons (Note 3)	6:9:12:15:18:40:80
U Upper	-	ON (Note 2)
V Video	ON or OFF	OFF (Note 4)
W Width	Number (min=15, max=160)	72

Table 17.2: Setmode Commands Available via Subsystem Command SETMODE

Notes

1. These setmode commands require a single character parameter. This can only be specified as a numeric value, in the range 2-127. This value is the numeric code associated with the required character in the ISO character code (see Appendix 2).

2. Selecting Upper mode automatically switches off Lower and Graph modes.

3. The Tabs command accepts up to seven parameters, separated by colons (":") containing values in the range 1-160 arranged in ascending order. An "*" instead of a number terminates the list and causes the remaining values to be taken from the list of default values (see above), possibly extended with the value 160, repeated if necessary; the use of the value 160 depends upon the last tab parameter specified by the user (see the first example below).

   Examples:

   TABS=10:15:*   is expanded to   10:15:18:40:80:160:160

   TABS=4:8:10:*   is expanded to   4:8:10:12:15:18:40

   TABS=*   is expanded to   6:9:12:15:18:40:80

4. The Video command is effectively a switch, so that selecting Video when the TCP is already in Video mode has the effect of terminating Video mode.

Note that the SETMODE command takes effect only when the process is running in foreground mode - in background mode it is ignored.

The command STOP

          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 above).

* Destroys all temporary files created during this session (including the default compiler listing file T#LIST if it exists).

* Disconnects the interactive terminal - making it available for another user.

* Stops the user's virtual process - freeing a slot for another user to log on.

The command SUGGESTION

SUGGESTION

This command, which takes no parameter, is provided to make it easy for users to send
suggestions for changes or improvements to EMAS 2900 to the System Manager. Its use is
intended primarily for minor items 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 the suggestion being implemented.

The method of use is to type the command with no parameter. The replies to the prompts
"Surname:", "Address:" and "Text:" should be the personal (not EMAS 2900) name of the
user, the address for a reply, and the text of the suggestion terminated with an asterisk
on a line by itself. The following example shows this:

```
Command:SUGGESTION
Surname:DR G. JONES
Address:119 GEORGE SQUARE
Text: ... text of suggestion ...
Text: ... text of suggestion ...
Text:*
```

As many lines of text as desired may be given.

The command TELL

TELL(username, messagefile)

This command enables a user to send a message to another user of the System, immediately
if the other user is logged on, or when he next logs on.

The first parameter is the username of the recipient. Note that this can be the same as
the username of the sender (useful when, for example, TELL is invoked within a background
job).

The second parameter is optional. If given, it must be the name of a character file
containing the text of the message to be sent. If the second parameter is omitted, TELL
prompts for the message to be input on the primary input device (usually the sender's
interactive terminal). The message is terminated by an asterisk on a line by itself. As
many lines of text as desired may be given.

For example, suppose that user ABCD01 inputs the following:

```
Command:TELL(ERCC27)
Text:My file INFO1A is now on OFFER to you.
Text:*
```

If user ERCC27 is on-line, or when he next logs on, he will have the following printed on
his terminal:

```
**ABCD01 dd/mm/yy hh.mm: My file INFO1A is now on OFFER to you.
```

The date and time indicate when user ABCD01 sent the message.

Note that the receipt of such messages can be deferred by use of the MESSAGES command,
described above.

It is suggested that users confine this facility to sending short messages. If they have
a long message then it is probably more convenient to put it in a file which can be
OFFERed or PERMITted to the user concerned. The TELL facility can then be used to
indicate that such a file exists. For example:

```
Command:TELL(ERCC97)
Text:File ERCC98.LETTERS is on offer to you.
Text:*
```

This appendix contains a brief introduction to paging mechanisms and virtual memory. For two reasons it is felt necessary to explain these features of EMAS 2900: first because they form a vital part of the System, and secondly because an understanding of them, although not necessary, can sometimes enable a user to make more efficient use of the System.

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 A1.1). 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 500 Kbyte available for user programs and all user programs currently awaiting execution require 200 Kbyte, then 100 Kbyte 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 A1.2 program A terminates and the only available programs to run require 250 Kbyte, one will be loaded (program C) as in Figure A1.3, leaving a gap of 50 Kbyte at the top of the store. If program B now terminates there will be a total of 250 Kbyte of store free, but a single program requiring 250 Kbyte will not be able to run because the areas of store which are free are not contiguous.

USER PROGRAM A (500 Kb)
operating system

USER PROGRAM A 300 Kb
USER PROGRAM B 200 Kb
operating system

50 Kb unused
USER PROGRAM C 250 Kb
200 Kb unused
operating system
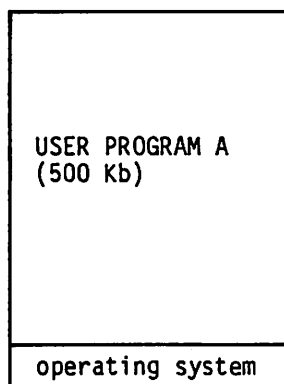
Figure A1.1

All available store
allocated to one
user program
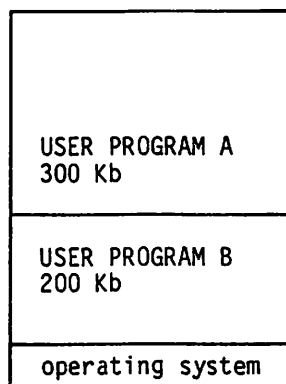
Figure A1.2

Available store
allocated to two
user programs
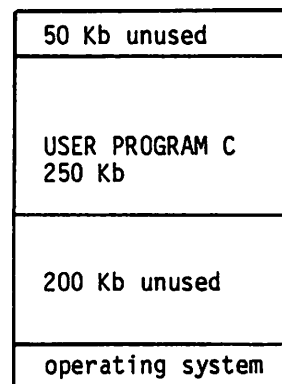
Figure A1.3

A second 250 Kb 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 three or four 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 main store into discrete partitions of fixed size, which is feasible in a multi-programming batch service, is not practicable 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 magnetic 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 2900 uses paging. This involves some special hardware and part of the code in the Global Controller (see Chapter 1). The main store of a 2900 computer can be divided into pages of 4096 bytes each (strictly, of 1024 bytes each, although EMAS 2900 works exclusively with a 4096-byte unit of storage known as an "extended page" or "epage"). 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 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 0 in the main store (see Figure A1.4).

The name "virtual store" or "virtual memory" is given to the effective store addressed by the program.
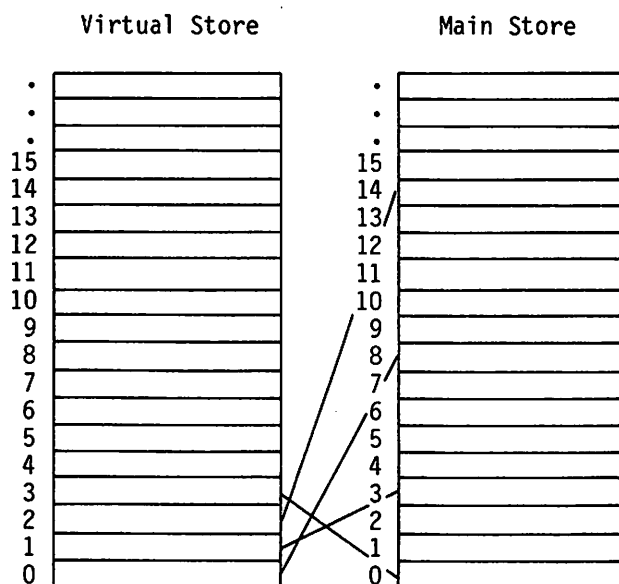
Figure A1.4 Example of paging

One of the problems not considered so far is the great difference between the total storage requirement of all active processes and the size of the main store. For example, for a 2900 mainframe with approximately 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 disc storage, which holds fairly permanent copies of material, and also - in some cases - drum storage, which holds transient copies of recently accessed pages. The drum storage is not essential, but makes the paging faster.

Part of a
user's virtual
memory

Backing store

User access
to file A

A

A

User access
to file B

B

B

Mapping
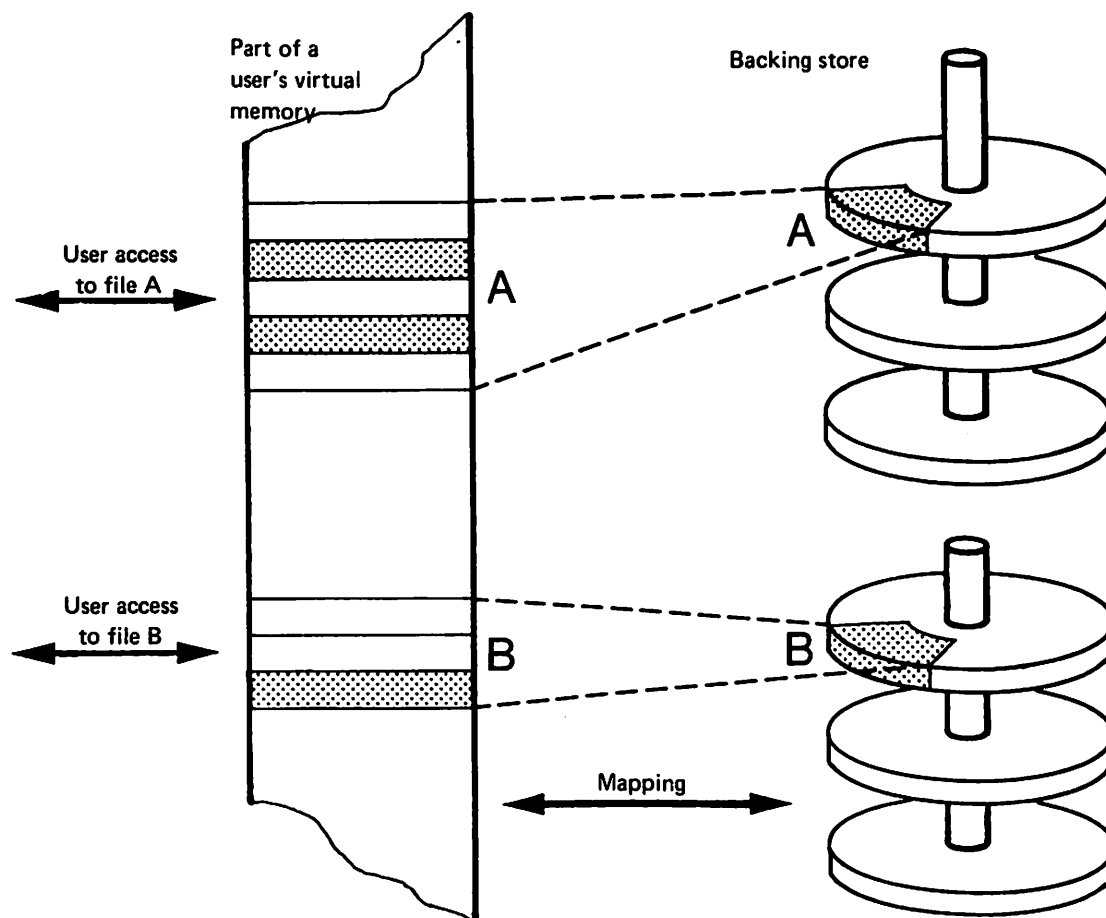
Files A and B are connected into the user's virtual memory. The user accesses them as though they were part of contiguous main store — it is one of the functions of EMAS 2900 to sustain this fiction.

Each file is divided into fixed-size pages. A page of a file is only brought into "real" main store if it is referenced by a program or if code within that page is to be executed. The pages shaded in the diagram of the virtual memory represent those actually in main store; they will be copied to backing store, if they have been altered, when the main store space they occupy is required for other pages (probably by other users).

A file can be shared by two or more users, i.e. it can be connected simultaneously in two or more virtual memories. If a page of such a shared file is required simultaneously by two or more users, nonetheless only *one* copy of that page is held in main store.

Note that the diagram above is considerably simplified, and does not, for example, show the two levels of backing store which may be used.

Figure A1.5: File connection in EMAS 2900

Page faulting and virtual memory

An important characteristic of the paging technique is its 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 by the System from main store to 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.

The effect of paging on user programs

Paging enables the System to allocate storage efficiently between the users. In so doing it provides each user with a virtual memory, i.e. an addressible space, far larger than the real main store: a user of EMAS 2900 can address an apparently continuous area of approximately 40 Mbyte. (In fact a user's virtual memory is 48 Mbyte in size, but the first 8 Mbyte are used by the Local Controller and Director (see Chapter 1), and are not accessible to the user.)

One of the attributes of paging is that its operation is transparent to the user, who can proceed on the assumption that his "virtual" memory is real in every respect, and use the benefits of such an enormous main store. If one had such a main store, one could, for example, run far larger programs than a normal size of main store would allow, without having to resort to overlaying. More generally, one could dispense with transferring information between backing store and main store in small sections, with all the complications of buffering entailed. Instead one could simply move all of the relevant files on backing store into main store and access them directly thereafter.

In EMAS 2900 this is precisely what happens: a file is accessed by first being "connected" into a virtual memory, at a suitable place, i.e. where there is an unused area big enough to take it. Thereafter reference to an address within the appropriate part of the virtual memory accesses the file directly. The System does not in fact move the complete file into some special area when connecting it; it merely notes the disc location of the file and associates that location with a range of addresses in the virtual memory in question. Connection is described further in Chapter 1, and in Reference 1. See also Figure A1.5.

It is important to understand that there is no distinction, to the System, between data files and program files and even work areas such as the stack, used by IMP and ALGOL programs to hold variables and arrays. Each connected file has an address in the virtual memory and consists of one or more pages, of which there is always a copy on a disc; there may also be a copy on a drum, and when the file is in use 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 7, file handling routines have been made available which simulate those normally provided, in order to simplify the transfer of programs to and from EMAS 2900. In addition, however, there are routines available which 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 9).

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 variation 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

Further details of the paging hardware and software are contained in References 1 and 2.

EMAS 2900 uses internally the ISO character code. Within EMAS 2900, code values in the range 0-127 are associated with graphical or control characters (see Table A2.1). The graphical representations of some codes vary from one output device to another; common alternatives are shown in the table.

The code values 128-255 are not associated with graphical or control characters within EMAS 2900. However, codes which are in this range are not modified by EMAS 2900 in any way and can thus be used when sending character information to remote terminals or processors by means of a communications network.

Consult your local Advisory Service for information about the character sets supported by local communication networks, and the facilities available for communicating between an EMAS 2900 mainframe and another mainframe on such a network.

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	$	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*	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

*referred to within EMAS 2900 as NL

Table A2.1: EMAS 2900 Internal Character Code

Common error messages produced by the Subsystem should be self-explanatory. Some of the general ones are described more fully below, and those specific to an individual command are described in the part of this Guide relating to the command.

GENERAL ERROR MESSAGES RELATING TO FILES

These messages can be produced following faults in many commands. The character "&" in a message is replaced by the relevant name at the time of the failure.

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. For example:

1) When an attempt is made to compile into an object file belonging to this user which is also permitted to another user and is currently being executed by him.

2) When an attempt is made to read from a file which belongs to another user (and is permitted to this user in READ mode) at the same time as the owner of the file is running a program which is writing to the file.

———

File & connected in another VM

The file is currently connected in another virtual memory. While this is the case it is not possible to DESTROY, RENAME or OFFER it.

———

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 is insufficient room in the user's file index to hold information about the file being created. Note that the creation may be explicit, e.g. as in COPY, or may be as a result of an attempt by the Subsystem to create a file, e.g. as in FILES(,,.LP) or LIST(FILE27,.LP).

———

File system full

    This message occurs when the disc containing this user's files is full.

Inconsistent use of file &

    This message will occur when an attempt is made to operate on a file in a way which is inconsistent with its current use. For example, the command

        IMP(TEST,TEST)

    will cause this message to be produced, since an attempt is being made to compile into a file from which source is being read.

Invalid filename &

    A full filename is of the form

        username.filename_membername

    The username, if present, must contain 6 characters. The filename can contain up to 11 characters, as can the membername, if used. When creating new files or members, all characters must be upper case letters or numerals and the first character must be an upper case letter.

Invalid username &

    This message will occur if an attempt is made to access a file belonging to a user whose files are not available. This may be because no such username exists, or because the disc containing the user's files is not currently on-line.

Requested access to file & not permitted

    If a file is permitted to this user in READ mode and an attempt is made to connect the file for writing, e.g. EDIT(filename), this message will be produced.

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 relating to the error messages produced by the Subsystem.

This appendix gives brief definitions of a number of terms and abbreviations used in the Guide and elsewhere in relation to EMAS 2900. 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.

Access Permission	An attribute of a file, which can have several associated access permissions. Each one specifies a user, a group of users, or all users, and a corresponding permitted level of access. Access permissions determine which connect modes are possible.
Accredit	To register a user on the EMAS 2900 System. This is done by the System Manager, and entails giving the user a File Index and access to it by means of a foreground password and a background password.
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.
Address	An integer value which uniquely specifies a byte location in a store (memory). A "physical address" refers to a location in the computer's main store; a "virtual address" refers to a location in a virtual memory. Addresses of data items which are larger than a byte specify the leftmost byte of the data items.
Archive Index	Each user has an index of those of his files which are held in the archive store (cf. File Index, below).
Archive Store	A long-term store for users' files, held on magnetic tape (see Chapter 1).
Background Mode	A mode of use of EMAS 2900 where access is not from an interactive terminal. Background jobs may be put in a queue by the user during a foreground session for execution later. They can also be submitted on cards or paper tape, or from other computer systems.
Backup Store	A magnetic-tape based store holding copies of users' disc files, the purpose being to enable their restoration should they be corrupted or destroyed through hardware or software malfunction. This is distinct from archiving, in which disc files are copied to magnetic tape and the disc copy then deleted.
Byte	The smallest addressible unit in the main store (or virtual store) of ICL 2900 series computers. It contains 8 binary bits. It is used in IMP for variables of type %BYTEINTEGER and in FORTRAN for variables of type LOGICAL*1.
Connection	The only method provided by EMAS 2900 for gaining access to files. It is a simple concept central to the provision to each user of a virtual memory. See Appendix 1.
Console	A keyboard device connected to a computer system and permitting dialogue with it. A "user console", or "interactive terminal", is normally an electric typewriter or video device.
Core Store	An anachronistic synonym for Main Store, which used to be constructed from ferrite cores threaded on wires.
Digital Equipment Corporation (DEC)	A computer manufacturer. In many EMAS 2900 installations the Front End Processor is a DEC computer.
DIRECT	A paged System process which controls the logging on of users (checking passwords, initiating user processes, etc.). It also carries out consistency checks on the File Indexes at IPL time.

Director   A collection of routines, forming part of the EMAS 2900 System. Each
                  user process has its own Director. If EMAS 2900 is seen as an
                  onion-like structure, Director is the level directly below the
                  Subsystem. All the services provided for a user by EMAS 2900 are
                  requested by calls on Director routines, which thus form a procedural
                  interface between a user process and the rest of EMAS. Normally,
                  however, there is another layer between this interface and the user,
                  viz. the Subsystem.
                  Director is paged; it is contained in the associated user's virtual
                  memory.

Disk Drive   An I/O device with a rotating spindle onto which a disc pack can be
                  mounted, together with read/write heads for each recording surface, a
                  mechanism for positioning the heads, and electronics for controlling
                  the reading and writing of data. In some designs the read/write heads
                  are part of the disc pack, not the disc drive.

Disc Pack   A magnetic storage device consisting of a number of flat circular
                  plates, each coated on both surfaces with some magnetizable material.
                  The plates are mounted on a central column. On each surface there is a
                  series of concentric tracks and when the disc pack is mounted on a disc
                  drive and rotated at high speed, a read/write head (one per surface)
                  can be moved radially to a required track for accessing or changing the
                  information stored there.

Disc File   See File (below).

Disc Store   See Store (below).

Drum   A rotating magnetic storage device, similar in principle to disc
                  storage. The primary difference is that there is one read/write head
                  associated with each track so that the heads do not move. Drums are
                  sometimes called "fixed-head discs".

Edinburgh Subsystem The name given to the Subsystem provided with EMAS 2900.

Epage   (Stands for "extended page".) See Page.

File   A collection of information held within some storage device in the
                  computer system. EMAS 2900 operates - from the user's point of view at
                  least - in terms of files, and extensive facilities are provided by
                  Director for using them, keeping them, sharing them, etc. Director
                  does not impose any conditions or conventions regarding the
                  interpretation or use of the contents of files. The Edinburgh
                  Subsystem provides facilities for working with particular types of file
                  (see Chapter 4), although other Subsystems need not adhere to the
                  conventions adopted by it. A user's disc files are all kept on a
                  single disc pack.

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.

File Index   Each accredited user of EMAS 2900 has a File Index which is maintained
                  by his Director. It keeps information about all the disc files
                  belonging to that user, including their sizes, access permissions to
                  other users, etc. The File Index is kept on the same disc pack as all
                  that user's files.
                  It also contains some information not related to the user's files. See
                  System File Information, below.

Filename   (Sometimes written as two words.) The name given to a file by a user
                  when it is created or renamed. The name must be unique within that
                  user's File Index, but other users may have files of the same name; see
                  also Full Filename. A filename can contain up to 11 upper case letters
                  or digits, the first character being a letter.

File System   Each user of the System has a File Index, which contains information
                  about all his disc files. Each user's File Index and associated files
                  are held on a single disc pack. A file system consists of the files
                  and File Indexes held on a single disc pack, together with other
                  information stored there to manage the space allocation and to ensure
                  the self-consistency and integrity of the disc pack. The file system

is the collection of all the disc pack file systems. Note that the term "File System" refers only to on-line files, i.e. those on disc.

Foreground Mode   A mode of use of EMAS 2900 where access is from an interactive terminal.

Front End Processor (Sometimes called "FEP") A computer which is directly connected to the computer running EMAS 2900. Its job is to manage communications with remote processors and terminals. A large EMAS 2900 mainframe may have several Front End Processors.

Full Filename   A filename together with the owner's username. Under the Edinburgh Subsystem the two parts of the full filename are joined by a '.' to form a single string: thus "username.filename". The full filename is unique to that file and must be specified by users other than the owner when referring to the file.

Hardware   The physical components of a computer, such as the central processor unit, the magnetic tape units, etc.

Head Crash   A hardware fault usually affecting a disc pack and corrupting or destroying some users' disc files. It occurs when a recording head comes into contact with the surface of a disc, causing physical damage to the recording medium.

Hexadecimal   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, since the contents of each byte can be represented by two hexadecimal digits.

ICL   See International Computers Ltd.

IMP   A block-structured problem-oriented high level programming language of the ALGOL type. It was developed in Edinburgh from Manchester University's Atlas Autocode, primarily for implementing system software - hence its name. Apart from a very small amount of machine code, the EMAS 2900 System software is written entirely in IMP; this includes the Edinburgh Subsystem and the compilers currently available (ALGOL(E), FORTAN(G), IMP). See Reference 7.

Interactive   Descriptive of the type of computer use which permits a user to respond to the output as it is being generated, and thus to stop or change the course of a computation. For such interaction a user console is required. With the Edinburgh Subsystem of EMAS 2900 three levels of interaction are permitted to a user: 1) after each command has been completed, successfully or unsuccessfully, the user is asked for his next command; 2) during the execution of a program, the user may be asked for input data, depending on how the program was written; 3) the user can interrupt any program running in his process and redirect or abort it. The standard alternative to interactive (or foreground) operation is batch (or background) operation.

Interactive Terminal   The general name for a terminal used for accessing EMAS 2900 in foreground mode - i.e. interactively. It can be a Teletype, a VDU or some other form of terminal. Also called a console.

International Computers Ltd (ICL)   The manufacturers of the 2900 series of computers, on which EMAS 2900 runs.

I/O   (Stands for "Input/Output".) Information travelling to or from a computer system.

I/O Device   A piece of hardware, attached to a computer, which can read, store or write information in some form.

IPL   (Stands for "Initial Program Loading".) Strictly a hardware function involving reading in a minimal control program to an otherwise empty machine. In the case of EMAS 2900 the term covers the whole process of initialising the computer for an EMAS 2900 session. This entails discovering the configuration, initialising store and the peripheral controllers, carrying out a consistency check of the File Indexes, and starting the continuously running paged System processes.

Keyword Parameter   A command parameter which is assigned a value, when the command is invoked, by use of a keyword. For example, the Edinburgh Subsystem command SETMODE (Chapter 17) uses keyword parameters.
Most Edinburgh Subsystem commands use positional parameters, where the correspondence between parameters and assigned values is by means of a pre-determined order of specification.

Main Store   The part of the computer hardware which can store instructions or data to be executed or read or written. With respect to the EMAS 2900 System, main store is divided into "page frames", each able to hold a page. It is allocated thus to user processes, although EMAS 2900 manages main store and backing store in such a way that they appear to form a continuous main store much larger than the pages of main store actually allocated. See also Store.

MANAGR   A paged System process administered by the System Manager, and used for accrediting users, changing control parameters for users, etc.

Memory   A synonym for Store.

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.

On-line   1) Of peripheral equipment: logically connected to the EMAS 2900 System, the alternative being "off-line". 2) Of a user: having successfully carried out the standard log on procedure via an interactive terminal, so that the associated user process has been started up and interactive working can be undertaken.

Operating System   A suite of programs that controls usage of the computer; in EMAS 2900 this includes the Supervisor, Director, Subsystem and the paged System Processes.

Page   The basic unit of store in 2900 Series hardware, of size 1 Kilobyte. EMAS 2900 works with "extended pages" (epages), which are defined in units of pages. The size of an epage is currently 4 pages, although this could in principle be altered. Main store is divided into epages, and information is transferred between main store and backing store in units of epages. From the point of view of a user or a Subsystem, however, information is handled in terms of files, the paging operating transparently. See also Units of Storage, below.

Paged System Process   Similar to a normal user process but belonging to the System. The paged System processes are used to carry out System tasks such as spooling files, managing the archive store, running Engineer test programs, etc. They differ from user processes in that: 1) they may have access to facilities not given to users; 2) in some cases they are not run from an interactive terminal; 3) in some cases they may not make use of the Edinburgh Subsystem.

Page Turn   A term used to describe the transfer of a page to or from main store. Page turns are recorded, and the total number of page turns caused by a process is used as accounting information.

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 discs, it allows many processes with varying storage requirements to use a limited main store as if each had a large allocation of continuous store. See Appendix 1.

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 11.

Process   An autonomous activity which can compete for the resources of the System. It is the basic unit of scheduling. The main function of EMAS 2900 is to provide a suitable environment for user processes. Each user process or paged System process is autonomous in the sense that it apparently has a virtual processor and virtual memory to itself. In a single-processor computer only one process will actually be executing at any instant, but the resources are shared in such a way that all

	active processes appear to be executing. Two types of process are distinguished: a virtual process, which is paged, and a Supervisor process, which is resident in main store. See Chapter 1.
RJE	(Stands for "Remote Job Entry".) The technique of using a computer, non-interactively, from a remote site. With respect to EMAS 2900, RJE terminals (usually, but not necessarily, comprising a card reader and a line printer) are connected to the computer running EMAS 2900 via its Front End Processor, which handles all communications traffic.
Software	A general name for programs as distinct from hardware. Also used to describe components of the operating system, as against user and applications programs.
SPOOLR	A paged System process which controls the transfer of files between the computer running EMAS 2900 and 1) local slow peripherals, or 2) remote devices (via the Front End Processor). All input files destined for, and output files generated by, user processes are transferred via SPOOLR, which also handles and schedules batch jobs, however submitted.
Store	Four types of physical storage media are distinguished in EMAS 2900: archive store (magnetic tape), disc store, drum store (optional), and main store. A file is either archived (held on magnetic tape) or it is a disc file.
Subsystem	The facilities provided by EMAS 2900 are made available to users via the Director procedural interface. These facilities, however, are at a basic level and are intended as the primitives from which a higher-level user interface is to be constructed. Such a user interface is known as a Subsystem, which in EMAS 2900 is intended to provide such things as text editors, compilers, program loaders, subroutine libraries, run-time support, etc. Each user has his own Subsystem, although in practice this is almost always an incarnation of a standard Subsystem. This Guide describes the facilities provided by the Edinburgh Subsystem.
Supervisor	That part of the EMAS 2900 System below the Director. See Chapter 1.
Supervisor Process	See Process, above.
System	"Computer System" is the term used to describe the composite of a computer and associated software; it may also include satellite computers (e.g. Front End Processors) and associated software. The "EMAS System" (or "EMAS 2900 System"), is sometimes used to mean the software up to the Director interface, plus the paged System processes. More usually it implies a 2900 Series computer with EMAS software, up to and including Subsystems and user programs.
System File Information (SFI)	A part of a user's file index which does not relate to his disc files, but which contains information that is required to be retained between sessions, such as his passwords.
System Manager	The person who is responsible for the day-to-day running of the EMAS 2900 System, and who takes the management decisions which arise (such as those relating to the accrediting of users).
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 a communications network to the computer running EMAS 2900.
Track	On a disc or drum, the part of the surface which is accessible from a single read/write head position.

Units of Storage   The table below shows the relationship between the main units used for
                    measuring storage:

	Byte	Kbyte	Epage
1 Mbyte =	1048576	1024	256
1 Segment (max) =	262144	256	64
1 Epage =	4096	4	
1 Kbyte =	1024		

User Process   See Process, above.

Username   A unique six-character alphanumeric name assigned by the System Manager
                    to a user when he is accredited, and used to distinguish him from other
                    users. The user's process is given this name, and the full filename of
                    each of his files includes this name. The user must also specify it
                    when he is logging on to EMAS 2900.

Virtual   The primary function of the System is to maintain for each user process
                    the illusion that the process has a complete machine at its disposal
                    without interference from other users. Most of the features of EMAS
                    2900 follow logically from this objective. The components of this
                    illusion are described as "virtual", to distinguish them from their
                    "real" counterparts in the actual computer.

Virtual Address   An address by which a program accesses a location in a virtual memory.
                    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. See also Appendix 1.

Virtual Memory (VM) An apparently continuous, addressible area of up to 48 Megabytes. It
                    is mapped by the paging mechanism onto pages in the main store, or on
                    drum or disc. Of the 48 Megabytes in a complete virtual memory, 40
                    Megabytes 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.

VOLUMS   A paged Systems process responsible for the backing up, archiving and
                    restoring from archive of users' files. See Chapter 1.

REFERENCES

1. EMAS 2900: Concepts - Editor P.D. Stephens, ERCC 1980.

2. 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. Published in R.A.I.R.O., B-3, September 1975.

3. ERCC Graphics Manual - Editor N. Hamilton-Smith, ERCC 1979.

4. Edinburgh 2900 FORTRAN Language Manual, ERCC 1979.

5. Edinburgh ALGOL Language Manual - Mrs F. Stephens, ERCC 1976.

6. System Library Manual for Edinburgh 2900 Compilers - Editor Mrs L. Griffiths, ERCC 1978.

7. Edinburgh IMP Language Manual - Editor R.R. McLeod, ERCC 1974.

8. FORTRAN (G) Language - ICL 2900 Technical Publication TP 6586.

9. ALGOL (E) Language - ICL 2900 Technical Publication TP 6855.

10. Mathematical Procedures - ICL 2900 Technical Publication TP 6828.

11. OS/VS Tape Labels - IBM Form GC26-3795.

12. OS/VS2 MVS Data Management Services Guide - IBM Form GC26-3875.

13. EMAS 2900 IMP Note 4, Interactive Debugging in IMP - A. Shaw, ERCC 1979.

The UKC ICL 2960 computer is sited in the Computing Laboratory, which is part of the Cornwallis Building in the University of Kent at Canterbury. It has its own entrance adjacent to the Registry.

Service Schedule

See the notices posted at Reception, or ring 0227-66822 ext 626, or consult the Edinburgh Subsystem command HELP(SCHEDULE) for up-to-date information.

User registration

In order to become a user of the 2960 service it is first necessary to be registered. Computing Laboratory Reception should be contacted for an application form and a copy of the document "Introduction to Computing Laboratory Services".

Charges

As explained in Chapter 17, there can be charges for file space rental and charges for the use of computing resources. However, there is no charging at UKC.

Logging on

The method of logging on to the 2960 EMAS service is described in Chapter 3. The name of the service (i.e. the reply to the "HOST:" prompt) is "EMAS", or "2960".

File store

These limits can be set individually by the System Manager. The users may find out their personal limits by means of the Edinburgh Subsystem command FILES(,P).

Concurrency limit

Each user may have, at most, one batch process and one interactive process running concurrently.

CPU limits

See the notice board opposite Reception for details of current CPU limits.

The units for the second parameter of the DETACH and DETACHJOB command are "seconds".

Use of magnetic tapes

Magnetic tapes are used exclusively for the back-up and archive of disc files. Disc files are either "cherished" or "uncherished" (see Chapter 1). A cherished file will be copied to magnetic tape during the evening of every day on which it has been changed (strictly, connected in a virtual memory in write mode). This magnetic tape copy provides a back-up to the disc file, and the collection of all such copies constitutes the "back-up file store".

In addition to the back-up store, there is an archive store. This is also held on magnetic tape, but consists of files which have been moved from the disc store. Archiving is carried out once a week, on Mondays. A user can cause any of his disc files to be marked for archiving by use of the Edinburgh Subsystem command ARCHIVE (Chapter 5). In addition, a cherished file which has not been accessed for 3-4 weeks will be archived automatically. An uncherished file which has not been accessed for 3-4 weeks will be destroyed automatically.

PARM defaults

The only defaults for the Edinburgh Subsystem command PARM which differ from those given in Chapter 11 are: R8, QUOTES.

Output devices

See the UKC EMAS 2900 Information Card.

Scheduling batch work

Background, or batch, working in EMAS 2900 is handled by the paged system process SPOOLR, as described in Chapter 2. Batch work can be submitted on cards or paper tape, or can be "detached" during an interactive session (see the Edinburgh Subsystem commands DETACH and DETACHJOB, described in Chapter 16).

Normally a short job (i.e. one with a small amount of CPU time requested) will be run before a longer job. However, the scheduling scheme uses the technique of "ageing", in which increasingly more priority is given to jobs the longer they are awaiting execution. This ensures that no job can get "stuck" in the batch queue - it will eventually be executed.

Normally any job requiring 360 seconds or less of CPU time will be run during the day. However, this depends on the machine loading and may be altered at the discretion of the Management.

Scientific Jobber

The Scientific Jobber, which is available on the UKC 2960, is a batch Subsystem for fast compilation and execution of FORTRAN, ALGOL and IMP programs. Users do not need to be registered. The document "Scientific Jobber", published by ERCC, gives details of its use, together with DOC/EMAS.KO.2/1 "Using the Jobber".

It is also possible to submit a Jobber job from an interactive terminal:

        Command:JOBBER(jobfile, option)

Type HELP(JOBBER) for further details.

Amendment

The PASSWORD command is not as documented in this manual.
Please note the following:

> The PASSWORD command can be used to alter the
> passwords for this user. Two passwords are
> used (foreground and background), one for
> console use and one for identifying data input
> they can be the same. Passwords must consist
> of four printable characters.
>
> It is not possible to alter passwords from a
> batch job, for obvious security reasons.
>
> To change a password, simply type
>
> PASSWORD(F)   (for the foreground password)
>
> or
>
> PASSWORD(B)   (for the background password)
>
> You will then be asked to type the password twice,
> so that there is no confusion between what you
> typed, and what you think you typed. Terminal
> echo, and the journal, are suppressed during
> password input for security reasons.
>
> PASSWORD is equivalent to PASSWORD(F).

9 October 1981