Oxford University Computing Laboratory

Computer Manuals

Writing GEORGE 3 and 4 Job Descriptions

4401

OPERATING SYSTEMS

Writing GEORGE 3 & 4 Job Descriptions

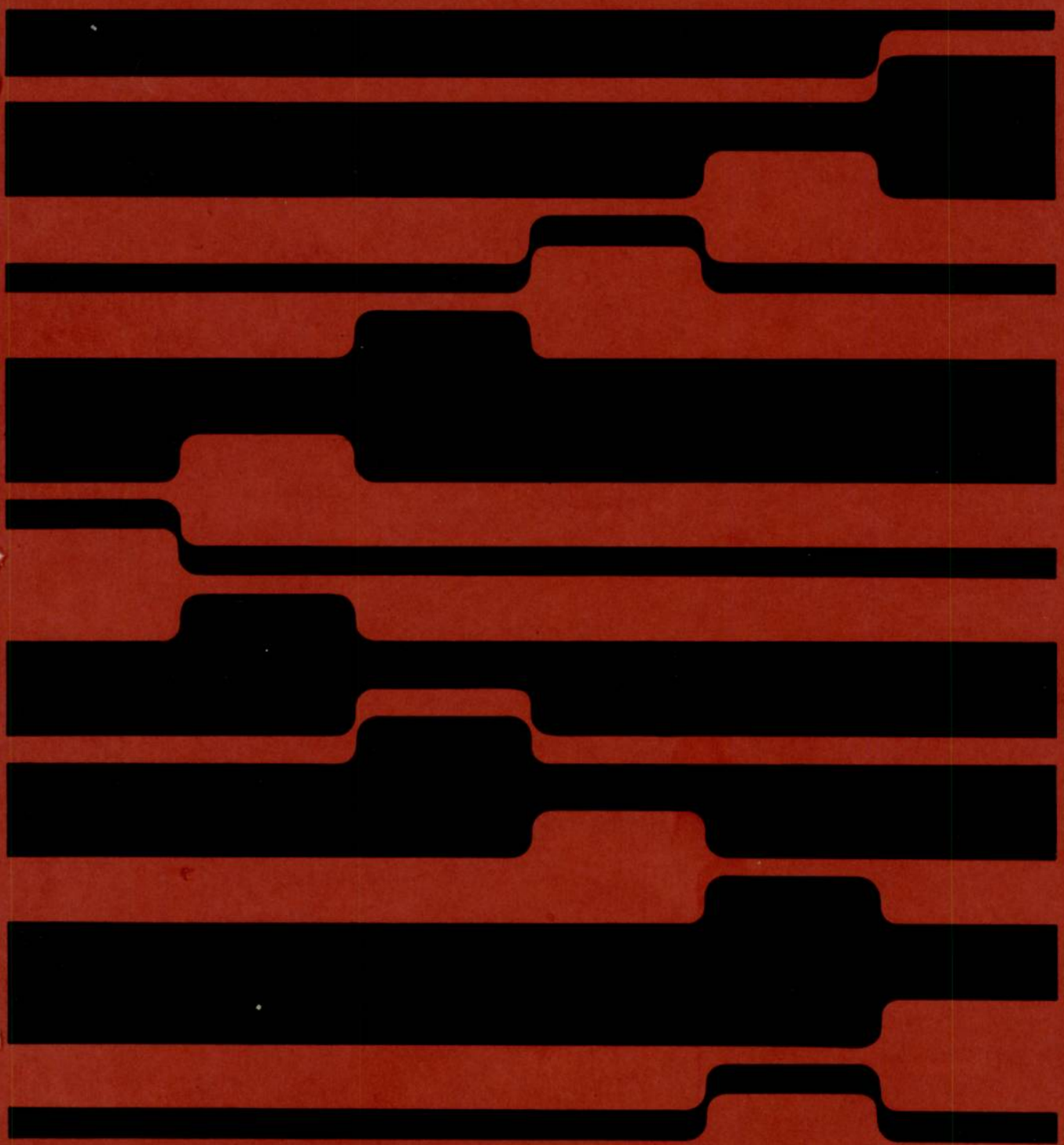
1900

ICL

Writing GEORGE 3 and 4 Job Descriptions

1900 Series

4401



ICL

Writing GEORGE 3 and 4 Job Descriptions

1900 Series

A

Technical Publication 4401

First Edition December 1974

Distributed by Software Distribution Department
International Computers Limited
Head Office: ICL House, Putney, London SW15 1SW
Printed by ICL Systemset
Works Road, Letchworth, Herts SG6 1JY

Preface

This manual describes how background jobs are run by users on those ICL 1900 Series computers employing the GEORGE 3 and 4 operating systems.

The manual is divided into three parts. Part 1 introduces the main features of GEORGE and gives an example of how a job is run under GEORGE. Part 2 describes how jobs are run using off-line peripherals which is the method recommended for the majority of jobs. Part 3 describes how jobs are run using on-line peripherals such as magnetic tapes. Part 3 also describes how the process of writing job descriptions can be simplified by making use of the macro facility. Thus the majority of users will find all the information required to run their jobs in the first two parts of this manual.

Full details of the GEORGE 3 and 4 operating systems are given in the ICL 1900 Series manual *Operating Systems GEORGE 3 and 4* (Edition 5, TP4345).

Contents

PART 1 INTRODUCTION TO RUNNING JOBS UNDER GEORGE

Chapter 1 Introduction

On computers of the ICL 1900 Series, work is executed in units called *jobs.* Although a job will frequently involve the running of a single program, a job can in fact comprise the running of several programs. Jobs may also be submitted to perform other activities, apart from the running of programs, such as program compilation.

A number of operations are involved in the process of running a job and these must be specified by the user each time the job is run. This manual describes the type of operations necessary and the way in which these operations are specified for jobs run on those computers employing the GEORGE 3 and 4 operating systems.

GEORGE 3 AND 4

GEORGE 3 and GEORGE 4 are the operating systems used with the larger computers of the 1900 Series. These systems enable many of the operations associated with running a job to be performed by the computer itself, under the direction of the user, with the minimum of operator intervention. In addition to increasing the efficiency of an installation, the GEORGE operating systems also provide a number of facilities to the user which would not be possible if an operating system was not provided.

Many facilities are common to both the GEORGE 3 and GEORGE 4 operating systems, and where this is the case the manual will refer simply to GEORGE. Where facilities differ, the number will be given.

GEORGE COMMAND LANGUAGE

In order to run a job under GEORGE, each user provides a set of instructions written in the GEORGE *command language.* This set of instructions resembles a small program and each instruction, or *command,* defines a particular operation to be performed.

Commands are normally executed in sequence although this order can be changed by using conditional commands (such as IF) and GOTO commands similar to the equivalent instructions in programming languages. In this way, the operations performed in a job can be made dependent, for example, upon the results of a program run.

A command consists of a *verb* normally followed by one or more *parameters.* The command verb (which is also the name of the command) indicates the type of action to be performed. Any parameters present give additional and more specific information about the type of action to be performed in a particular instance. A command may also be preceded by a label so that it can be referred to in conditional or GOTO commands.

An example of a command specification is:

ENTER *number*

where *number* can be from 0 to 9. The *number* parameter indicates the point at which the program is to be entered. If *number* is specified as 0, the program is entered at word 20, if 1 at word 21, if 2 at word 22, and so on up to word 29. Thus if the following command was issued:

ENTER 6

the program would be entered at word 26.

It is possible to use a shortened two-letter version of each command verb when a command is issued. For example, using the shortened form, the ENTER command above could have been issued as:

EN 6

In this manual, both versions of the command verb will be given when the format of the command is given.

Command default values

It was shown above how a program run could be initiated by entering the program at a word in the range 20 to 29. The normal entry point for a program is word 20. When the normal entry point for a program is used the *number* parameter used with ENTER can be omitted, and the run thus initiated by simply issuing the command:

ENTER

Thus, when *number* is omitted, a value of 0 is assumed by the system, and the number parameter is then said to be *set by default.* The value 0 assumed by the system is the *default value* of this parameter.

The default system is used with many other GEORGE commands. In these commands, parameters are set by default to their most frequently used values. Thus the majority of users will normally need to supply a few parameters only, when a command is issued. The parameter default values of those commands using the default system will be indicated when the command format is given.

Note: In some GEORGE commands, certain parameters are used to give additional information about operations to be performed. This information will not normally be required except in special circumstances. These parameters will thus often be omitted when a command is issued and will not be included in the command formats given in this manual. However, when necessary, the reader will be referred to either Chapter 12 or 13 of the ICL 1900 Series manual *Operating Systems GEORGE 3 and 4* (Edition 5, TP4345); these two chapters contain a full specification of all GEORGE commands.

Macros

The same set of operations will often need to be performed in different jobs. The same group of commands will thus need to be issued as part of the set of commands required to run each job. In circumstances like these, a user is able, by making use of the macro facility, to define and issue a single command to perform the operations normally performed by a group of GEORGE commands.

In order to make use of this facility, the user first creates a *macro,* which is the name given to a set of commands held on the computer's backing store. This set of commands can then be executed at any time by simply issuing a corresponding *macro command.* Macros thus provide facilities similar to those provided by subroutines and procedures in programming languages. Writing macros is described in Chapter 8.

Many GEORGE commands are in fact macro commands and thus cause other sets of GEORGE commands to be executed.

BACKGROUND AND MOP JOBS

Jobs run in batch processing mode as described in this manual are known as *background jobs.* The set of commands required to run a background job is known as a *job description* (JD). The commands in the job description are punched, on cards usually, and submitted to the operator at the installation. Background jobs may also be run by submitting the job description at a remote job entry terminal (RJE).

Jobs may also be run (using the same GEORGE command language as used to run background jobs) from remote terminals such as teletypewriters and video display units connected on-line to the computer. This facility is known as MOP (Multiple On-line Programming) and enables the user to communicate with the system while the job is running. Under MOP, the user types in a command upon receipt of a signal from the system. The command is then executed by the system. Messages are sent by the system indicating the results of the execution of the command and another signal output indicating that the next command is to be typed. In this way the type of command typed in at each stage of the job, can be made dependent upon the results of the execution of the previous command. This facility is especially useful when programs are being developed. MOP is described in the manuals *Introduction to MOP* (TP4194) and *Operating Systems GEORGE 3 and 4* (TP4345).

Usernames

In order to run a job on the computer, a user must have been allocated a *username* by the installation manager. A username is a code by which a user is known to the system and will normally reflect the area of work in which the user is engaged. For example, a username could be the name of a project or department and a username may thus be shared by a number of users.

Note: In this manual the term *user* can refer either to an individual user or to a set of users when more than one person is running jobs under the same username.

ON-LINING AND OFF-LINING

Jobs may be run making use of either *on-line* or *off-line* peripherals.

When a peripheral is used on-line, the peripheral is connected to the program and remains connected throughout the program run. Thus transmission of data from an input peripheral to a program, and transmission of data from a program to an output peripheral, occurs whilst the program is running.

When a peripheral is used off-line, the peripheral is not connected directly to the program. Instead, in the case of input peripherals, the set of data to be read by the program is transmitted from the peripheral to the computer's backing store before the program is run. Then, when the program is run, the data is transmitted to the program from the area of backing store on which it is held. Output peripherals are off-lined in a similar way: that is, data generated by the program is first transmitted to an area of backing store from where it can subsequently be transmitted to an output peripheral. Off-lining thus enables input operations to be completed before a program is run, and enables output operations to be performed at any time subsequent to the program run.

Off-lining is normally more efficient than on-lining as far as system resources are concerned, especially if slow or *basic* peripherals (such as card readers, card punches, tape readers, tape punches and line printers) are used.

Off-lining should therefore be used in preference to on-lining whenever possible.

Commands used when programs are run using off-line peripherals are described in Part 2. Commands used when peripherals are used on-line are described in Part 3, Chapter 7.

THE GEORGE FILESTORE

Off-lining makes use of the *filestore.* The filestore is the name given to a part of the computer's backing store where information can be stored by users and accessed with maximum efficiency by GEORGE.

The amount of filestore space allocated to a set of stored information is known as a *file.* The process of allocating filestore space to a set of information is termed creating a file, and the set of information is said to be contained in the file. A file could contain for example, a set of data designed to be read by a program, or a set of data generated by a program.

Files are created by users as a result of issuing certain GEORGE commands. For example, a file containing a copy of information held on a pack of punched cards can be created as a result of prefacing the card pack with a card, on which an INPUT command is punched, before the card pack is submitted to the operator. (A card containing four terminating characters must also be present as the last card of the pack.)

A name is chosen by the user for the file when it is created. The name chosen is supplied as a parameter in the command employed to create the file. The file can then be accessed at any time by simply issuing an appropriate GEORGE command, containing as one of its parameters, the name of the file. Thus users need never concern themselves with the actual location on backing store of their files.

When the information in a file is no longer required the file can be erased by issuing an ERASE command. This causes all traces of the file to be removed from the filestore. The filestore space previously allocated to the file is then available for the creation of other files.

A copy of the information held in a file can be obtained by *listing* that file. Normally, when a file is listed, a copy of the contents of that file is printed out on a line printer. However, basic files (see *File types* below) may also be listed on card punches and paper tape punches (assuming that the installation has these peripherals). In these cases the listing produced will consist of either a set of cards or a reel of paper tape as appropriate. Files are normally listed by issuing GEORGE commands such as LISTFILE, although for some types of file special utility programs are used. Note that listing a file does not cause its contents to be altered in any way and that a file may be listed any number of times.

Programs as well as data are normally also held in filestore files. This enables the speed at which they can be accessed by GEORGE in order to run them, to be greatly increased.

File handling commands are described in detail in Chapter 3.

File types

Filestore files can be created to hold information normally held on media such as cards, paper tape, magnetic tape disc and drum. The type of medium on which a set of information would be held outside the filestore determines the type of file. Thus a file created to hold a set of information normally held on media such as cards or paper tape (and thus associated with basic peripherals) is known as a *basic file.* A file created to hold information normally held on a reel of magnetic tape is known as a *magnetic tape file.* Finally, a file created to hold information normally held on magnetic disc or drum is known as a *direct access file.*

When a file containing program data is connected to a program, the program reads data from that file in exactly the same way as it would if connected to a peripheral of the appropriate type. Thus, for example, a program expecting to read data from a reel of magnetic tape, can be connected instead to a magnetic tape filestore file. Similarly, a program expecting to output data on, for example, a line printer, can be connected instead to a basic file. The use of filestore files therefore enables the action of peripherals to be simulated.

Directory files

When a username is introduced to the system, a system file known as a *directory* is created. The names of any files are automatically entered in the directory as they are created by a user operating under the username associated with the directory. Similarly, as files are erased, their names are removed from the directory.

Each directory also contains other information about files, such as the time and date when a file was created and last accessed. A user can obtain a printout, or *listing*, of the information in a directory at any time by issuing a LISTDIR command. The format of this command is described under *Obtaining information about files*, page 21.

File access

File access is controlled by the owner of that file, that is, by the user in whose directory the name of the file is entered. The owner of a file will normally be the only user permitted by the system to access that file unless he specifically grants access to another user.

Files can be accessed in different ways. For example, in order to connect a file containing program data to a program, the user must be allowed READ access to that file. Similarly, a user must be allowed ERASE access to a file before he can erase that file.

When a user creates a file, he is allowed full access to that file. He can however, restrict his own modes of access to a file by issuing a TRAPSTOP command. He might do this if, for example, he wished to prevent himself accidentally erasing a file at some time in the future. These modes of access can be restored at any time by issuing TRAPGO commands.

The owner of a group of files can also grant other users access to his files by issuing appropriate TRAPGO commands. These rights of access can subsequently be withdrawn at any time by issuing TRAPSTOP commands.

File access is fully described, and the format of the TRAPGO and TRAPSTOP commands given in Chapter 3 under *Accessing files* page 18.

Monitoring file

Each time a job is initiated, a system file associated with that job, and known as a *monitoring file*, is created. Throughout the job, any information relating to the running of the job (such as commands issued, and the results of the execution of those commands) is sent to this file. At the end of the job the contents of this file are automatically printed out on a line printer (unless the user has specifically requested that this should not be done). The monitoring information can then be examined to check that the job has been executed successfully.

BUDGETS

At most installations, before the system will allow a user to run a job, that user must have been allocated sufficient *budgets*. The budgets allocated to each user are determined by the installation manager and enable the manager to control the use of installation resources (such as computer processing time or *mill time*) and enable accounting information to be provided on users' activities.

MONEY budgets are used to control the amount of mill time and filestore space used. Each user is allocated a quantity of money (which may be real or imaginary) over a fixed period of time. During this period of time, charges are made for the use of filestore space and mill time used in running jobs. These charges are made automatically by the system and recorded in system files. The quantity of money used by a job and the total amount left is recorded, each time a job is run, in the job's monitoring file.

TIME budgets provide an additional way of controlling the amount of mill time used. When a TIME budget is employed, each user is allocated a number of units of mill time over a fixed period of time. This budget can be subdivided into amounts of time available for running jobs at different priorities. At the end of a job, the amount of mill time used by that job, and the total amount left, is recorded in the job's monitoring file.

SPACEMT budgets are used to control the use of those magnetic tapes designed explicitly for use with GEORGE. At installations where a SPACEMT budget is employed, each user is allowed to bring under his ownership as many tapes as are specified in his SPACEMT budget.

REALTIME budgets, which are similar to TIME budgets, can also be used in order to control the running of realtime programs.

Chapter 2 Running jobs under GEORGE

This chapter describes the process of running jobs under GEORGE, using as an example a job involving compiling and running a simple program. All the commands referred to in this chapter are described in detail in Part 2 of this manual. A full specification of all GEORGE commands is given in Chapters 12 and 13 of The ICL 1900 Series manual *Operating Systems GEORGE 3 and 4* (Edition 5, TP4345).

A SIMPLE JOB

Consider a job in which a program held on punched cards is to be compiled and run. The set of data required to be read by the program is also held on punched cards and the results obtained by the program are to be output on a line printer.

This job is to be run using off-line filestore files. Therefore the following operations need to be performed:

1 Filestore files containing copies of the program and associated data must be created

2 A job description must be written to compile and run the program

These operations are described in the following sections.

INPUTTING THE PROGRAM AND DATA TO THE FILESTORE

An INPUT (IN) command (see page 3) is used to create a filestore file containing a copy of information held on punched cards or paper tape. This command is issued before the cards or paper tape are read by the appropriate reader. In addition, a *terminator* must follow the cards or paper tape to indicate the end of the data. Thus the following set of cards would cause a file to be created (named SOURCEDIST) containing a copy of the user's source program; the terminator used here is the standard one of four asterisks.

```
INPUT  :CALCULATIONS,SOURCEDIST
       PROGRAM DIST
       INPUT 1 = CR1
       OUTPUT 2 = LP1
       END
       MASTER (EXMP)
10     READ (1,100) SPEED,MINS
100    FORMAT (F5.1,13)
       DIST = SPEED * MINS/60.0
       WRITE (2,200) DIST
200    FORMAT (1H ,F6.2)
       IF (SPEED .NE. 0 .AND. MINS .NE. 0) GOTO 10
       STOP
       END
****
```

The INPUT command preceding the program statements consists of the *command verb* INPUT and two *parameters.* The first parameter, :CALCULATIONS, is a *username* and the second, SOURCEDIST, is a *filename.*

A *username* is a code by which a user is known to the system and consists of a colon followed by up to 12 letters, digits or spaces beginning with a letter. Each user must be allocated a username before he is permitted to run jobs on the computer. The username chosen may be either the name of an individual or may be, for example, the name of a department or a project.

A *filename* is a name chosen by the user to identify a file. (The format of a filename is given in Chapter 3, page 11.) In this case, the file created as a result of the INPUT command will be named SOURCEDIST. This file will contain a copy of the source program which is designed to calculate the distance travelled by an object given the necessary speed and time data.

The program's data, also held on cards, is input in a similar way. Here the file created will be named DISTDATA:

```
INPUT :CALCULATIONS,DISTDATA
 37.6 52
 41.2 57
 39.9 57
 48.2 59
 61.7 61
 63.5 49
 54.1 50
 89.8 70
  0.0  0
****
```

Command format

In the examples above each INPUT command and each terminator would be punched on a separate card in the same way as the program statements and data. In the case of paper tape, a newline character would be used to separate the command or terminator from the other records.

Normally each command will occupy one record, that is, either a single card, or the characters between two newline characters. However, if necessary, a command may be carried over several records by specifying a hyphen as the last non-space character of each record.

Commands are punched in free format. The command verb (in this case INPUT) is separated from the first parameter by one or more spaces. Each subsequent parameter is separated from the previous one by a comma. Extra spaces may be included between parameters (before and after commas) to improve readability. Commands may be labelled and if so the label precedes the verb and is separated from it by one or more spaces. (A label may be of any length and must start with a digit.)

Note: A terminator must always occupy the first four characters of a record.

The user's installation may specify a particular format for punching commands. For example, it may be usual to begin punching each command verb in a certain column, with any labels occupying one or more of the preceding columns. Alternatively, each command verb or label may be punched starting in the first column, that is, the first character of the record. The first format is often used as it improves the readability of a job description.

COMPILING AND RUNNING THE PROGRAM

A job description can be submitted to the operator once the program and data have been submitted to the operator and input. The JD will instruct GEORGE to carry out the following functions:

1 Produce an object program from the source program held in the file SOURCEDIST

2 Set up a connection between the existing file DISTDATA and the object program to enable the program to read from this file.

3 Create an empty file and set up a connection between this file and the program to enable program-generated data to be written to this file.

4 Run the program.

5 Print the contents of the file containing program-generated data on a line printer.

The job description

The operations above could be carried out by the commands in the JD shown below, which would be punched on either cards or paper tape:

```
JOB :CALCULATIONS, DISTCOMPRUN
WHENEVER COMERR, ENDJOB
FORTRAN *CR SOURCEDIST
SAVE OBJECTDIST
ASSIGN *CR1, DISTDATA
ASSIGN *LP1, DISTRESULTS
LISTFILE DISTRESULTS, *LP
ERASE DISTRESULTS
```

```
ENTER
ENDJOB
****
```

The JOB (JB) command (see page 25) introduces the job to the system and identifies it with the name DISTCOMPRUN. This command also sets up a monitoring file to which information relating to the running of the job will be sent. The username :CALCULATIONS will be used to charge for the job.

The WHENEVER (WE) command (see page 26) is an example of a command that may be used to increase job and system efficiency. In this case the job is abandoned if a *command error* occurs anywhere within the JD. (A command error can arise either from a command being incorrectly specified or may occur because of some external circumstance such as a file not existing.) This command thus prevents the user's budgets being wasted when a job goes wrong.

The FORTRAN command loads and runs the FORTRAN compiler. The parameter *CR SOURCEDIST specifies that the program to be compiled is in card format and is held in the file SOURCEDIST. Program compilation is described in Chapter 6, page 31.

If the source program has compiled successfully, the object program produced will be held in main store and can be run once the file connections have been made. After the run has been completed the object program is deleted from main store and thus lost to the user. However, a SAVE command may be issued before the program is run if the user wishes to retain a copy of the object program for future use.

The SAVE (SV) command (see page 31) issued here creates a file named OBJECTDIST containing a copy of the object program. In future runs the program can be loaded from this file (using the LOAD command) thus avoiding the need for re-compilation.

The ASSIGN (AS) commands (see page 15) specify files to which data will either be read from or written to by the program. The first command connects the file DISTDATA to the program's *input channel.* The second command creates an empty file named DISTRESULTS and connects it to the program's *output channel.* If the file DISTRESULTS already existed it would be overwritten provided that the user was allowed WRITE access to it (see Chapter 3, page 11).

The program's input and output channels were defined in the program by the statements:

```
INPUT 1 = CR1
OUTPUT 2 = LP1
```

CR1 identifies a card reader from which the program expects to receive data when a READ state executed. Similarly, LP1 identifies a line printer to which the program expects to output data when a WRITE statement is executed. As files are being used instead of peripherals, the ASSIGN commands associate each channel with the appropriate file, by specifying a *peripheral name* parameter (*CR1 and *LP1) corresponding to the peripheral specified in the statements above. When a READ or WRITE statement is executed a *peripheral transfer* is said to have taken place on either *CR1 or *LP1 as appropriate.

Note: In this manual the term *program channel* is used for programs written in other languages besides FORTRAN. Thus if the program being run in this job was written in COBOL, the input and output channels would be defined in the program's environment division by statements of the form:

```
SELECT filename1, ASSIGN CARD-READER 1.
SELECT filename2, ASSIGN PRINTER 1.
```

where $filename_1$ and $filename_2$ are names used within the program and are independent of the filenames specified in the GEORGE ASSIGN commands.

The next two commands, LISTFILE (LF) and ERASE (ER), are not executed until the program run (initiated by ENTER) has ended. These commands are not executed until then because the file to which they refer (DISTRESULTS) remains open while it is ASSIGNed to the program and an open file cannot be listed or erased. Once the program run has ended all ASSIGNed files are closed and can then be listed and erased. LISTFILE and ERASE commands should always be issued before (instead of after) the ENTER command as it is more efficient in terms of GEORGE processor time to do so.

The ENTER (EN) command (see page 29) is executed after the file connections have been made. This causes the execution of the program to commence.

The LISTFILE (LF) command (see page 19) is executed at the end of the program run. This command causes a copy of the contents of the file DISTRESULTS (containing program-generated data) to be printed, or *listed,* on a line printer (*LP). Listing a file does not cause its contents to be altered in any way and the file remains in

existence for as long as the user requires. Thus further LISTFILE commands could be issued if additional copies of the file were required. Note that if the program was run on-line (that is using peripherals instead of files) an additional program run would be required for each additional copy of program-generated data.

The ERASE (ER) command (see page 20) removes all traces of the file DISTRESULTS from the filestore, as this file is no longer required. Files should be erased whenever possible in order to prevent the filestore becoming overloaded and thus to increase system efficiency.

The ENDJOB (EJ) command (see page 25) terminates the job and causes a copy of the job's monitoring file to be listed on the line printer. This file contains information relating to the running of the job, such as the length of time taken by the job and whether or not each command was executed correctly.

A terminator must always be present after the last command in the JD. Here, the standard terminator of four asterisks is used.

RE-COMPILING THE PROGRAM

If the source program contains errors the compilation will fail or the program will fail to run successfully. The source program must then be corrected and a job description submitted once again.

The source program can be corrected either by editing the file in which it is held (using the GEORGE Editor facility) or by correcting and re-inputting the card pack or paper tape on which the program is punched. Thus in the previous example, the card pack on which the FORTRAN program is punched could be corrected and INPUT once again to the file SOURCEDIST. A new file of the same name would then be created and the original JD would be re-submitted.

Compiling and developing programs is described in Chapter 6, page 31.

Note: In the example above it is assumed that the user is allowed WRITE access to the file SOURCEDIST when the card pack is re-input. This will normally be the case unless he has issued a TRAPSTOP command (denying himself WRITE access to this file) since the file was created. Accessing files is described in Chapter 3, page 11.

RUNNING THE PROGRAM ON SUBSEQUENT OCCASIONS

In the previous JD, a copy of the object program was stored in the file OBJECTDIST. The program can now be re-run without the need for re-compilation by loading the object program into main store from this file. This procedure is illustrated in the JD shown below.

This JD is basically an amended version of the first with a LOAD command replacing the FORTRAN command. (In the previous JD the loading process was carried out automatically by the FORTRAN command after compilation). However, this JD contains a number of other features designed to illustrate some other important facilities of the GEORGE command language:

1 A *comment line* appears: this is used for annotation purposes only

2 A *workfile* is used to store program output

3 In the event of a program failure, an error message is sent to the job's monitoring file

```
JOB :CALCULATIONS, DISTRUN
# THIS JOB LOADS AND RUNS THE PROGRAM HELD IN OBJECTDIST
WHENEVER COMERR, ENDJOB
LOAD OBJECTDIST
ASSIGN *CR1,DISTDATA
CREATE !
ASSIGN *LP1, !
LISTFILE !, *LP
ERASE !
ENTER
IF FAILED, GOTO 1
ENDJOB NONE
1 DISPLAY 0, JOB HAS FAILED
ENDJOB
****
```

A comment line appears in the second line after JOB. Comments are not commands and are used to provide documentation of the JD. The character # must always appear as the first non-space character of a record to indicate a comment. A comment cannot be carried over to the next record by using the continuation character (-).

The LOAD (LD) command (see page 29) causes the object program to be loaded into main store from the file OBJECTDIST.

In the previous job a filename was specified when each file was created. When a filename is specified the file created will be a *named file* which will remain in existence until explicitly erased by the user. If instead of a filename the symbol ! is specified, the file created will be a *workfile*. A workfile is a temporary version of a named file, can only be created within a job, and is automatically erased at the end of the job in which it was created.

In this job a workfile is used to store program output as this file will not be required in subsequent jobs. The workfile is set up by a CREATE command before being ASSIGNed, and is then LISTFILEd and ERASEd in the same way as a named file. Although the workfile will be erased by the system at the end of the job, it is more efficient in terms of job time to issue an ERASE command.

Workfiles should be used whenever possible as they are accessed more quickly than named files. The format of a workfile name, when more than one workfile is used within a job, is given in Chapter 3, page 11.

The IF command (see page 27) is used to identify conditions which may arise within the job, and to specify the action to be taken should these conditions arise. In this case a GOTO (GO) command will be executed if the program fails to complete its run. If the run is successful GOTO will be ignored and the next command, ENDJOB NONE, will be executed and the job terminated. The parameter NONE indicates that the monitoring file is not to be listed.

The GOTO (GO) command (see page 11), if executed, causes a branch to the command labelled 1, that is the DISPLAY (DP) command. The first parameter, 0, is a *routing parameter*. This indicates that the text JOB HAS FAILED is to be sent to the monitoring file. This message will be printed with the rest of the contents of the monitoring file when the ENDJOB command is executed.

Running programs is described in Chapter 5, page 29.

COMMAND CONTEXTS

Each GEORGE command may only be issued in certain appropriate *contexts*. These contexts depend upon the nature of the command. For example, a program cannot be entered before it is loaded, that is, until there is a *core image*. The ENTER command is therefore allowed only in CORE IMAGE context. If an attempt is made to issue this command in NO CORE IMAGE context, a command error will result.

Two of the most important command contexts are USER and NO USER. A command allowed in USER context only, must be issued within a job. A command allowed in NO USER context only, must be issued outside a job.

The examples in this chapter illustrate the concept of USER and NO USER contexts. The INPUT command, which was used to create files containing a copy of the user's source program and data, was issued outside a job, and thus in NO USER context. The JOB command, used to initiate the job is also issued in NO USER context. All commands following JOB are issued in USER context.

The INPUT command may also be issued within a job, although in this case the username parameter is omitted. Thus INPUT may be issued in either USER or NO USER context.

A further distinction is made in the *type* of job in which a command may be issued. Commands issued in background jobs, as described in this manual, are said to be issued in OFF-LINE context. Commands which may be issued from a MOP terminal are said to be issued in MOP context. Most commands may be issued in either context.

PART 2 RUNNING JOBS USING OFF-LINE PERIPHERALS

Chapter 3 Files

This chapter describes the different types of filestore file and how file handling operations (such as connecting data files to programs) are carried out using GEORGE commands. Peripherals may also be connected directly to programs, although this process, known as *on-lining*, is normally less efficient. A full description of on-lining is given in Chapter 7, page 37.

TYPES OF FILESTORE FILE

Filestore files are of different *types* to reflect different types of data. Thus, for example, a set of data held in a *drum* file will be organised in the same way as data held on a magnetic drum.

The different types of filestore file can be classified into three main groups:

1 BASIC FILES These consist of a sequence of records which are accessed in order. They are used to simulate the action of basic peripherals such as card readers, paper tape readers, card punches, paper tape punches and line printers

2 MAGNETIC TAPE FILES These have a format which allows GEORGE to simulate the action of magnetic tape peripherals. In this way programs can use large records and refer to sentinels

3 DIRECT ACCESS FILES These files simulate the action of magnetic discs and magnetic drums. Thus data held in these files can be organised in any of the ways allowed for direct access peripherals

Because each of these three groups of files has a different structure, GEORGE treats each somewhat separately. For example, data cannot be INPUT to a direct access file and a magnetic tape program peripheral channel cannot be ASSIGNed to a basic file. Full details of file handling operations appropriate to each group of files are given later in this chapter.

Basic files

There are 3 types of basic file:

1 GRAPHIC These files contain GRAPHIC data, that is, data in which the characters belong to the basic ICL 64-character set. Most data is GRAPHIC and users are recommended to use this type of file whenever possible since it is most efficient to represent internally, to edit, and to manipulate generally

2 ALLCHAR These files contain ALLCHAR data, that is, data belonging to the 128-character 1900 Series shift set. This set includes lower case letters and certain control characters

3 NORMAL These files contain NORMAL data. NORMAL data is essentially the same as ALLCHAR data except that the two characters 'runout' and 'delete' are not included

NORMAL and ALLCHAR files are often referred to collectively as *shift* files.

Magnetic tape and direct access files

Magnetic tape files are of only one type. Direct access files are of two types, DA (disc) and DR (drum).

Named files and workfiles

Each type of filestore file may be either a *named file* or a *workfile*. A named file is a file that once created, remains in existence until explicitly erased by the user. A workfile is a file that can only be created within a job (that is, in user context) and is automatically erased at the end of the job.

Workfiles are more efficient than named files as far as GEORGE resources are concerned. A full description of their use is given in *Workfiles*, page 22.

A named file will thus normally be created when the same file is to be accessed in different jobs. A workfile will normally be used when a file that is created within a job contains data (such as program output) that once listed, or read by another program in the same job, is no longer required.

REFERRING TO FILES

A file-handling command will normally contain a *file description* parameter. Where this is the case, the user is permitted to specify either a named file or a workfile. However, when instead a command contains a *filename* parameter, the user is permitted to specify a named file only.

This section describes the format of a filename, used when referring to named files. The format of a *workfile name* is given in the section *Workfiles*, page 22. In the rest of this chapter filenames will be used when illustrating the use of file handling commands.

The format of a filename

A filename has the format of a *general local name.* This consists of a *local name* followed by either a *generation name* or a *language code,* or both. The use of generation numbers and language codes is optional, and in many cases a filename will be simply a local name.

A local name consists of up to 12 letters, digits, hyphens or spaces and must begin with a letter. Examples of valid local names are:

VIBRATIONS
MASTER-DATA
ACCOUNTPROG1
PROGRAM DATA

A user can make use of the generation number and/or language code facility if he wishes to retain the same local name for each file in a group of related files. This is illustrated in the example below. Here, two files having the same local name are distinguished from each other by different generation numbers:

MASTERFILE(1)
MASTERFILE(2)

The format of a filename when generation number and language code details are included is:

local name (generation number/language code)

The *generation number* can be any number between 1 and 4095 inclusive. The *language code* consists of up to four letters or digits and must begin with a letter. The solidus (/) must be present if a language code is specified, either by itself or together with a generation number. This is illustrated in the following examples:

STOCKUPDATE (/PROG)
STOCKUPDATE (4/DATA)

The generation number facility is especially useful in jobs where file updating is involved. For example, suppose a file containing details of a company's stock is to be periodically updated. This file, when first created, was named STOCKFILE(1). From this file, an updated version named STOCKFILE(2) has been created. Subsequent versions are produced in the same way using the latest version of the file. If the files are numbered in ascending order as they are created, the latest version of the file can be referred to by simply specifying the name STOCKUPDATE. Thus the system will always access the file having highest generation number if this number is not specified.

The relative generation number facility may also be used. In this case the generation number is preceded by either a + or − sign. This facility allows a user to refer to files relative to that having the highest generation number. For example, if the user owned 4 files named:

STOCKUPDATE(1)
STOCKUPDATE(2)
STOCKUPDATE(3)
STOCKUPDATE(4)

the name STOCKUPDATE(−1) could be used to refer to the file STOCKUPDATE(3). Similarly the name STOCKUPDATE(+1) could be given if the user wished to create a file named STOCKUPDATE(5) and thus having a generation number one above that of the file having previous highest generation number.

If relative generation numbers are used, the same JD can be submitted each time a file-updating job is run. The latest version of the file is accessed by specifying a filename without a generation number such as STOCKUPDATE. The updated version of the file is created with highest generation number by using a name of the form STOCKUPDATE(+1).

REFERRING TO OTHER USERS' FILES

In most jobs a user will be accessing his own files and the file will be referred to in a command by either a *filename* or a *workfile name* as appropriate. The user may however need to access one or more named files belonging to another user. He can do this either by using an absolute name or by changing his directory. These processes are described in the sections below.

Note: The system will not allow a user to access files owned by another user unless the owner has granted him permission to do so. The way in which file access is controlled is described in a later section *Accessing files*, page 18.

Absolute names

A user can refer to a file owned by another user by an *absolute name*. This has the format

username.filename

where *username* identifies the owner of the file. Thus a user such as: RESEARCH could issue the command

LOAD :ACCOUNTS.OBJPROG

if he wished to load a program from the file OBJPROG belonging to the user :ACCOUNTS.

Changing the directory

Associated with each username is a *directory*. A directory is a file containing a list of the names of files (and magnetic tapes) owned by a particular user. Directories are system files that are automatically updated as files are created and erased. A user can check the contents of a directory by issuing a LISTDIR (LD) command (see the section *Obtaining information about files*, page 21).

The user's own directory is known as his *proper directory*. The *current directory* of a user in a job will normally be his proper directory unless he has changed his directory since the job started. If the directory has not been changed the user is said to be running the job under his proper directory.

The user's directory can be changed to that of another user by issuing a DIRECTORY (DY) command:

DIRECTORY *username*

where *username* identifies the directory of the associated user. When this command has been issued, all files referred to by *filename* alone by the *current user* will identify files belonging to the *proper user*, that is, the owner of the directory. The use of DIRECTORY is illustrated below:

```
JOB :RESEARCH,JOB22
—
—
DIRECTORY :ACCOUNTS
LOAD OBJPROG
ASSIGN *CR1,OBJPROGDATA
—
—
ENDJOB
****
```

This would be equivalent to issuing a JD of the form:

```
JOB :RESEARCH,JOB22
—
—
LOAD :ACCOUNTS.OBJPROG
ASSIGN *CR1,:ACCOUNTS.OBJPROGDATA
—
—
ENDJOB
****
```

Note: The job's proper directory is not changed by a DIRECTORY command. Thus a user cannot acquire another user's budgets to gain access to another user's files by this command.

TRANSFERRING INFORMATION TO THE FILESTORE

Filestore files can be created to hold copies of information from cards, paper tape, magnetic tapes and other magnetic media. The following sections show how these files are created by giving appropriate GEORGE commands.

Inputting information held on cards or paper tape

An INPUT(IN) command is used to create a basic file containing a copy of information held on either punched cards or paper tape. The INPUT command has two formats:

1 INPUT *username,filename,terminator*

2 INPUT *file description, terminator*

The first format is used when a named file is being created outside a job, that is, in no user context. Examples of the use of INPUT having this format were given in Chapter 2, page 5.

The second format is used when a file is being created in user context. In this case the INPUT command must be issued within a job when the username has already been established. Thus the file created can be either a named file or a workfile. An INPUT command issued within a background job is known as an embedded INPUT command. An example of INPUT used in this way is given in a later section, *Workfiles*, page 22.

The terminator need not be specified if the set of information is followed by the standard terminator of four asterisks. In this case the four asterisks are stored at the end of the file followed by a blank record. The next example illustrates this use of INPUT:

```
INPUT :ACCOUNTS,PROGRAMDATA
first record
.
.
.
last record
****
```

In certain cases the user may need to specify a different terminator. For example, a program may expect to read a particular set of characters after the last record in a file. In this case, the user can specify the appropriate four character terminator in the INPUT command.

The terminator parameter in the INPUT command consists of the letter S, or the letter T, followed by the four characters chosen for the terminator. If S is specified, the terminator will be stored at the end of the file followed by a blank record. If T is specified, the terminator will not be stored. These two cases are illustrated in the following examples:

```
INPUT :ACCOUNTS,PROGRAMDATA,S1234
first record
.
.
.
last record
1234
INPUT :ACCOUNTS,PROGDATA,T????
first record
.
.
.
last record
????
```

The S type terminator is useful when the program for which the data is intended uses double-buffering.

Note: As the second format of INPUT (described at the beginning of this section) will be used within a job, the terminator chosen must be different from the terminator appearing at the end of the job description.

All files created from punched cards will be of type GRAPHIC. A file created as a result of inputting paper tape containing GRAPHIC or shift data will normally also be GRAPHIC. However, if the tape holds shift data, the user can specify that a shift file be created by qualifying the filename with either ALLCHAR or NORMAL as appropriate. Thus, in the next example, a shift file containing ALLCHAR data, would be set up:

INPUT :ACCOUNTS,TAPEDATA(ALLCHAR),S////
first record
.
.
.
last record
////

Inputting information held on magnetic media

Subfiles on a magnetic tape can be copied into basic files in the filestore by means of the COPYIN or FILEIN system macros. The converse operation is performed by the COPYOUT system macro. Full specifications of these macros are given in Chapter 12 of *Operating Systems GEORGE 3 and 4.*

It is also possible to create magnetic tape filestore files from magnetic tapes by using special utility programs. These programs are described in the ICL 1900 Series manual *Magnetic Tape Utilities* (Edition 1, TP4207).

Information held on direct access devices outside the filestore can be copied into filestore files by using special utility programs. These utility programs are fully described in the ICL 1900 Series manual *Direct Access Utilities* (Edition 1, TP4190).

CONNECTING AN INPUT FILE TO A PROGRAM

A file from which data will be read by a program is connected to the program's input peripheral channel by an ASSIGN (AS) command. This has the format:

ASSIGN *peripheral name,file description*

The *peripheral name* corresponds with the peripheral identified in the program's input channel. This name consists of a *peripheral type* followed by an integer in the range 0 to 64. The format of a *peripheral name* parameter is given in Table 1, below.

Peripheral name	*Peripheral specified in program*
*TR*n*	Paper tape reader *n*
*TP*n*	Paper tape punch *n*
*CR*n*	Card reader *n*
*CP*n*	Card punch *n*
*LP*n*	Line Printer *n*
*MT*n*	Magnetic tape unit *n*
*DA*n*	Magnetic disc *n*
*DR*n*	Magnetic drum *n*

Table 1 Peripheral names

Thus, in the following example, a magnetic tape file could be connected to magnetic tape input peripheral channel 0 of a program:

ASSIGN *MT0,MAGTAPEDATA

The type of peripheral specified in the program will be either basic, magnetic tape, or direct access. The file assigned must therefore be of the appropriate type. Thus, for example, a magnetic tape file or direct access file cannot be connected to a program's basic input channel.

Any basic peripheral file can be connected to a program that is expecting data from a particular basic peripheral. This is illustrated in the next example:

ASSIGN CR1,RESULTS

Here, the file RESULTS was created during a previous program run by the card punch output channel in that program. This file is now to be connected to the input channel of another program. In this program, input channel 1 specifies that data is expected from a card reader.

Note: A user must be allowed READ access to a file that is to be read by a program. This will normally be the case unless he has issued a TRAPSTOP command or used a TRAPSTOP qualifier (denying himself READ access) since the file was created. Further details are given in a subsequent section *Accessing files*, page 18.

Assigning the job source

If a program uses one basic peripheral channel for input, the data to be read by the program can be embedded in the JD after the ENTER command. The job source is then said to be ASSIGNed to the program.

The embedded data is connected to the program's input channel by an ASSIGN command having the format:

ASSIGN *basic peripheral name*

The following is an example of a job in which the data (which is to be read by the program's card reader input channel) is embedded in the JD:

```
JOB :ACCOUNTS,JOB8
WHENEVER COMERR,ENDJOB
LOAD ACCPROG
ASSIGN *LP1,RESULTS
ASSIGN *CR1
((((
ENTER
first record
.
.
.
last record
))))
IF FAILED, (GOTO 10) ELSE (ENDJOB)
10 DISPLAY 0, JOB HAS FAILED
ENDJOB
****
```

The symbols ((((and)))) are known as *command delimiters.* These should normally be present when embedded data is used and should enclose the command accessing the data and the data itself. The use of delimiters ensures that a GOTO command ignores all labels (apart from a label addressing the first line after the initial delimiter) between a pair of delimiters. Thus when delimiters are used, a GOTO command cannot attempt to read part of the embedded data as part of the JD if the start of a line of data duplicates a label.

Note: Other characters can also be used for command delimiters which should be different from the data they enclose. The use of command delimiters is fully described in Chapter 2 of *Operating systems GEORGE 3 and 4* (TP4345).

CONNECTING AN OUTPUT FILE TO A PROGRAM

A user will normally wish to create an empty file and connect it to his program's output channel. The data generated during his program run will then be sent to this file. After the program run the file can then be listed on the line printer or may be used as input data for another program.

For basic and magnetic tape files, an empty file is created and connected to the program's output channel (if a file of that name does not already exist) as a result of giving an ASSIGN command. However, in the case of direct access files, the empty file must first be set up by means of a CREATE command before the file is ASSIGNed. Details of these processes are given in the following sections.

An existing file may also be connected to a program's output channel. In this case the file will be emptied before being connected and the effect will be the same as if an empty file had been used.

It is also possible to append to an existing basic file. In this case the file will be enlarged and new data will be written to the file, beginning after the last existing record.

Note: A user must be allowed WRITE access to a file that is to be overwritten, and, in the case of a basic file, APPEND access to a file to which information is to be appended. This will normally be the case unless the user has issued a TRAPSTOP (TS) command or used a TRAPSTOP qualifier (denying himself WRITE or APPEND access as appropriate) since the file was created. Further details are given in the section *Accessing files*, page 18.

Connecting a basic file for output

An empty basic file is created and connected to a program's output channel by an ASSIGN (AS) command (if a file of that name does not already exist). This has the same format as that used for connecting a file to an input channel. Thus the command:

```
ASSIGN *LPO,RESULTS
```

would create a file named RESULTS and connect it to the program's line printer output channel number 0. If the file RESULTS already existed before ASSIGN was issued, the file would be emptied before being connected.

If the file description parameter is omitted (as shown below) program output will be sent to the monitoring file. This output will then be listed at the end of the job together with the rest of the information in the monitoring file (unless the listing of the monitoring file is suppressed by a parameter following ENDJOB).

```
ASSIGN *LP1
```

The filename must be qualified by APPEND if the user wishes to write to the end of an existing basic file. For example:

```
ASSIGN *LP1,PROGRESULTS(APPEND)
```

Connecting a magnetic tape file for output

An empty magnetic tape file is created and connected to a program's output channel by an ASSIGN (AS) command in which the filename is qualified by WRITE. This is illustrated in the next example, where the file is connected to magnetic tape output channel 2 of the program:

```
ASSIGN *MT2,MTOUTPUTDATA(WRITE)
```

If an existing magnetic tape file is to be completely overwritten, the filename should be qualified by EMPTY (as shown below) instead of WRITE, for reasons of system efficiency.

```
ASSIGN *MT3,MASTERFILE(4)(EMPTY)
```

Connecting a direct access file for output

An empty direct access file must first be set up by a CREATE (CE) command before it is ASSIGNed to a program's output channel. The CREATE command has the format:

CREATE *file description(qualifiers)*

The first qualifier must be either *DA or *DR specifying whether the file to be created is disc or drum. The second qualifier is KWORDS *number* which specifies the maximum size of the file in units of 1024 words. For example, the command:

```
CREATE NEWFILE(*DR,KWORDS8)
```

will set up a drum file having a size of 8K, that is, 8192 words.

In the case of a disc file, other qualifiers may optionally be specified to give additional information about the organisation of the file. For example, in the command:

```
CREATE NEWDISCFILE(*DA,KWORDS6,BUCKET8)
```

the parameter BUCKET8 indicates that the data in the file is to be organised into 8-block buckets. If this parameter is not specified, the data will be organised into 4-block buckets. A list of the other optional qualifiers, and their associated default settings, is given under the CREATE command in Chapter 12 of *Operating Systems GEORGE 3 and 4* (TP4345).

The file set up by a CREATE command is connected to the program's output channel in the normal way by using an ASSIGN (AS) command in which the filename is qualified by WRITE. Thus the file created in the previous example would be connected to output channel 1 of a program by the command:

```
ASSIGN *DA1,NEWDISCFILE(WRITE)
```

If an existing direct access file is to be overwritten the filename should be qualified by EMPTY instead of WRITE for reasons of system efficiency.

The name of an existing disc file should be qualified by OVERLAY if the file is to be written to in open mode 100 or qualified by OFFSET if the file is to be written to in open mode 400. Details of open modes are given in the ICL 1900 Series manual *Direct Access* (Edition 2, TP4385).

ACCESSING FILES

Every file can have one or more *user traps* associated with it. These traps specify which users are allowed to access the file and in what way access is permitted. The user can alter the traps by giving appropriate GEORGE commands: this is described in the section *Altering traps*. For example, a user may wish to set traps to prevent himself accidentally erasing the contents of a file.

There are five types of traps:

READ	The user can read, list or copy (see page 20) the contents of a file
WRITE	The user can write to the file from the beginning thus overwriting the existing contents
APPEND	The user can write further information to the end of the file
EXECUTE	The user can execute the contents of the file, for example, load and run a binary program held in a file, or use the file as a macro (see Chapter 8, page 43)
ERASE	The user can erase all traces of the file from the filestore, or can rename the file (see page 20)

When a file is initially created, the user is permitted access in ALL (that is READ,WRITE,APPEND,EXECUTE and ERASE) modes. The user to whom the file belongs controls the access of himself and all other users to it. Unless he specifically grants access to another user, he will be the only one permitted to access it.

Altering traps

The user can prevent himself accessing a file in any of the five modes by giving a TRAPSTOP (TS) command:

TRAPSTOP *file description,access mode,access mode. . .*

For example, the commands:

```
TRAPSTOP DATAFILE,WRITE
TRAPSTOP BLUEFILE,ALL
```

would prevent the user overwriting the contents of DATAFILE and prevent him from accessing BLUEFILE in any way.

Similarly, a TRAPGO(TG) command can be given to allow the user to access a file in any of the permitted modes:

TRAPGO *file description,access mode,access mode. . .*

Thus, if the user wished to be able once again to WRITE and APPEND to the file BLUEFILE (referred to in the previous example), he would give the following command:

```
TRAPGO BLUEFILE,WRITE,APPEND
```

Granting other users access to files

The owner of a file can allow another user to access it by giving a TRAPGO(TG) command specifying the appropriate username:

TRAPGO *file description,username,access mode,access mode. . .*

Thus the command:

```
TRAPGO DATAFILE, :PLANNING,READ
```

would allow the user with username :PLANNING to read the contents of the file called DATAFILE.

A TRAPSTOP command, having the same parameter format, can be given to withdraw, from other users, the rights of access to a file.

The TRAPGO and TRAPSTOP qualifiers

File access can also be controlled by using TRAPGO(TG) and TRAPSTOP(TS) file description qualifiers. These have the format:

TRAPGO(*access mode, . . .*)
TRAPSTOP(*access mode, . . .*)
TRAPGO(*username,access mode, . . .*)
TRAPSTOP(*username,access mode, . . .*)

The first two formats are used to specify the access modes permitted by a user to his own files. The last two formats are used when a user is specifying the access modes permitted by other users to his named files.

These qualifiers are often used when a file is being created, by for example, an INPUT or ASSIGN command. Thus the command:

INPUT :ACCOUNTS,PROG22(TRAPSTOP(WRITE,ERASE))

would create a file named PROG22 which the owner :ACCOUNTS would not be able to overwrite or erase. The command:

ASSIGN *CP0,CARDDATA(TRAPGO(:RESEARCH,READ))

would create a file named CARDDATA (if a file of this name did not already exist) and connect it to the program's card punch output channel. The user :RESEARCH would be permitted to read from this file.

Checking traps

A user can check whether he is allowed access to a file (belonging to himself or another user) and if so, in what modes access is permitted, by issuing a TRAPCHECK(TC) command:

TRAPCHECK *file description*

The system replies to any TRAPCHECK command and sends this information to the monitoring file. For example a command (given by the user :ACCOUNTS) such as:

TRAPCHECK MYFILE

might generate a reply such as:

PERMITTED ACCESS MODES IN MYFILE BY :ACCOUNTS: READ, EXECUTE,WRITE

If the user :ACCOUNTS was not allowed access to a file OBJPROG belonging to the user :RESEARCH, the following command (issued by :ACCOUNTS):

TRAPCHECK :RESEARCH.OBJPROG

would generate the reply

YOU ARE NOT ALLOWED ANY ACCESS TO THIS ENTRANT

A user can also check the traps of his own files to see whether a specified user is permitted access. In this case the TRAPCHECK command used has the format:

TRAPCHECK *file description,username*

For example :ACCOUNTS would issue the command

TRAPCHECK MYFILE,:RESEARCH

if he wished to know in what modes (if any) :RESEARCH was allowed access to the file MYFILE owned by :ACCOUNTS.

LISTING FILES

Normally, when a file is *listed,* a copy of the contents of that file will be printed out on a line printer. However, in the case of basic files, it is possible to list files on a card or tape punch (assuming that the installation has these peripherals). The following sections describe how basic, magnetic tape and direct access files are listed.

Listing basic files

Basic files are listed by giving a LISTFILE (LF) command:

LISTFILE *file description, output peripheral type,* FROM *number,*LINES *number,* NUMBER

All parameters are optional apart from the first. In this case, when the only parameter specified is *file description,* the contents of the file will be sent to the job's monitoring file. The contents of the file will then be listed on a line printer at the end of the job, together with the job's monitoring information.

The *output peripheral type* may be either *LP for a line printer, *CP for a card punch, or *TP for a paper tape punch. For example the command:

LISTFILE OUTPUTDATA,*LP

would cause the contents of the file OUTPUTDATA to be listed on a line printer.

Any type of basic file may be listed on any of the three types of output peripheral. However, if a shift file is listed on a line printer or card punch the data is converted to GRAPHIC as it is output.

The parameter FROM *number* causes the file listing to begin at the line specified. Thus, the command:

LISTFILE OUTPUTDATA,*LP,FROM 30

would cause the file to be listed on the line printer, beginning at the 31st line. (Numbering starts at 0.)

The LINES *number* parameter indicates how many lines of the file are to be listed. Thus, in the following example:

LISTFILE OUTPUTDATA,*CP,LINES 100

the first one hundred lines of the file OUTPUTDATA will be listed on the card punch.

The FROM *number* and LINES *number* parameters may be used in conjunction, if required, as in the following example:

LISTFILE OUTPUTDATA,*TP,FROM 60,LINES 90

The NUMBER parameter causes the lines or cards to be numbered as they are output. This parameter is not allowed in conjunction with device type *TP.

A PROPERTY parameter can also be specified if for example the file is to be listed on a peripheral with certain specified properties, or special stationery is required. For further details see the specification of LISTFILE, Chapter 12, *Operating Systems GEORGE 3 and 4* (TP4345).

Note: Under the MK8.20 release of GEORGE, the LISTFILE command will be enhanced to provide additional facilities for the selective listing of files. These facilities will be described under the specification of the LISTFILE command in the above manual.

Listing magnetic tape files

The MTPRINT command can be used to list magnetic tape filestore files. This command provides a selective printing of the file in subfile format. A full specification of this command is given in Chapter 12 of *Operating Systems GEORGE 3 and 4* (TP4345).

Listing direct access files

Special utility programs, such as HRAPRINT, are used to list the contents of direct access files. A full description of this, and other utility programs, is given in the manual *Direct Access Utilities* (TP4190).

ERASING FILES

To remove all traces of a basic, magnetic tape or direct access file from the filestore, an ERASE (ER) command, having the following format, is given:

ERASE *file description*$_1$,*file description*$_2$,...,*file description*$_n$

Thus the command:

ERASE MYFILE

will remove MYFILE from the filestore, and the command:

ERASE PROGFILE,CALCFILE,OBJCALC(−1)

will remove PROGFILE, CALCFILE and OBJCALC(−1) from the filestore.

COPYING FILES

A copy of an existing file can be created by giving a COPY (CY) command:

COPY*filename,filename*

The first parameter is the name of the file to be copied; the second is that of the file to be created or overwritten.

By issuing this command, a user can create a file in his own directory, which is a copy of another user's file. Thus the command:

COPY :FRED.DATAFILE,MYFILE

will create a file MYFILE (belonging to the user issuing the command) which is a copy of DATAFILE held under username :FRED.

EDITING BASIC FILES

A user may wish to alter or remove certain records in a basic file or add records to a basic file if for example the file contains errors or needs to be amended before being read by a program. These amendments can be carried out by using the GEORGE Editor facility which is described in Chapter 6 of *Operating Systems GEORGE 3 and 4* (TP4345). Files may also be merged using the Editor.

OBTAINING INFORMATION ABOUT FILES

Each user has a directory file containing a list of files and magnetic tapes belonging to him. The directory also contains other information about the files and tapes, for example, when a file was last written to. The contents of a directory can be listed by a LISTDIR (LD) command:

LISTDIR *username,output level, output peripheral type*

The *username* identifies the directory to be listed. If *username* is omitted the job's current directory will be listed. The *output level* may be specified as LOW or may be HIGH if a more detailed listing (specifying for example when each of the user's files was created) is required. If *output level* omitted a LOW level listing will be produced. The *output peripheral type* must be *LP. If it is omitted information is sent to the job's monitoring file.

Examples of this command are:

```
LISTDIR
LISTDIR ,,*LP
LISTDIR ,HIGH
LISTDIR ,HIGH,*LP
LISTDIR :RESEARCH
LISTDIR :RESEARCH,,*LP
LISTDIR :RESEARCH,HIGH
LISTDIR :RESEARCH,HIGH,*LP
```

In the first four commands the user's current directory will be listed. If, for example, the user :ACCOUNTS issued any one of these four commands his current directory would be his own directory (that is his proper directory) unless he had issued a DIRECTORY command (see *Changing the directory*, page 13).

If any one of the last four commands was issued by :ACCOUNTS, the directory of the user :RESEARCH would be listed provided that :RESEARCH had granted :ACCOUNTS permission to READ his directory.

RENAMING FILES

A user can change the name of a file by giving a RENAME (RN) command.

RENAME *filename,filename*

The first parameter is the existing name of the file; the second is the new name chosen by the user. Thus the command:

RENAME OBJPROG,NEWOBJPROG

will cause the file OBJPROG to be renamed NEWOBJPROG.

Note: A user must be allowed ERASE access to a file that is to be RENAMEd.

RETRIEVING FILES

Files are normally held on direct access backing store, but are periodically dumped on to magnetic tape backing store. The system automatically retrieves files when they are required by a job and copies them back into direct access backing store. However, this operation takes time and may delay a job when, for example, a program is waiting to be connected to a file.

A user can shorten the time required for his job by giving a RETRIEVE (RV) command at the beginning of his job description. The files specified in the RETRIEVE command will then be copied back into direct access backing store whilst other operations (such as program compilation) are being performed, and will thus be available when required. The format of the RETRIEVE (RV) command is:

RETRIEVE *filename*$_1$,*filename*$_2$,...,*filename*$_n$

For example, if the user wished to retrieve the files ACCOUNTSDATA, MASTERDATA and :RESEARCH.PROG he would give the command:

```
RETRIEVE ACCOUNTSDATA,MASTERDATA,:RESEARCH.PROG
```

WORKFILES

Workfiles are temporary files which are automatically erased at the end of the job that created them. They are useful for holding data, such as that output by a program, that is no longer required after the job has ended.

A basic or magnetic tape workfile is set up by giving a CREATE (CE) command having the format:

```
CREATE!
```

The type of file is not determined until it is referred to by another command, usually ASSIGN (AS) or INPUT (IN). For example, the command:

```
ASSIGN *LP0,!
```

would connect the workfile named ! to the line printer output channel of the program. Thus a basic file would be created which could be listed on the line printer at the end of the program run by giving the command:

```
LISTFILE !,*LP
```

A direct access workfile may also be set up and assigned in a similar way although in this case it is necessary to include details of the file when issuing the CREATE command. These qualifiers are exactly the same as those used when setting up named direct access files, describ 1 earlier in the section *Connecting a direct access file for output,* page 17. For example, the command:

```
CREATE !(*DA,KWORDS6)
```

would set up a magnetic disc workfile having a capacity of 6K. This file could then be assigned to a program in the normal way.

The following is an example of a job using a workfile:

```
JOB :ACCOUNTS,MERVYNSJOB
WHENEVER COMERR,ENDJOB
CREATE !
INPUT !,T????
lines of input data
????
LOAD OBJECT
ASSIGN *TR0,!
ASSIGN *TP0,OUTPUTDATA
LISTFILE OUTPUTDATA,*TP
ENTER
ENDJOB
****
```

This job contains an embedded INPUT command. All data following this command up to the terminator ???? is stored in a workfile, created by the previous command. This workfile is then connected to the program's tape reader input channel. Program generated data is sent to a file which is then listed on a paper tape punch. The workfile containing the input data is automatically erased at the end of the job.

The trap system (see the earlier section *Accessing files*, page 18) applies to workfiles in exactly the same way as to named files.

Using more than one workfile

Any number of workfiles can be created and used within a job. In this case each workfile is identified by a relative name. The most recently created workfile is always given the name !. The second most recently created workfile is given the name !1, the third is given the name !2, and so on.

The term *workfile stack* is often used when referring to workfiles. The newest workfile is assumed to occupy the top position of the stack whilst the oldest is assumed to occupy the bottom position.

The following example illustrates the concept of the workfile stack. Here, four workfiles have been created within a job by the commands:

(a) CREATE !

(b) CREATE !

(c) CREATE !

(d) CREATE !

The diagram below shows how the names of previously created workfiles change as each new workfile is created. Each column gives the name of one or more workfiles at the point when the corresponding CREATE command has been issued. Thus, for example, in column (d), !3 is the name used to identify the oldest workfile (that is, the workfile at the bottom of the stack) when the last CREATE command has been issued.

	(a)	(b)	(c)	(d)
name of fourth created workfile				!
name of third created workfile			!	!1
nameof second created workfile		!	!1	!2
name of first created workfile	!	!1	!2	!3

The next example is of a job using two workfiles:

```
JOB :ACCOUNTS,PROGRUN
WHENEVER COMERR,ENDJOB
LOAD OBJECTPROG
ASSIGN *TR1,PROGDATA
CREATE !
CREATE !
ASSIGN *CP,!1
ASSIGN *LP1,!
LISTFILE !1,*CP
LISTFILE !,*LP
ERASE !1
ERASE !
ENTER
ENDJOB
****
```

In this job the program makes use of one input channel and two output channels. The input channel is connected to the file PROGDATA which contains data designed to be read by the program. The two output channels are connected to workfiles which are to be listed at the end of the program run.

The program generates data in both card and line printer format. The first workfile is connected to the card punch output channel and the second, that is, the newest workfile, is connected to the line printer output channel. After the program run, a pack of cards is produced by listing the first workfile on a card punch, and a line printer listing is obtained from the second. Both workfiles are then erased.

Note: When a number of workfiles are used it is often easier (and more efficient as far as system resources are concerned) to ERASE them as soon as possible. Thus the above commands dealing with workfiles could have been issued as:

```
CREATE !
ASSIGN *CP1,!
LISTFILE !,*CP
```

```
ERASE !
CREATE !
ASSIGN *LP1,!
LISTFILE !,*LP
ERASE !
```

When a workfile is erased within a job by issuing an ERASE command all workfiles older than the one erased are renamed by subtracting 1 from their associated names. For example, if workfile !2 was erased, the workfiles previously referred to as !3 and !4 would now be renamed !2 and !3.

Workfiles may also be referred to by a workfile name in which the number is preceded by a minus sign. In this case, the name !−1 refers to the oldest workfile, !−2 to the second oldest, and so on.

The position of a workfile in the workfile stack can be altered by issuing a WORKFILEMOVE (WF) command. This has the format:

WORKFILEMOVE *workfile name,workfile name*

The first parameter gives the current position of the workfile. The second specifies the position to which the workfile is to be moved. Thus the command:

WORKFILEMOVE !1,!-1

would cause the workfile second from the top of the stack to be moved to the bottom. The position (and therefore names) of all intermediate workfiles will thus also be changed.

Chapter 4 Initiating jobs

This chapter describes how a job is started and ended by issuing JOB and ENDJOB commands. It also describes how, in the case of frequently run jobs, the job description itself may be stored in a permanent file and the job run by issuing a single command. The final section shows how a job may be terminated as a result of errors in commands, lack of time and events such as program failures.

STARTING A JOB

A job is initiated by giving a JOB (JB) command, which introduces the job description to the system and causes a temporary job description file (containing all commands following JOB up to the terminator) to be set up in the filestore. The job description file which is known as the job source is automatically erased at the end of the job. In addition, JOB causes a monitoring file to be set up to which all information about the job's progress will be sent. The format of the JOB command (see also page 7) is:

JOB *username,jobname,terminator*

The *jobname* parameter is the name chosen by the user for his job and consists of up to twelve letters, digits or minus signs beginning with a letter. (Note that the *jobname* must not coincide with the name of one of the user's files or must not be the same name as another job being run under the same username.) The *username* is the code by which the user is identified by the system and consists of a colon followed by up to twelve letters, digits, spaces or hyphens, beginning with a letter. The *terminator* parameter need not be specified if the job description is terminated with the standard terminator of four asterisks.

The following is an example of a JOB command which precedes a job description terminated by four asterisks:

```
JOB OFFLINEJOB,:PLANNING
```

The following two examples show how different terminators may be used:

```
JOB JOBNO1,:ACCOUNTS,SENDJ
JOB JOBNO2,:RESEARCH,T////
```

The terminators specified in these examples are ENDJ and ////. The letter S specifies that the characters ENDJ terminating the job description are to be stored. The letter T specifies that the terminating characters //// are not to be stored.

ENDING A JOB

Besides ending a job, and erasing the temporary job description file, the ENDJOB (EJ) command can provide a user with information about the job by causing the contents of the monitoring file to be listed on the line printer. The format of the ENDJOB command is:

ENDJOB *action on monitoring file,*RETAIN(*local name*)

All parameters are optional, and if none are specified the whole of the job's monitoring file will be listed on the line printer. The *action on monitoring file* parameter specifies that certain categories only, of the monitoring file information are to be printed. (A full description of these categories is given in Chapter 10 of *Operating Systems GEORGE 3 and 4* in the section *Output from the monitoring file.*) Finally, the RETAIN(*local name*) parameter allows the user to create a permanent file to contain a copy of the monitoring file information. The name chosen by the user for the file can consist of up to twelve letters, digits, spaces or hyphens, beginning with a letter and the file can be erased at any time by giving an ERASE command.

The following are examples of ENDJOB commands:

```
ENDJOB
ENDJOB NONE
ENDJOB OBJECT,RETAIN(MONFILE)
```

In the first example, the whole of the job's monitoring file will be listed. In the second example, none of the jobs' monitoring file will be listed. In the third example, object program output to the monitoring file will be listed and the whole monitoring file will be copied to a file named MONFILE.

STORING A JOB DESCRIPTION

A job description may be input and stored in a file in the same way as programs and data. The job can then be run at any time by issuing a single RUNJOB command. This procedure is useful when the same job is run regularly using different sets of data as it avoids the need to re-submit the job description for input each time. Background jobs stored in this way may also be initiated from MOP terminals by issuing an appropriate RUNJOB command in user context.

Inputting the job description

The format of a job description which is to be stored is exactly the same as a normal job description except that the first command, JOB, is replaced by an INPUT (IN) command. Non-standard terminators may be specified, if desired, in the same way as for JOB, and these are described in Chapter 3 in the section *Inputting information held on cards or paper tape*, page 14.

The following is an example of a job description (designed to update a disc master file from data punched on paper tape) which is to be input to a file named JOBDESCRIP:

```
INPUT :ACCOUNTS,JOBDESCRIP
WHENEVER COMERR,ENDJOB
LOAD UPDATEPROG
ASSIGN *TR0,UPDATEDATA
ASSIGN *DA0,MASTERFILE
ENTER
ENDJOB
****
```

Running a job from a stored job description file

A single RUNJOB (RJ) command, punched on cards or paper tape, is given to the operator to initiate a job from a stored job description. The format of the RUNJOB command (see also above) is:

RUNJOB *username,jobname,filename*

The *jobname* parameter is the same as that used with the JOB command, described earlier. The *filename* parameter specifies the file in which the job description is stored (that is, the job source).

The following RUNJOB command could be used to activate the job description in the previous section:

```
RUNJOB :ACCOUNTS,JOBNO1,JOBDESCRIP
```

ENDING A JOB PREMATURELY

A user can give instructions causing the job to be terminated at any time, if, for example, a program halts because of an execution error. The following sections show how certain commands can be included in the job description to identify events such as program failures, and if necessary, cause the job to be terminated should these events occur. By making use of these commands, the user is thus able to avoid his time budget (see Chapter 1, page 1) being wasted in the event of his job going wrong.

The WHENEVER command

The WHENEVER (WE) command specifies a condition and causes another command to be obeyed if that condition arises at any time whilst the job is running. The format of the WHENEVER command is:

WHENEVER *condition,command*

A list of the different conditions is given in Chapter 12 of *Operating Systems GEORGE 3 and 4.* However, the most common format of this command is:

WHENEVER COMERR,*command*

In this case *command* is obeyed if a *command error* occurs in any of the remaining commands in the job

description. A command error might arise either from an incorrect format of the command in the job description, or might arise because, for example, a file needing to be accessed does not exist.

More than one WHENEVER command may be present in the job description. In this case the range of each WHENEVER command extends only up to the command before the next WHENEVER command.

The WHENEVER command is usually issued in the following format and is normally the first command after JOB:

```
WHENEVER COMERR,ENDJOB
```

This causes the job to be abandoned if a command error occurs at any time whilst the job is being run.

Note: The conditions that may be used with WHENEVER are different from those allowed with IF (described later) and are far fewer in number.

The JOBTIME command

The time taken by a job is known as *jobtime* and is measured in terms of *mill* or *processor* time. Job time consists of:

1 The total run time for all the programs within the job

2 The time required by the system to process the commands in the job description

The maximum time permitted for any job is normally determined by the installation parameter JOBTIME which is set by the installation manager. However, a user may himself set a time limit for his job if, for example, he knows that his job should not exceed a time which is less than that specified by the JOBTIME parameter. Conversely, he may wish to run a job requiring more time than that permitted by the installation parameter, and in both these cases he is able to do this by giving a JOBTIME (JT) command:

JOBTIME *integer* { SECS / MINS }

SECS is assumed if neither SECS nor MINS is specified. Thus the command:

```
JOBTIME 40
```

would cause the job to be abandoned if the jobtime exceeded 40 seconds.

A time limit for each program within the job can also be set by giving a TIME command before the corresponding program is entered. The format of this command is described in Chapter 6 in the section *Controlling the program run*, page 30.

The GOTO command

The GOTO (GO) command causes a branch to be made to a labelled command. Its format is:

GOTO *label*

where *label* may be of any length and must start with a digit. GOTO is normally used in conjunction with IF, described in the next section.

The IF command

The IF command provides some of the most important facilities of the GEORGE command language. It is used to specify conditions that might arise and to specify what action is to be taken in the event of these conditions arising.

For example, IF can be used, in conjunction with GOTO, to cause a branch to be made to another part of the job description in the event of a program failure. The IF command may have either of the following formats:

1 IF *condition,command*

2 IF *condition,*(*command*$_1$)ELSE(*command*$_2$)

The following is a straightforward example of the use of IF:

```
IF FAILED,ENDJOB
```

This command would be issued after the commands used to run the program, and would cause the job to be abandoned in the event of a program failure. The command would be ignored in the event of a successful program run, and the commands following IF would then be executed.

If the user wished to carry out one or more operations before the job was abandoned, he could give the command:

```
IF FAILED,GOTO 2
```

Here, the command identified by the label 2, (and all subsequent commands) would be executed in the event of a program failure. These commands might specify, for example, that the files holding the object program and input data be erased and the job then ended.

The next example illustrates the use of IF when used with the second format:

```
IF HALTED(OK),(DISPLAY 0,JOB SUCCESSFUL) ELSE (DISPLAY 0,- JOB HAS FAILED)
```

Here, the condition to be tested for is HALTED(OK), indicating a successful program run. If the run is successful, the DISPLAY (DP) command is used to send the message JOB SUCCESSFUL to the job's monitoring file. If the run is unsuccessful, the message JOB HAS FAILED is sent to the monitoring file.

Note: The DISPLAY command may be followed by another *routing parameter* besides 0, if, for example, the user wishes to send a message to the operator's console. A full specification of the DISPLAY command is given in Chapter 12 of *Operating Systems GEORGE 3 and 4* (TP4345).

Some further examples of the use of IF to test for program events (such as FAILED,HALTED and DELETED) are given in Chapter 5, page 29. In Chapter 8, the use of the IF command, to test for the presence or absence of a parameter, is described, page 46.

It is also possible to test for compound conditions, and thus replace a number of separate IF commands, by using AND,OR,NOT. Some possible formats of IF used in this way are as follows:

IF *condition*$_1$ AND *condition*$_2$,*command*
IF *condition*$_1$ OR *condition*$_2$,*command*
IF NOT *condition*,*command*
IF *condition*$_1$ AND *condition*$_2$ OR *condition*$_3$,*command*

In the last example *command* would be executed only if *condition*$_3$ was true or if both *condition*$_1$ and *condition*$_2$ were true.

A complete list of possible conditions is given in Chapter 12 of *Operating Systems GEORGE 3 and 4.*

Note: The command specifying the action to be taken may itself be another IF command.

Chapter 5 Running programs

This chapter describes how an object program is loaded and run by issuing appropriate GEORGE commands. A program requiring limited input and output facilities may also be loaded and run by issuing a single command, RUN. The format of this command is described in the last section of the chapter.

In this chapter it is assumed that the program to be run has already been fully developed during previous test runs. Compilation and facilities designed to assist program development are described in Chapter 6, page 31.

LOADING A PROGRAM

A program will normally be loaded from a filestore file. Sometimes, however, a user may need to load a program from a magnetic tape or an exofile, that is, a disc file outside the filestore. (The use of magnetic tapes and exofiles is described in Chapter 7, page 37.) The following sections show how loading is carried out in each case.

Loading a program from a filestore file

A program is loaded from a filestore file by a LOAD (LD) command. This has the format:

LOAD *file description,* SIZE *number*

The first parameter is the name of the file from which the program is to be loaded. The second parameter is optional and specifies the size in words of the program being loaded.

The SIZE *number* parameter may be used to override the amount specified in the program's request slip. For example, if the program to be loaded from the file MYPROG has a size requirement of 7000 words specified in its request slip, but the user knows that only 6000 words will be required on this run, he can issue the command:

LOAD MYPROG, SIZE 6000

Loading a program from a magnetic tape or exofile

When a program is to be loaded from a on-line magnetic tape or exofile, the FIND command must be used. In this case, one of the following formats will be used as appropriate:

1 FIND *program name,*MT*,magnetic tape description,*SIZE *number*

2 FIND *program name,*DA*,exofile description,*SIZE *number*

The first parameter, *program name,* has the standard format of # followed by four characters. The format of the parameters *magnetic tape description* and *exofile description* are given in Chapter 7, page 37. The SIZE *number* parameter is optional and is used in the same way as when used with LOAD.

The next example shows how the program #AL01 would be loaded from the on-line magnetic tape identified by the name PROGRAMTAPE:

FIND #AL01,MT,PROGRAMTAPE

INITIATING A PROGRAM RUN

Once a program has been loaded and the necessary program corrections made a user can initiate the run by issuing an ENTER (EN) command. This has the format:

ENTER *number*

where *number* must be in the range 0 to 9.

If a program is to be entered at word 20, (the normal point of entry for a program) *number* will be 0 and can be omitted. The program will be entered at word 21 if 1 is specified, word 22 if 2 is specified, and so on.

Controlling a program run

A user may wish to set a maximum time limit on the program run if he knows from previous test runs how long the program should run. This will prevent his budgets being wasted should unforseen errors such as endless loops occur. The time limit is specified by a TIME (TI) command having the format:

TIME *milltime*

where *milltime* is a number giving the maximum permitted run time, in seconds, for the program run. *milltime* may be followed by MINS if the run time is measured in minutes. Thus the following commands perform identical functions:

```
TIME 120
TIME 2MINS
```

An IF command should be issued after ENTER to test if the program has completed its run and to specify the appropriate action to be taken in the event of a program failure. The format of IF when used in this way is given in Chapter 6 in the section *Program events*, page 32.

RUNNING PROGRAMS REQUIRING LIMITED INPUT/OUTPUT FACILITIES

A user may want to run a program in which the input data is held in a basic file, and the output data is to be listed on a line printer. In this case the program can be loaded and run (with the appropriate peripheral connections made automatically) by issuing a single macro command, RUN.

The format of RUN depends upon the facilities required. For example, the user is able to specify that peripherals are to be connected either directly to the program's channels (that is on-line) or used off-line.

A commonly used format of RUN for running programs off-line is:

RUN #*file description*, $\left\{\begin{matrix}*CR\\*TR\end{matrix}\right\}$ *file description*,TIME *milltime*

#*file description* defines the file from which the program is to be loaded. *CR *file description* or *TR *file description* defines a file that is to be connected to the program's card reader input channel 0, or tape reader input channel 0, as appropriate. The TIME *milltime* parameter sets a maximum time limit on the program run and has the same format as the TIME command described in the previous section. If this parameter is omitted the program will be allowed 5 seconds of mill time.

The following is an example of the use of RUN in a job using card input and line printer output.

```
JOB :ACCOUNTS,PROGRUN
WHENEVER COMERR,ENDJOB
RUN #PROGFILE,*CR INPUTDATA,TIME 20
ENDJOB
****
```

The RUN command would load the program held in PROGFILE and connect the file INPUTDATA to card reader input channel 0 of the program. The program's line printer output channel 0 would be connected to the job's monitoring file. The program would then be entered at word 20 (its normal entry point) and would be permitted to run for a maximum of 20 seconds. Any program generated data would be sent to the monitoring file. This data, together with the rest of the monitoring files contents, would be listed on the line printer after the ENDJOB command has been executed.

It is also possible to specify that program output should be sent to a file other than the monitoring file by including the parameter *LP *file description* in RUN. This file will not be listed or erased unless LISTFILE and ERASE commands are issued after RUN.

An ENTRY *number* parameter may be specified if the program is to be entered at a word other than word 20 (its normal entry point). The *number* parameter has the same format as the *number* parameter in the ENTER command described earlier.

Other parameters may also be specified if for example it is desired to specify a particular action to be taken in the event of a program failure. These additional parameters are described under RUN in Chapter 13 of *Operating Systems GEORGE 3 and 4* (TP4345).

Note: The parameters used with RUN can appear in any order.

Chapter 6 Compiling and developing programs

The first part of this chapter shows how a source program is compiled to produce an object program. The rest of the chapter describes some facilities available to the user to assist him in testing and developing his programs.

COMPILING PROGRAMS

The source program to be compiled is punched on cards or paper tape and stored in a filestore file using the INPUT command. The format of INPUT is described in the section *Transferring information to the filestore,* page 14.

The program is compiled by issuing a compilation macro command in the job description. The command used depends upon the language in which the program is written and the format of this command is given in the appropriate compiler manual. For programs written in Algol, FORTRAN or COBOL, this information is given in the ICL 1900 Series manuals:

Algol: GEORGE 3 and 4 Compilers (Edition 1, TP4302)
FORTRAN: GEORGE 3 and 4 Compilers (Edition 1, TP4303)
COBOL Compilers (Edition 1, TP4236)

Many compilation macros will also compile programs held on magnetic tape or disc.

Note: The installation may prefer to use its own compilation macros. The user should therefore consult the installation literature before attempting to compile a program.

Saving programs

If the source program has compiled successfully, the object program produced will normally be loaded into core. At this point the user can issue a SAVE (SV) command to retain a copy of the object program in a file. The program can then be loaded from this file in subsequent jobs. The format of the SAVE command is:

SAVE *file description,device type*

file description may be the name of a new file to be created or of an existing file to which the user has WRITE access. The *device type* is optional and may be either *DA (specifying a disc file) or *CP (specifying a GRAPHIC file). If this parameter is omitted *DA is assumed.

DEVELOPING PROGRAMS

Many programs will initially contain errors of a logical nature. These errors will remain undetected during the compiling process and will not be discovered until an attempt is made to run the object program.

The type of execution error occurring will be indicated by either a standard error message or error message number depending upon the programming language used. The message or message number will be sent to the job's monitoring file which is printed after the job has finished.

The possible causes of each type of execution error are given in the appropriate compiler manuals. For Algol, FORTRAN and COBOL programs, this information is given in

Algol: GEORGE 3 and 4 Compilers (TP4302) Appendix 4
FORTRAN: GEORGE 3 and 4 Compilers (TP4303) Appendix 3
COBOL Compilers (TP4236) Chapter 7

Normally the source program will then be corrected, recompiled and the resulting object program re-run. However, if the program is written in PLAN, it is possible to amend the object program itself before re-running it, thus avoiding the need for re-compilation. This and other facilities normally used by PLAN programmers are carried out by using ALTER,PRINT and ON/OFF commands. The format and use of these commands are described in the last part of this chapter.

It is advisable to set a time limit on the program run initially so that the user's budgets will not be wasted in the event of the program looping, for example. The time allowed for the program run is specified by a TIME command issued before the program is ENTERed. The format of TIME was described in the section *Controlling the program run*, page 30.

Program events

A user may wish to perform a certain set of operations depending on the result of a program run. For example he may wish to abandon the job in the event of a program failure. This section describes how conditions such as program failures can be identified within the job, and how the appropriate action can be taken in the event of these conditions arising.

When a program is entered, control passes from the job description to the program. Control will not return to the job description until the program stops running.

A program 'stop' is known as a *program event.* The program event will be of either the HALTED, DELETED or FAILED category unless a MONITOR command was issued before the program was entered. In this case the program may stop as a result of a MONITOR program event. The use of the MONITOR command is described in a subsequent section.

HALTED AND DELETED PROGRAM EVENTS

Program events of the HALTED or DELETED category will occur when an instruction such as PAUSE (in FORTRAN) causes the program to halt temporarily, or the final END instruction in an Algol program is encountered. A HALTED event will also occur as the result of an execution error in a FORTRAN or Algol program. The ways in which program events of the HALTED or DELETED category arise in PLAN, COBOL FORTRAN and Algol programs are indicated in Table 2, below.

PLAN	*COBOL*	*FORTRAN*	*Algol*	*Associated program event*
SUSWT	STOP	PAUSE	PAUSE	HALTED
SUSTY		Execution error	Execution error	HALTED
DEL		STOP	END (final)	DELETED
DELTY				DELETED

Table 2 How program events of the HALTED and DELETED categories arise in PLAN, COBOL, FORTRAN and Algol programs

If the program event is of the HALTED category, the program remains in core (and can be restarted) or in a form suitable to be put back into core. In this case a *core image* of the program is said to exist. If the program event is of the DELETED category, the core image is destroyed.

An IF command can be issued after the program is ENTERed to specify what action is to be taken in the event of a program event occurring. For example, an event of the HALTED category would be identified by a command such as:

```
IF HALTED,GOTO 20
```

Here, a branch will be made to the command labelled 20 if a HALTED event occurs. In this situation a PLAN programmer for example might wish to issue the command:

```
20 PRINT ,ALL
```

This command would cause a print-out of the current core image to be sent to the monitoring file. The print-out could then be examined at the end of the job when the monitoring file was printed. (The PRINT command is described in detail on page 35.)

An event of the DELETED category would be identified in a similar way. In the next example the job is ended if a DELETED program event occurs:

```
IF DELETED,ENDJOB
```

Note: If a user wishes to SAVE the program or obtain a print-out of the core image existing before the program is deleted, he can do this by making use of the MONITOR command (described in the section *Monitor program events*, page 34).

When a program event occurs, an associated *program event message* is output by the system and sent to the monitoring file. For HALTED and DELETED events this message will consist of the text associated with the instruction causing the event. In the case of an execution error the message will be one of a range of standard error messages. These error messages depend upon the compiler being used and a list is given in the appropriate compiler manual.

A particular program event can be identified from the associated program event message. For example, a PAUSE instruction such as:

```
PAUSE 1HALT
```

will cause a HALTED program event and the program event message output will be 1HALT. In the command:

```
IF HALTED(1HALT),GOTO 10
```

the GOTO command would be executed when the above PAUSE instruction was executed.

The parentheses need not contain the whole text of the message. However, sufficient leading characters to make the message uniquely identifiable (from other expected messages) should be included.

FAILED PROGRAM EVENTS

A program can fail for a number of different reasons. For example, a failure will occur if the time allowed for a program run has been used or because a program has attempted to read beyond the end of the file.

A command of the form:

```
IF FAILED,command
```

can be used to identify events of the FAILED category. Specific failures can be identified in the same way as HALTED and DELETED events, that is, by including sufficient leading characters from the associated program event message in the condition of the IF command. For example, the message TIME UP is output when the time allowed for a program run has been used. In this case

```
IF FAILED (TIM),command
```

will specify a command to be executed should this condition arise.

A list of events of the FAILED category (together with their associated messages) is given in the section *Program events* in Chapter 14 of *Operating Systems GEORGE 3 and 4* (TP4345). The state of the core image existing after each type of FAILED program event is also described.

EXAMPLE

The following is an example of a job description incorporating IF HALTED, IF DELETED and IF FAILED commands:

```
      JOB :ACCOUNTS,JOB26
      WHENEVER COMERR,ENDJOB
      LOAD PROG1
      ASSIGN *CR0,INDATA1
      ASSIGN *LP0,PRINTOUT
      ASSIGN *MT0,MYTAPE(WRITE)
      ENTER
1     IF HALTED (DATA),GOTO 2
      IF DELETED (FINISH),GOTO 3
      IF FAILED,ENDJOB RETAIN (MONF1)
      ENDJOB
2     ASSIGN *TR0,INDATA2
      RESUME
      GOTO 1
3     LOAD PROG2
      ASSIGN *MT0,MYTAPE
      ASSIGN *LP0,RESULTS
      LISTFILE RESULTS,*LP
      ENTER
      IF HALTED (OK) OR HALTED (END),ENDJOB NONE
      IF FAILED,ENDJOB RETAIN (MONF2)
          ENDJOB
      ****
```

In this job a program is loaded from the file PROG1. Data in card format is read from INDATA1 until a HALTED program event (identified by the message DATA) occurs. The file INDATA2 is then connected to tape reader input channel 0 and the run resumed at the next instruction by issuing the command RESUME (RM).

If the program fails, the job is abandoned, the monitoring file is printed and a copy of the monitoring file information is retained in the file MONF1. In the event of a successful run, the program is deleted and a branch made to the command labelled 3.

The command identified by the label 3 loads a second program from the file PROG2. The file MYTAPE (which contains data generated during the previous program run) is connected to the program's magnetic tape input channel 0. Data generated during this program run is sent to the file RESULTS which will be listed on the line printer.

If this program run ends successfully the job is ended without the monitoring file being printed. In the event of a program failure the monitoring file is printed and a copy of the information in the monitoring file retained in the file MONF2.

MONITOR PROGRAM EVENTS

MONITOR program events arise as the result of issuing a MONITOR (MN) command before the program is entered. If the command:

```
MONITOR ON,DELETE
```

is issued before the ENTER command, DEL or DELTY instructions are treated as if they were SUSWT or SUSTY instructions (see Table 1). Thus these instructions will generate events of the HALTED and not the DELETED category. A core image will thus be retained if for example a FORTRAN STOP or the final END in an Algol program is executed.

A PLAN programmer would issue MONITOR ON,DELETE if for example he wished to obtain a print out of the core image existing at the end of his program run. In this case he would give the command:

```
IF MONITOR (DELETE),PRINT ,ALL
```

after the ENTER command.

Certain instructions (for example, DISPLAY in COBOL and DISP, DISTY in PLAN) allow a program to display a message without halting. Under GEORGE, such messages are sent to the monitoring file but do not normally halt the program. If it is necessary to halt the program to allow the JD to take some action when a display occurs, the command:

```
MONITOR ON,DISPLAY
```

can be issued before the ENTER command. The program will then halt each time an instruction causing a display is executed.

The action to be taken when a particular display is output can then be specified by a command of the form:

```
IF MONITOR (DISPLAY) AND DISPLAY (message),command
```

where *command* will be executed if the current program display is *message*.

The monitoring facility may be switched off at any time by using one of the following commands as appropriate:

```
MONITOR OFF,DELETE
MONITOR OFF,DISPLAY
```

An illustration of the use of MONITOR ON,DISPLAY is given in the following job description:

```
      JOB :ACCOUNTS,PROG22
      WHENEVER COMERR,ENDJOB
      LOAD PROG22
      ASSIGN *CR1,INDATA1
      ASSIGN *LP1,PRINTFILE1
      LISTFILE PRINTFILE1,*LP
      MONITOR ON,DISPLAY
1     ENTER
      IF MONITOR (DISPLAY) AND DISPLAY (LP),GOTO 2
      IF MONITOR (DISPLAY) AND DISPLAY (TR),GOTO 3
      IF HALTED (OK),(ENDJOB NON) ELSE (ENDJOB ALL)
```

```
2   ASSIGN *LP2,PRINTFILE2
    LISTFILE PRINTFILE2,*LP
    RESUME
    GOTO 1
3   ASSIGN TR1,INDATA2
    RESUME
    GOTO 2
    ****
```

In the above job, files are ASSIGNed to line printer output channel 2 and tape reader input channel 1 at different points of the program run.

Printing out areas of core

A printout of part or all of the core occupied by a program can be obtained by issuing a PRINT (PT) command. This command has the format:

PRINT $region_1$,$region_2$,...,$region_n$

where *region* may either be

1 *n* where *n* is the number of the location to be printed

2 (*n*,*m*) where *n* is the first location and *m* the last location of a region of core to be printed out

3 *n*(*m*) where *n* is the first location of a region of core, and *m* indicates how many locations are to be printed

Thus the following commands would all have the effect of sending the contents of locations 20, 21 and 22 to the monitoring file:

```
PRINT 20,21,22
PRINT (20,22)
PRINT 20(3)
```

The following formats for PRINT may also be used:

PRINT *file description*,ALL
PRINT *file description*, REGION ($region_1$,$region_2$,...$region_n$)

file description is the name of a file to which the contents of the locations specified are to be sent. If the parameter ALL is given, the whole of the area of core occupied by the program (that is, the current *core image*) is sent to the file, as in the following example

```
PRINT PRINTFILE,ALL
```

The *file description* parameter may be null as in the next example. In this case the contents of the specified locations are sent to the monitoring file.

```
PRINT ,REGION((20,22))
```

The printout obtained as a result of using PRINT has the standard format A C S O I where:

A is the address of the location in decimal

C is the contents in character form

S is the contents as a signed decimal number

O is the contents in octal

I is the contents in instruction format (F X M/N)

Altering the contents of a location

After examining the core printout, a user may decide to alter the contents of certain core locations and attempt to run the program again. The alterations can be carried out by re-loading the program and issuing one or more ALTER (AL) commands before the program is ENTERed.

The ALTER command has the format:

ALTER *number,number*

The first parameter gives the address of a location to be altered. The second gives the new contents of the location as a number. Thus the command:

ALTER 100,0

would cause the contents of word 100 to be set to zero.

The following alternative format for ALTER may also be used:

ALTER *number,instruction*

In this case, *instruction* may be either a PLAN mnemonic or a function code. Thus the following commands would set the contents of word 90 to LDX 0 1, and set the contents of word 88 to the contents of word 87:

ALTER 90,LDX 0 1
ALTER 88, [87]

Some notes on instruction formats are given under ALTER in Chapter 12 of *Operating System GEORGE 3 and 4* (TP 4345).

The switchword facility

Word 30 of a program is known as the *switchword.* The switchword can be set to any value by issuing the commands ON and OFF and may be tested during or at the end of a program run. The format of the ON and OFF commands is:

ON *number list*
OFF *number list*

Each number in the number list specifies one of the switchbits 0 to 23. For example the command:

ON 0,2,4,6,8,10

will set bits 0,2,4,6,8,10 of word 30 to one. The command:

OFF 0

will set bit 0 of word 30 to zero.

The switchword is tested by the commands:

IF ON *number list,command*
IF OFF *number list,command*

In the first command the condition is satisfied (and thus *command* will be executed) if the bits specified in the number list all have the value 1. Similarly, in the second command the condition will be satisfied if the bits specified in the number list all have the value 0.

PART 3 ON-LINING AND MACROS

Chapter 7 Use of magnetic media and on-line peripherals

This chapter describes how basic peripherals, magnetic tapes and exofiles, (that is, non-filestore files held on magnetic discs), may be directly connected to a program's input and output channels. This process is known as *on-lining.* On-lining is not recommended for basic peripherals as it is less efficient than off-lining (described in Part 2) unless there are good reasons why a user should have direct control over these peripherals.

Exotic peripherals (such as graph plotters) and communications peripherals will normally have to be used on-line. A full description of the process of on-lining for these peripherals is given in Chapter 5 of *Operating Systems GEORGE 3 and 4* (TP 4345).

If peripherals are to be used on-line, a NEEDS command should be issued at the beginning of the JD. This command indicates to the system what peripherals will be required and the job will not be fully started until the peripherals requested are available. The format of the NEEDS command is:

NEEDS *number peripheral type,...,number peripheral type$_n$*

where *peripheral type* is not preceded by an asterisk. Thus the command:

NEEDS 3LP,5MT,2CR

would indicate that a maximum of three line printers, five magnetic tape units and two card readers will be required at any one time during the running of the job.

BASIC PERIPHERALS

A basic peripheral (defined in Chapter 3) is put on-line to a program by giving an ONLINE command. However, if the peripheral is to be used for input, the input data held on that peripheral must first be prefaced with a DOCUMENT (DM or DOCU) command, having the format:

DOCUMENT *document name*

This command identifies the data following it with a *document name.* Thus, in the following example, a set of data held on cards is identified by the name DATACARDS:

DOCUMENT DATACARDS
first line of data
.
.
.
last line of data
****

When used to connect a basic peripheral to a program, the ONLINE (OL) command has the format:

ONLINE *basic peripheral name, document name*

In the case of an input peripheral, *document name* is the name identifying the input data. In the case of an output peripheral, the ONLINE command finds a peripheral of the required type (for example, card punch or line printer) and heads the output produced with the specified *document name.*

If, for a basic input peripheral, *document name* is omitted, the data will be read from the job source in exactly the same way as if an ASSIGN command (specifying a *basic peripheral name* only) had been used. (See the section, *Assigning the job source*, page 16.) If *document name* is omitted for a basic output peripheral, the output produced by the program will be written to the monitoring file, once again in the same way as if an ASSIGN command (specifying a *basic peripheral name* only) had been used.

The next example illustrates the use of ONLINE when used to connect basic input and output peripherals to a program. In this example, the program PROG1 is loaded and entered and reads input data identified by the name DATACARDS, (see previous example). Copies of cards containing errors are listed on the card punch and the

corresponding error messages are sent to the job's monitoring file (identified here as *LP0). The results obtained by the program are headed with the name PROG1RESULTS and listed on the line printer.

```
JOB :PLANNING,PROG1JOB
WHENEVER COMMERR,ENDJOB
LOAD PROG1
ONLINE *CR0,DATACARDS
ONLINE *CP0,ERRORCARDS
ONLINE *LP0
ONLINE *LP1,PROG1RESULTS
ENTER
ENDJOB
****
```

The property system

A user may wish to specify that an output peripheral must have certain *properties.* A property is any characteristic that distinguishes the peripheral from another of the same type, for example the number of print positions on a line printer or the type of stationery loaded on the line printer. The installation will have a list of properties and the user is able to specify the required property by using a PROPERTY qualifier in his ONLINE (or LISTFILE) command. The property system is fully described in Chapter 5 of *Operating Systems GEORGE 3 and 4* (TP4345).

MAGNETIC TAPES

Magnetic tapes are of two kinds: *secure* and *insecure.* Secure tapes are those known to GEORGE and a list of these tapes is held in the file :SYSTEM.SERIAL. A user can acquire one or more secure tapes for his own use, details of these tapes being recorded, together with details of files owned, in his directory. Insecure tapes, that is, tapes outside the system, may also be used, either by connecting them directly to programs or by first converting them into secure tapes.

The advantage of using secure tapes instead of insecure tapes is that the former have the same security arrangements as filestore files. These arrangements were explained in Chapter 3 under *Accessing files*, page 18. The TRAPSTOP, TRAPGO and TRAPCHECK commands described for files have exactly the same format when used for magnetic tapes, except that the *file description* parameter is replaced by a *magnetic tape description parameter.* The format of this parameter is given in a subsequent section.

Secure tapes

Secure tapes which have been acquired by a user can either be *owned tapes* or *worktapes:*

1 An owned tape is a tape that has been acquired for permanent use from the *pool.* (The pool consists of a set of tapes known as *pool tapes* which are available to any user and designed for permanent ownership.) A user must have sufficient SPACEMT budget (see the section *Budgets*, page 4) before tapes can be acquired from the pool. An owned tape can be returned to the pool when it is no longer required.

2 A worktape is a tape which can be acquired for the duration of a job only. Worktapes are automatically returned to the *worktape store* at the end of the job in which they were acquired and are then available to other users. The SPACEMT budget does not apply to worktapes.

Referring to magnetic tapes

A magnetic tape is referred to in a command by a *magnetic tape description* parameter. This parameter may specify either the serial number of the tape, or the magnetic tape name, or both. For example, a magnetic tape with the name JOHNMAGTAPE in its header label, and having the serial number 1234, can be referred to in any of the following ways:

```
(1234)
JOHNMAGTAPE
(1234,JOHNMAGTAPE)
```

The magnetic tape name has the format of a *local name* and thus consists of up to twelve letters, digits, spaces or hyphens beginning with a letter. Other optional details, such as a generation number, may also be included, as in the following example:

```
JOHNMAGTAPE(1)
```

The generation number facility is described in Chapter 3 under the heading *Referring to Files*, page 12. A full description of this and other optional qualifiers is given in Chapter 4 of *Operating Systems GEORGE 3 and 4* (TP4345).

An owned tape belonging to another user can be referred to by an absolute name. For example:

:ACCOUNTS.BILLMAGTAPE

Note: The local name of an owned tape must not coincide with the local name of a file entered in the same directory.

Using an owned tape

An owned tape can be acquired from the pool by a GET (GE) command. This has the format:

GET *magnetic tape name* (*MT)

The *magnetic tape name* must be followed by the qualifier (*MT). Thus, the command:

GET MYTAPE (*MT)

would cause a tape from the pool to be brought under the user's ownership and the name MYTAPE written in its header label.

A user is initially allowed only READ and WRITE access to a tape acquired by a GET command. These traps may be altered by using TRAPGO and TRAPSTOP commands as described in Chapter 3 under *Accessing files*, page 18.

An owned magnetic tape is connected to a program by an ONLINE (OL) command, having the format:

ONLINE *magnetic tape peripheral name, magnetic tape description*

Thus, the command:

ONLINE *MT0,DATATAPE

would connect the magnetic tape named DATATAPE (which has previously been written to by another program) to input channel 0 of the program.

The magnetic tape peripheral name must be qualified by WRITE when an owned magnetic tape is connected to a program's output channel. Thus, if the user wished to connect MYTAPE (acquired by GET in a previous example) to output channel 1, he would give the command:

ONLINE *MT1(WRITE),MYTAPE

If a user no longer requires the use of a magnetic tape he can return it to the pool by giving a RETURN (RT) command:

RETURN *magnetic tape description* (*MT)

Note: An owned magnetic tape can also be acquired from the pool and connected to a program either by using an unanticipated open mode #400 PERI instruction in that program, or by issuing a GETONLINE (GL) command. The GETONLINE command has the format:

GETONLINE *magnetic tape peripheral name, magnetic tape name*

and combines the functions of the GET and ONLINE commands.

Using a worktape

A worktape can be acquired and connected to a program's channel either by issuing an ONLINE command or by using an unanticipated mode #600 PERI instruction in the program. The format of the ONLINE (OL) command is:

ONLINE *magnetic tape peripheral name* (WRITE)

Thus a worktape would be connected to a program's output channel 1 by the command:

ONLINE *MT1 (WRITE)

A *named worktape* must be used if the worktape is to be passed between a number of programs within the job. The format of a worktape name is:

!*name*

where *name* consists of up to twelve letters, digits, hyphens and spaces, beginning with a letter.

A named worktape is acquired and connected to a program by GET and ONLINE commands (or by a GETONLINE command) using the same format as that used for owned tapes, (see previous section). This is illustrated in the next example:

```
GET !MYWORKTAPE (*MT)
ONLINE *MT1 (WRITE),!MYWORKTAPE
```

All worktapes are automatically returned to the worktape store at the end of the job, and are then available for use by other users.

Using an insecure tape

Insecure tapes are connected to a program in the same way as owned tapes. However, it is advisable to specify the tape serial number to ensure that the correct tape is loaded. (Also, as these tapes are unknown to GEORGE, the relative generation number facility cannot be used). The next example shows two possible ways of connecting an insecure tape to a program:

```
ONLINE *MT0,(1234,MYTAPE)
ONLINE *MT0,(1234)
```

A user can change the status of one or more tapes from insecure to secure by giving a NEW (NE) command:

NEW (*tsn*) (*MT),...,(*tsn*$_n$) (*MT)

where *tsn* is the magnetic tape serial number. Thus, the command:

```
NEW (56213) (*MT)
```

would cause the magnetic tape with serial number 56213 to belong to the user.

Note: The magnetic tape name may optionally be specified in addition to the tape serial number in a NEW command.

A user is initially allowed only READ access to a tape acquired by a NEW command. To allow himself or other users WRITE or any other permitted mode of access, the user can issue a TRAPGO command (see page 38). The TRAPSTOP command is used in a similar way to withdraw rights of access.

A tape acquired by a NEW command is connected to a program in the same way as a tape acquired by a GET command.

A user can convert a secure tape back to an insecure tape by giving a DEAD (DD) command:

DEAD (*tsn*$_1$) (*MT),...,(*tsn*$_n$) (*MT)

Note: A tape acquired by a NEW command (in which the magnetic tape name was omitted) cannot be referred to in a command by tape name alone until it has been loaded at least once.

EXOFILES

Exofiles are files on disc which are held outside the filestore. They can be created dynamically with a PERI mode #1200 or alternatively by using the data file allocator (#XPJC).

Referring to exofiles

An exofile is referred to in a command by an *exofile description* parameter, having the format:

(*serial number,exofile name*)

The *serial number* is the number of the disc cartridge on which the file is held. This number is written in octal. The *exofile name* has the format of a *general local name* (see Chapter 3, page 11).

Thus the following exofile description parameter would refer to an exofile named XFILE held on the disc cartridge #100:

```
(100,XFILE)
```

Connecting exofiles to programs

An exofile can be connected to a program by an ONLINE (OL) command having the format:

ONLINE *direct access peripheral name,exofile description*

The peripheral type specified in the *direct access peripheral name* is *DA. (In job descriptions written for running under earlier versions of GEORGE, the peripheral type may have been specified as either *ED or *FD. These are still acceptable and these job descriptions will thus run without alteration under the current mark of GEORGE). The direct access peripheral name must be qualified by WRITE (as in the example below) if the exofile is to be written to by a program.

ONLINE *DA0(WRITE),(200,MYXFILE)

The direct access peripheral name may be qualified instead of WRITE by OVERLAY (if the exofile is to be opened in mode #100) or by OFFSET (if the exofile is to be opened in mode #400).

Chapter 8 Writing macros

When a large number of jobs are to be run, and the same set of operations occurs as a part of each job, the process of writing the associated job descriptions can be considerably simplified by making use of the macro facility. Under this facility, a *macro* (which consists of a set of commands held in a filestore file) can be created to perform the set of operations common to each job. The set of operations defined by the macro can then be performed in any job by issuing a single *macro command* in the associated JD. When a macro command is issued in a JD, the corresponding macro (which, being held separately in a filestore file, is independent of the JD) is *called* from the JD. The commands in the macro are then executed as part of the JD from which it was called. Issuing a macro command is thus equivalent to issuing a set of commands.

To illustrate these ideas consider the following set of commands which frequently occur in JDs where a basic file is used for program output:

```
CREATE
ASSIGN *LP0,!
LISTFILE !,*LP
ERASE !
```

Instead of issuing this set of commands in each JD, a macro incorporating these commands could be written and INPUT to a filestore file. If the name chosen for the macro was OUTPUTOPS, the above set of commands could be executed in any JD by issuing the macro command OUTPUTOPS in that JD.

In the example above, the parameters supplied with each command remain the same each time the command is issued. Normally however, although a set of commands of the same type will occur in different JDs, some or all of the parameters supplied with each command will be different each time the command is issued. For example, consider the two sets of commands below:

```
LOAD PROG1
ASSIGN *MT0,PROG1DATA
ASSIGN *MT1,PROG1RESULTS
ENTER

LOAD PROG2
ASSIGN *MT0,PROG2DATA
ASSIGN *MT1,PROG2RESULTS
ENTER
```

A set of commands of this type would be issued to run a program in which magnetic tape files are used for input and output. The filename parameters will therefore normally be different each time a set of commands of this type is issued. Each set of commands can however be replaced in the JD by a single macro command if a macro is written using the technique of parameter substitution. For example if the macro was named RUNPROG, the first set of commands could be executed in a JD by issuing the command

```
RUNPROG PROG1,PROG1DATA,PROG1RESULTS
```

The parameters PROG1,PROG1DATA and PROG1RESULTS following the macro command verb RUNPROG would be substituted into the appropriate LOAD and ASSIGN commands in the macro when the macro is called. Similarly, the second set of commands given above would be executed by issuing the command

```
RUNPROG PROG2,PROG2DATA,PROG2RESULTS
```

The set of parameters following the macro command verb RUNPROG are known as the *parameter list*.

In the macro RUNPROG, a *parameter identifier* would be present in each of the LOAD and ASSIGN commands in place of the filename parameter. When the macro is called, each parameter identifier would be replaced by the appropriate parameters specified in the parameter list, before the command containing the identifier was executed.

This chapter describes how users may write and use their own macros. The last part of this chapter (page 49) shows how a job description can itself be submitted as a macro.

TYPES OF MACRO

Besides macros written by himself, a user can make use of:

1 Macros written by ICL

2 Macros written by the installation

ICL macros are designed to run pieces of ICL standard software, such as program compilers. A list of these macros is held in the directory file :MACROS.

Installation written macros are designed to meet the needs of that particular installation. For example, a research establishment will usually have a number of macros designed to run scientific software.

INPUTTING AND CALLING A MACRO

The commands of a macro are punched on cards or paper tape in the same was as commands in a job description. The macro is then stored in a filestore file by issuing an INPUT command. A non-standard terminator (indicated by T in the INPUT command) must be used. In the following example the terminator ???? is used:

```
INPUT :ACCOUNTS,RUNPROG,T????
first command of macro
.
.
.
last command of macro
????
```

The name of the macro is the name of the file in which it is held. The macro is called from the job description by issuing the filename as a command. Thus the above macro would be called by a command of the form:

RUNPROG *parameter list*

where *parameter list* contains the parameters which are to be substituted in the commands in the macro.

TYPES OF PARAMETER SUBSTITUTION

Macros may be written using either of the following techniques:

1 Parameter substitution be keyword

2 Parameter substitution by position

Macros using substitution by keyword are much easier to use and can be made far more flexible than macros in which substitution by position is used. Most system macros use the technique of substitution by keyword. Both techniques are described in the following sections.

WRITING MACROS USING SUBSTITUTION BY KEYWORD

In this method each parameter in the parameter list has the format:

keyword string

(A space need not appear between *keyword* and *string*.) *keyword* identifies the type of parameter being submitted and consists of one or more characters chosen by the user. *string* is a character string (such as a filename) which will be substituted in the macro when the macro is called. For example ENTRY 2 consists of the keyword ENTRY followed by a one-character string.

Each parameter identifier in the macro has the format:

%(keyword)

When the command containing the parameter identifier is to be keyed, the parameter list is searched until a parameter beginning with *keyword* is found. The parameter identifier is then replaced by the string following *keyword*.

The following is an example of a simple macro in which this type of substitution is used:

```
INPUT :ACCOUNTS,RUNPROG,T????
LOAD %(OBJ)
ASSIGN *MT1,%(*MT)
ASSIGN *LP1,%(*LP)
LISTFILE %(*LP),*LP
ENTER
EXIT
????
```

Suppose the following job description was submitted:

```
JOB :ACCOUNTS,PROGRUNJOB
.
RUNPROG OBJ ACCPROG,*MT INDATA,*LP RESULTS
.
****
```

The macro would then be called from this job description by the command RUNPROG. This would be equivalent to including the following commands in the job description:

```
LOAD ACCPROG
ASSIGN *MT1,INDATA
ASSIGN *LP1,RESULTS
LISTFILE RESULTS,*LP
ENTER
```

The parameters following RUNPROG can appear in any order. Thus the command below could also be used to run a program.

```
RUNPROG *LP OUTDATA,OBJ PROG1,*MT PROG1DATA
```

The last command obeyed in a macro should always be an EXIT (EX) command or the last command in a file. When either of these commands is executed, control is returned either to the JD or to another macro if the macro is itself being called from another macro.

A number of tests would normally appear in this macro. The function and format of these tests are described in a subsequent section and an example of a macro incorporating tests is given.

Setting parameters in macros using substitution by keyword

A parameter can be assigned a value within the macro itself, or re-set to a new value in the macro, by using a SETPARAM (SP) command in the macro. The SETPARAM command has the format:

SETPARAM (*keyword*), (*keyword string*)

The first parameter refers to the parameter identifier, *%(keyword)*, which is to be given the value *keyword string*. For example, if the command

```
SETPARAM (ENTRY), (ENTRY 0)
```

was issued before

```
ENTER %(ENTRY)
```

this latter command would be executed as

```
ENTER 0
```

The SETPARAM command provides some of the most important facilities of the GEORGE command language and can be used for example to set up loops and to set default values. A full specification of this command is given in Chapter 12 of *Operating Systems GEORGE 3 and 4* (TP4345).

WRITING MACROS USING SUBSTITUTION BY POSITION

In this method each parameter identifier in the macro is of the form:

%letter

where *letter* is a letter of the alphabet from A to X inclusive. When the macro is called, %A is replaced by the first parameter in the parameter list, %B by the second, %C by the third, and so on.

The following is an example of a simple macro using this technique:

```
INPUT :ACCOUNTS, RUNPROGRAM, T????
LOAD %A
ASSIGN *CR0,%B
ASSIGN *CP0,%C
LISTFILE %C,*CP
ENTER
EXIT
????
```

This macro could be called from the job description by a command such as:

```
RUNPROGRAM PROG33, INDATA, OUTDATA
```

This would be equivalent to including the following commands in the job description:

```
LOAD PROG33
ASSIGN *CR0,INDATA
ASSIGN *CP0,OUTDATA
LISTFILE OUTDATA,*CP
ENTER
```

Setting parameters in macros using substitution by position

A parameter can be set to a particular value in the macro, or given a new value, by a SETPARAM (SP) command. This command has the format:

SETPARAM *letter*,(*string*)

where *letter* refers to the parameter identifier *%letter* which is to be set (or reset) to the value *string*. Thus the commands:

```
SETPARAM G, (INDATA)
ASSIGN *CR0, %(G)
```

would cause the file INDATA to be ASSIGNed to the program.

USING TESTS IN THE MACRO

Tests should appear in the macro to ensure that parameters have been correctly specified in the parameter list. For example, it is possible to check that the correct number of parameters have been given, or, in the case of substitution by keyword, to check that the keyword appearing in the parameter list corresponds to the keyword appearing in the macro.

In the event of an error occurring when a parameter is submitted, the associated test can specify the action to be taken. For example, a DISPLAY command can be used to send an error message to the monitoring file before the job is abandoned.

Besides parameter validation, tests can also be used to:

1 Allow parameters to be set by default in the macro when these parameters are omitted from the parameter list. This enables macros to be made very easy and convenient to use since parameters which are usually the same each time the macro is called may be omitted from the parameter list when this is the case.

2 Enable each macro to be designed for a wide range of applications. For example, a macro can be designed to run programs using input from either basic, magnetic tape or direct access files. In this case, the type of input file used will depend upon the type of parameter submitted in the parameter list.

The format of a test

Tests in the macro are performed by IF commands. Each IF command is used to test for a certain condition and to specify what action is to be taken should that condition arise. The general format of the IF command is:

IF *condition, command*

where *command* will be executed only if *condition* is true. *Condition* may also be preceded by NOT. In this case *command* will be executed only if condition is false. An example of IF used with this format is:

```
IF NOT CORE, GOTO 10
```

The condition CORE will be true if a core image exists, and can be used to check that a program has been loaded successfully. In this example, the GOTO command is executed if no core image exists.

The following format for IF may also be used:

IF *condition*, (*command*$_1$) ELSE (*command*$_2$)

In this case, *command*$_1$ will be executed if *condition* is true, and *command*$_2$ executed if *condition* is false. An example of IF used in this way is:

IF EXISTS (PROGFILE), (RETRIEVE PROGFILE) ELSE (GOTO 20)

Here, the condition EXISTS (PROGFILE) is true if the file PROGFILE specified exists. If the file is in existence, it is retrieved and if not the GOTO command is executed instead. This test is frequently used in macros in the form:

IF EXISTS (*identifier*), (RETRIEVE *identifier*) ELSE (GOTO 20)

where parameter *identifier* will be replaced by a filename when the macro is called.

Compound conditions may also be tested. A compound condition consists of two (or more) conditions linked together by AND or OR. For example, in the following IF command:

IF *condition*$_1$ AND NOT *condition*$_2$, *command*

command will be executed only if both *condition*$_1$ is true and *condition*$_2$ is false.

The main function of the tests in a macro was described in the previous section. The conditions associated with these tests depend upon whether substitution by keyword or substitution by position is being used, and are described in the next two sections.

Conditions tested for in macros using substitution by keyword

The format of a parameter in the parameter list when substitution by keyword is used is:

keyword string

Tests can be performed either on *keyword* or on the character *string* following *keyword*. The list below shows the type of conditions that can be tested.

	Condition	*Condition true only when*
1	PRESENT(*keyword*)	parameter beginning with *keyword* (or consisting of *keyword* alone) is present in parameter list
2	ABSENT(*keyword*)	no parameter in the parameter list is *keyword* or begins with *keyword*
3	STRING(%(*keyword*)) = ()	no parameter in the list begins with *keyword* or a parameter consisting of *keyword* alone appears in the parameter list
4	STRING(%(*keyword*)) = (*string*)	a parameter beginning with *keyword* is followed by *string*
5	STRING(%(*keyword*)) > (*string*)	a parameter beginning with *keyword* is followed either by *string*, or by *string* plus some additional characters
6	STRING(%(*keyword*)) < (*string*)	a parameter consisting of *keyword* alone is present, or a parameter beginning with *keyword* is followed either by *string* or by one or more leading characters of *string*, or no parameter beginning with *keyword* is present

Conditions 2 and 3 are often used in tests when parameters are to be set by default. For example, suppose the following commands appeared in the macro:

IF ABSENT(*LP) OR STRING (%(*LP)) = (), SETPARAM (*LP), (*LP !) ASSIGN *LP1,%(*LP)

In this case, a named file would be ASSIGNed to the program if a parameter in the list consisted of *LP *filename*. If this parameter was omitted from the list, or consisted simply of the keyword *LP, a workfile would be ASSIGNed to the program by the command:

ASSIGN *LP1,!

An example of a macro incorporating these type of tests is given in a subsequent section.

Conditions tested for in macros using substitution by position

The list below gives some conditions which are tested frequently in macros written using substitution by position.

	Condition	*Condition true only when*
1	STRING(*%letter*) = ()	the parameter whose position in list is given by *letter* is absent from parameter list (that is the parameter is null)
2	STRING(*%letter*) = (*string*)	the parameter whose position in list is given by *letter* is equal to *string*
3	STRING(*%letter*) > (*string*)	the parameter whose position in list is given by *letter* is equal to *string* or begins with *string*
4	STRING(*%letter*) < (*string*)	the parameter whose position in list is given by *letter* is equal to *string* or consists of one or more leading characters of *string*, or the parameter is null

The first condition identifier a *null parameter*, that is, a parameter omitted from the list. When any parameter (other than the last) is omitted, the associated comma must still be present. For example, in the previous section the macro was called by the command:

RUNPROGRAM PROG33, INDATA, OUTDATA

The following command would be issued if the first and second parameters were null and the macro had been written incorporating the appropriate tests:

RUNPROGRAM ,,OUTDATA

In the macro, tests of the form:

IF STRING(%A) = (),*command*
IF STRING(%B) = (),*command*

would identify the existence of these null parameters and specify the appropriate action to be taken. These type of tests are used when parameters are set by default.

AN EXAMPLE OF A MACRO USING SUBSTITUTION BY KEYWORD

The macro given in this section will perform the following functions:

1 Load a program from a filestore file

2 (a) Connect a magnetic tape file to magnetic tape input channel 1 if the program is designed to read data in magnetic tape format

or

(b) Connect a magnetic disc file to magnetic disc input channel 1 if the program is designed to read data in disc format

3 (a) Connect a named file to line printer output 1. This file will be listed but will not be erased at the end of the program run

or

(b) Connect a workfile to line printer output channel 1. This file will be listed and erased at the end of the program run

4 Run the program. The program will be entered at word 20 if data is read from a magnetic tape file or entered at word 21 if data is read from a disc file

The following parameters may appear in the parameter list:

OBJ *filename* where *filename* is the name of the file from which the program is to be loaded

*MT *filename* or *DA *filename* as appropriate to identify the file used for input

*LP *filename* where *filename* is the name of a file to which program is to be sent.

If the last parameter is omitted, or the keyword *LP alone specified, a workfile will be used for output.

```
      INPUT :ACCOUNTS,PROGRUN,T????
      WHENEVER COMERR,ENDJOB
      IF STRING(%(OBJ)) = (), GOTO 90
      IF EXISTS(%(OBJ)), (RETRIEVE %(OBJ)) ELSE (GOTO 90)
      IF PRESENT(*MT), GOTO 91
      IF PRESENT(*DA), GOTO 92
      GOTO 90
91    IF STRING(%(*MT)) = (), GOTO 90
      IF EXISTS(%(*MT)), (RETRIEVE %(*MT)) ELSE (GOTO 90)
      LOAD %(OBJ)
      IF NOT CORE, EXIT
      ASSIGN *MT1,%(*MT)
      SETPARAM (ENTRY), (ENTRY 0)
      GOTO 93
92    IF STRING(%(*DA)) = (), GOTO 90
      IF EXISTS(%(*DA)), (RETRIEVE %(*DA)) ELSE (GOTO 90)
      LOAD %(OBJ)
      IF NOT CORE, EXIT
      ASSIGN *DA1,%(*DA)
      SETPARAM (ENTRY), (ENTRY 1)
93    IF PRESENT(*LP) AND NOT STRING(%(*LP)) = (), GOTO 94
      CREATE !
      SETPARAM (*LP), (*LP !)
94    ASSIGN *LP1,%(*LP)
      LISTFILE %(*LP),*LP
      IF STRING(%(*LP)) = (), ERASE !
      ENTER %(ENTRY)
      IF FAILED, DISPLAY 0,ERRORS
      EXIT
90    DISPLAY 0,ERROR IN PARAMETER
      EXIT
      ????
```

THE JOB DESCRIPTION OF A MACRO

Parameter substitution can take place within the job description itself. In this case, the parameter identifiers appearing in the commands of the job description are replaced by *job* parameters specified in the initial JOB command. The job parameters in JOB are enclosed in parentheses and preceded by PARAM. For example, in the following job description:

```
JOB :ACCOUNTS,MYJOB,PARAM(OBJ PROG1,TIM 10,IN INDATA,ENT 0)
WHENEVER COMERR, ENDJOB
LOAD %(OBJ)
ASSIGN *CR1, %(IN)
ASSIGN *LP1, !
LISTFILE !,*LP
ERASE !
TIME %(TIM)
ENTER %(ENT)
ENDJOB
****
```

the commands containing parameter identifiers would be executed as:

```
LOAD PROG1
ASSIGN *CR1, INDATA
TIME 10
ENTER 0
```

Parameter substitution by position may also be used in exactly the same way as used in macros. For example, the above four commands would be executed if the job description was preceded by:

```
JOB :ACCOUNTS,MYJOB,PARAM(PROG1,INDATA,10,0)
```

and contained the commands:

```
LOAD %A
ASSIGN *CR1,%B
TIME %C
ENTER %D
```

A filed job description (see Chapter 4, page 25) can also be run as a macro. In this case the job parameters are specified in the RUNJOB command in exactly the same way as in JOB. For example,if the job description above had been stored in a file by issuing the command:

```
INPUT :ACCOUNTS,JOBDESCRIP
```

the job could be run by issuing the command:

```
RUNJOB :ACCOUNTS,MYJOB,JOBDESCRIP,PARAM(OBJ PROG1,TIM10,IN INDATA,ENT 0)
```

Index

Amendment record

Should any additions or amendments to this Manual be necessary, they will be issued in the form of Amendment Lists.
When such changes have been incorporated, an appropriate entry should be made in the table below and the pink instruction sheet(s) should be filed behind this page.

List number	Item number
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to