Chapter 2 Running jobs

JOB DESCRIPTIONS

As was described in Chapter 1, the operating instructions that normally would be given to a human operator, in order to run the job, are presented instead to GEORGE in the form of a job description. A job description is a series of commands written in the GEORGE command language. The general format of a command is:

label VERB parameters

The label is optional and allows branching within the job description to specific commands. The VERB is present in all commands and the parameters (if any are present) define the action of a verb. For example, one of the parameters to the verb LISTFILE describes the file that is to be listed.

COMMANDS

There are two types of GEORGE command, *built-in commands* and *macro commands.* A built-in command requests GEORGE to carry out a relatively simple function; for example, to load a program the user would issue a LOAD command. As their name implies, built-in commands are implemented by built-in portions of the GEORGE program. Macro commands are typically used to perform more complex functions such as compiling a program. A macro command is a command which expands into one or more commands according to a definition stored in a file (unless the file is empty). The macro command itself is the name of the file which contains the definition of its expansion. When the file name is issued as a command it may be followed by one or more parameters which give values to parts of the definition that have been left undefined. Macro commands may be divided into *system macros* and *user macros.* System macros are one of the most important features of GEORGE. They allow the user to run almost all the standard software produced by ICL for the 1900 Series. Thus as far as the user is concerned the items of software are part of the system, and may be used by issuing a simple command. For example to call in the FORTRAN compiler, the user simply issues the command FORTRAN. This command is a system macro. When GEORGE encounters such a command it finds the appropriate system file and expands the macro into the commands necessary to perform a complex function such as loading the FORTRAN compiler from backing store, entering it, running it and dealing with the object program it produces. As new software becomes available it can be simply integrated into the GEORGE system by the creation of new system macros. System macros may also be created to provide overlapping facilities with those that already exist, for example the QFORTRAN macro provides a subset of the facilities catered for by the FORTRAN macro. Thus it is possible to provide a precise interface to meet the needs of all users of the GEORGE system simply by defining system macros to perform the tasks which are the workload of a specific installation.

User macro commands are defined and run in the same way as system macros, that is a string of built-in commands is held in a file and the commands are obeyed when the name of the file is issued as a command. A user macro is available only to the user who defined it and to other users who have specifically been allowed to access to it. System macros are freely available to all users of the system. Full details of how to write user macros are given later in this Chapter.

Full specifications of built-in commands and system macros which use programs peculiar to the GEORGE environment are given in Chapter 12. Full specifications of other system macros (those which use only standard library programs) are given in Chapter 13.

COMMAND CONTEXTS AND PROCESSOR LEVELS

Command contexts

There are only a few commands which may be issued at any stage of a job. Most commands can only be issued in certain environments; for example, a command to enter a program cannot sensibly be issued until there is a core image (that is until the program has been loaded) and some commands can only be issued from a paper tape or card reader or from the operator's console.

The environments in which commands are issued are known as *contexts.* GEORGE keeps a record of the contexts in which each built-in command may be issued and checks the context whenever a command is issued.

The contexts which GEORGE checks for are as follows:

Context	*Meaning*
USER	Within a MOP or background job
NO USER	Outside a job
OFF-LINE	Within a background job
MOP	From a MOP terminal or in a job that is connected to a MOP terminal.
OPERATOR	From the operator's console or by a macro issued from the operator's console.
READER	From a card or paper tape reader or by a macro issued from a card or paper tape reader
NO CORE IMAGE	No core image exists
CORE IMAGE	A core image exists
PROGRAM	From a program or a macro issued by a program
NOT BREAK-IN	Not during a break-in
BREAK-IN	During a break-in
REMOTE	From an RJE terminal

Commands which are issued before a job has been initiated are issued in the NO USER context. This may be NO USER and MOP, NO USER and READER or NO USER and OPERATOR. Any command that initiates a job changes the context from NO USER to USER. When a LOGIN command is issued from a MOP terminal, the context changes from NO USER and MOP to USER and MOP. If a JOB command is issued from a card reader, the context changes from NO USER and READER to USER and OFF-LINE. Within either of these USER contexts, the context will change if a LOAD command is issued (USER, MOP/OFF-LINE, and CORE IMAGE), and again if a program is entered and a PERI type 60 MODE 1 instruction issues a command (USER, MOP/OFF-LINE, CORE IMAGE and PROGRAM). If in a MOP job, the user breaks in on a program issued command, the context becomes USER, MOP, CORE IMAGE, PROGRAM and BREAK-IN. These context changes apply only to the subsequent commands within the job, for example an INPUT command on another device would still be in NO USER context.

In each command specification in Chapter 11 the section FORBIDDEN CONTEXTS gives details of both contextual restrictions and command processor level restrictions (see below). Contextual restrictions are described in terms of the contexts in which the command must not be issued.

If a command is issued in USER context and the installation parameter CONTEXT is set to A, B or C, GEORGE will also check that the user has the appropriate CONTEXT privilege (see page 156).

Command processor levels

When a job is initiated by a JOB or RUNJOB command, the source of commands to GEORGE changes from a card reader, paper tape reader or MOP terminal to a temporary or permanent job description file (the command JOB is usually issued from a basic peripheral but the commands which follow it are stored in a file and then issued from the file). The change of the source of commands is called a change of command processor level. The JOB command is said to be obeyed at command processor level zero; the commands in the job description file (for example LOAD, ASSIGN, ENTER) are obeyed at a level one greater than this, in other words at command processor level one. Within a given job the command processor level is increased if a command in the job description file issues one or more commands. Thus if the job description contains a macro, the commands within the macro definition file will be obeyed at a command processor level one greater than that at which the macro is issued; if the macro is issued at level one, the commands that constitute the macro will be issued at level two. If one of the commands in a macro is itself a macro, it will cause a further increase in command processor level. Thus by nesting macros within macros a hierarchy of command processor levels can be constructed. Every time the source of the commands changes to a new macro definition file the command processor level is increased by one. When the commands in a macro have been obeyed the current command processor level will be destroyed and control will pass to a level one less than this. Eventually control will pass to level one where the job description file is the command source. When the job is completed this level will be destroyed.

It should be noted that the deletion of a command processor level does not involve the deletion of the command source for that level; when a job is terminated a command processor level is destroyed but the job description file is destroyed only if the job is a once-only job, otherwise the job description file, like any other macro definition file, may be used as a command source for any number of subsequent jobs. For a full description of handling parameter strings of macros at various command processor levels see the section on multi-level jobs at the end of this chapter.

Two other command sources can cause an increase in the command processor level of a job; these are programs that issue commands and built-in commands that issue commands. A program can issue a command by means of an extracode, PERI type 60 instruction. The issued command is obeyed at a command processor level one greater than the level at which the program was entered. Full details of command issuing programs are given in the section *Command issuing programs* on page 40. Built-in commands have an internal mechanism whereby they are able to issue macros and other built-in commands; such a change of command source increases the command processor level in the normal way. Normally command issued commands need only concern the user when messages are output, since certain messages give details of all the command processor levels that belong to the job.

From the above it can be seen that the command processor level at which a command is obeyed normally varies with the command source. The command sources available are as follows:

1 Paper tape readers

2 Card readers

3 Operator's console

4 MOP terminal

5 Macro definition files (and job description files)

6 Object programs

7 Built-in commands

Commands issued from paper tape readers, card readers, the operator's console and MOP terminals (unless given during a break-in) are obeyed at level zero. Commands issued from macro definition files (including job description files), object programs or built-in commands may be obeyed at any level up to the maximum permitted (25).

Note that a job's command processor level and its context are distinct, though related, characteristics of a job. Certain context changes do correspond to changes of the command processor level; for example the change from NO USER to USER and OFF-LINE or from USER and MOP to USER, MOP and BREAK-IN. However, the command processor level may change many times within a given context; for example a number of macro levels may be created within a USER context. Similarly a change of context may occur without a corresponding change of the command processor level, for example the change from USER to USER and CORE IMAGE.

COMMAND DELIMITERS

The purpose of *command delimiters* is to ensure that parameters in a multi-record command and data in a *composite* command (that is a command followed by embedded data, see page 13) are not mistakenly interpreted as labels or commands by the command processor.

Format of command delimiters

The default values for command delimiters are:

((((	for a *starter,* and
))))	for a *stopper*

When used these must be the first four characters of the line. The value of the delimiters at any macro definition or job description file level may be changed by using the BRACKETS command.

The characters permitted for command delimiters are any non-alphanumeric characters, with the exception of % #-, and space characters.

Notes:

1 Command delimiters are only meaningful within macro definition or job description files.

2 A starter only takes effect in the macro level at which it appears.

3 The starter may not be set to the same four characters as a stopper.

4 A BRACKETS command has no effect on the value of the starter or stopper for command processor levels other than the one at which the command is issued.

The command processor and command delimiters

Command delimiters can be used to surround a command together with subsequent embedded data. After reading the current starter in a macro definition or job description file, the first time the command processor searches for the next command to obey or for a label specified in a GOTO command, it treats all lines of job source in the normal way. Once the first command or free-standing label has been processed, any subsequent search by the command processor ignores everything until the current stopper has been found. Thus an error in executing the command that followed the starter does not cause the command processor to misinterpret unread embedded data as further commands; such data is ignored. If the end of file is reached before the stopper has been found, the effect is as if the last record was a stopper.

As each record is read from a macro definition or job description file, the command processor notes the reading of a delimiter. If a stopper is encountered before a starter, an error is reported:

```
STOPPER FOUND BEFORE STARTER
```

If a starter is found between delimiters when a verb is expected, this gives rise to the standard command error message:

```
VERB FORMAT ERROR
```

Notes:

1 Any record in which the first four characters match the current starter or stopper qualifies as a delimiter.

2 The starter must come before the first record of a multi-record command or the command which is to access the embedded data.

Example 2/1

1 (This example refers to a job description file)

```
ONLINE *CR0
((((
ENTER
.....
.....Data for ONLINE
.....
))))
```

After the ENTER command has been obeyed, the next command to be obeyed will be that following the stopper, even if the program does not read all the data.

2
```
((((
INPUT  :JONES,TAPEDATA,ALLCHAR
lines of input
****
))))
```

3
```
((((
SUBJOB SOURCE1,OBJECT1,ASSIGN,*CR0,–
DAT1, *CP0,OUT1,SAVEDFILE,TIME –
2 SECS
))))
```
where SUBJOB is a macro command

GOTO and command delimiters

GOTO ignores lines between delimiters except the first one. Thus a label between delimiters will only be recognised if it comes on the line immediately after the starter.

Example 2/2

1 In the following sequence of commands:

```
GO TO 2
. . . . . . .
. . . . . . .
((((
SUBJOB SOURCE1,OBJECT1,ASSIGN,*CR0,–
DATA1,*CP0,OUT1,SAVEDFILE,TIME –
2 SECS
))))
. . . . .
. . . . .
2 ENDJOB
```

The command GO TO 2 will pick up the label 2 from the line 2 ENDJOB.

2 In the following sequence of commands:

```
GO 1
.
.
.
.
ONLINE *CR0
((((
1 ENTER
. . . . .
. . . . . Data
. . . . .
))))
```

The command GO 1 will cause a branch to the line 1 ENTER.

3 In the following sequence of commands:

```
.
.
GO TO 2
.
.
.
.
BR <<<< , >>>>
.
<<<<
```

```
SUBJOB SOURCE1,OBJECT1,ASSIGN *CR0,DATA1,*CP0,OUT1,SAVEDFILE,TIME –
2 SECS
>>>>
.
.
2 ENDJOB
```

if the delimiters are not <<<< >>>> at the time the GO TO 2 is obeyed, the label found will be 2 in 2 SECS.

It should be noted that if terminators are specified to be the same as the current starter or stopper anomalies may arise when the GO TO command is used (see *Modes and terminators,* page 175).

TYPES OF JOB DESCRIPTION

Permanently stored job descriptions

The standard way to run a background job under the control of GEORGE is to use programs, data and job descriptions which have been stored in filestore files. Programs are loaded from the filestore and all peripheral transfers are off-line transfers between core and filestore files. The alternative arrangements for handling jobs (with temporary job description files, temporary data files, embedded data or on-line peripherals) can be regarded as variations on the standard method.

INPUT

To store a job description, a program or a program's data in a permanent file the user may issue an INPUT command from a paper tape or card reader, or a MOP terminal (see Chapter 7). The INPUT command informs GEORGE that the information following the command is to be stored in a serial file, the format is as follows:

INPUT *username, filename, terminator*

The formats of user names and file names are explained in Chapter 3. If the INPUT command is introducing a job description, the file name must consist of a local name without internal spaces (for the format of a local name see Chapter 10), since the file name is issued as a macro when the job is run.

The final parameter of the INPUT command is optional and defines an alternative terminator to the input information. If this parameter is omitted GEORGE will expect the input to terminate with a record beginning with four asterisks. If the user specifies that the terminator is to be stored (see the section on terminator formats in Chapter 10) or if the terminator parameter is not given, GEORGE will read to the terminator and store the terminator record followed by a blank record. The blank record is stored to allow programs which use double-buffering to function correctly.

Example

```
INPUT :JONES,TAPEDATA,T####
```

If the file specified in the INPUT command exists and the user includes the APPEND qualifier to the file name, the information following the INPUT command will be added to the named file.

For example:

```
INPUT :JONES,TAPEINPUT(APPEND)
```

OTHER FORMS OF INPUT

As well as inputting programs and data to the filestore from paper tape, card readers, and MOP terminals, by means of the INPUT command, it is also possible by using other commands and system macros to store in filestore files programs and data held on magnetic tape or discs. To store program data, or non-overlaid binary programs in the filestore, the FILEIN command may be used (for full details of the FILEIN command see Chapter 12). FILEIN is in fact a system macro which loads and runs a specially written program. Other cases of magnetic tape or disc to filestore copying can be dealt with by standard ICL library programs; the system macros which run these programs are described in Chapter 13. Examples of the cases that are catered for are copying non-overlaid and over-laid binary programs from a disc, and copying library subroutines from a tape or disc. Since these facilities are provided by means of system macros it is possible for each installation to develop its own specialized media to filestore file copying macros using standard ICL software or specially written programs.

ASSIGNING PERIPHERALS

To use off-line data, held in filestore files, as input to a program, or to send output from a program to a filestore file, it is necessary to connect the input and output channels of the program to the actual source and destination of the data. The command which performs this function is called ASSIGN. The ASSIGN command must be issued in the CORE IMAGE context, normally between the LOAD and ENTER commands in the job description.

The format of the command is:

ASSIGN *peripheral name, file description*

where the file description may be either a file name or a workfile name and may be followed by one or more qualifiers in parentheses. The file is opened by this command, and it remains open until the named peripheral is released.

If a qualifier is given in the file description which is incompatible with the peripheral type that this file is to represent, then the qualifier is ignored. For example

```
ASSIGN *CP0, BILL(*ED,KWORDS100)
```

requests that a card punch file BILL be set up; the qualifiers *ED and KWORDS100, however, indicate a direct access file and are ignored. Later versions of GEORGE may detect contradictory qualifiers and output an error message.

A peripheral can be released and its associated file closed in one of five ways:

1 By the deletion of the program

2 By a REL extracode in the program

3 By an ASSIGN or ONLINE command with the same peripheral name parameter

4 By a RELEASE command

5 By close mode PERI (magnetic media only)

In the standard case that was defined at the beginning of this section both input and output is being handled off-line by means of permanent filestore files. The following two ASSIGN commands could be used to off-line tape reader input and tape punch output:

```
ASSIGN *TR0,TAPEDATA
ASSIGN *TP0,TAPERESULTS
```

These commands will connect the files TAPEDATA and TAPERESULTS to the program, so that PERI instructions for tape reader zero and tape punch zero will cause transfers between the program and these files. When the tape reader is ASSIGNed to TAPEDATA this file must already exist. If there is no such file as TAPERESULTS when the tape punch is ASSIGNed then the ASSIGN command will create the specified file. Full details about connecting each type of filestore file to programs are given in Chapter 3.

JOB INITIATION

A job that is defined by a filed job description is initiated by a RUNJOB command. This command may be issued in any context and at any command processor level. The parameters of RUNJOB specify a name for the job, the user name (if the command is issued in NO USER context) and the local name of the file containing the job description:

RUNJOB *job name, user name, file name*

An example of a permanently stored job to compile and run a program is given below. The job description source program and data are input to the filestore from a paper tape reader.

Note: The macro LINGO used in this and many other examples in this chapter is a purely fictitious example of a compilation macro. The user will find details of the compilation macros currently available under the headings of the various programming languages in Chapter 13 of this manual, or in the relevant compiler manuals.

Example 2/3

```
INPUT :JOHN,JOBFILE1
LINGO *TRPTPROG,BIN,ER1FAIL
ASSIGN *TR0,TAPEDATA
ASSIGN *TP0,TAPERESULTS
ENTER
LISTFILE TAPERESULTS,*TP
1FAIL
ENDJOB
****
INPUT :JOHN,PTPROG
lines of source program
****
INPUT :JOHN,TAPEDATA,ALLCHAR,T####
lines of data
####
RUNJOB ODDJOB,:JOHN,JOBFILE1
DISENGAGE
```

The first INPUT command inputs the lines that follow it, up to the standard terminator (****), and stores them in a job description file named JOBFILE1. The second INPUT command inputs a source program to the file PTPROG. The third INPUT command inputs lines of data to the file TAPEDATA. The mode of this input is ALLCHAR to correspond with the mode that the paper tape reader PERIs in the program require; an alternative terminator is specified because the last line of the data is a standard terminator.

The RUNJOB command initiates the job held in the file JOBFILE1. The source program held in PTPROG is compiled and loaded by means of the LINGO command, the program's peripheral channels are connected to filestore files, and the program is run. When the run is over the results held in TAPERESULTS are output to a tape punch, by the LISTFILE command, and the job is terminated by the ENDJOB command. The batch of paper tape input is terminated by the DISENGAGE command which frees the tape reader for other input operations.

If this job was to run each time with a different set of data it could be modified to erase the input and output files when they were closed:

Example 2/4

```
INPUT :JOHN,JOBFILE1
LINGO *TRPTPROG,BIN,ER1FAIL
ASSIGN *TR0,TAPEDATA
ASSIGN *TP0,TAPERESULTS
ENTER
ERASE TAPEDATA
LISTFILE TAPERESULTS,*TP
ERASE TAPERESULTS
1FAIL
ENDJOB
****
INPUT :JOHN,PTPROG
```
lines of source program
```
****
```

The new data would be input before a RUNJOB command each time the job was to be run:

```
INPUT :JOHN, TAPEDATA, ALLCHAR, T####
```
lines of data
```
####
RUNJOB ODDJOB, :JOHN,JOBFILE1
```

A more practical method of running the same program with different data would be to compile the source program and store the object program in a filestore file. Then, in a separate job, load and run this program with whatever data was required. If it was only necessary to compile the program once this could be done in a once-only job, the run of the program being controlled by a permanently stored job description.

Once-only job descriptions

There is no point in storing a job in a permanent file if the job is to be run once only. Instead the job should be stored in a temporary *working job description file,* which will be erased when the job ends. One command combines the functions of inputting a once-only job to the filestore and initiating the job. This command is the JOB command, which specifies the job name and user name (like RUNJOB) and has optional terminator and mode parameters (like INPUT).

When the JOB command has been read and checked for errors, a *monitoring file* and temporary working job description file are opened with the name that has been given in the job name parameter of the command. The job description is read up to the terminator and stored in the working job description file. The name of the file is then issued as a macro command (as in the RUNJOB command) and the job is run and terminated in the same way as with RUNJOB. When a job introduced by a JOB command is terminated, however, the working job description file containing the job description is erased.

The job in *Example 2/3* could be run as a once-only job. The job description would be as follows:

Example 2/5

```
INPUT :JOHN, PTPROG
lines of source program
****
INPUT :JOHN,TAPEDATA,ALLCHAR,T####
lines of data
####
JOB ODDJOB, :JOHN
LINGO *TR PTPROG,BIN,ER1FAIL
ASSIGN *TR0,TAPEDATA
ASSIGN *TP0,TAPERESULTS
ENTER
LISTFILE TAPERESULTS,*TP
1FAIL
ENDJOB
****
```

Note that in this example the source program and data are input before the job description, because otherwise the job would be initiated before the necessary information had been stored.

If, as in Example 2/4, the job was required to be run with a different set of data each time, the compilation of the program should be done in a once-only job. Then each time the program was to be run the data should be input and a permanently stored job to load and run the program should be initiated. The job description could be as follows:

Example 2/6

```
INPUT :JOHN,PTPROG
lines of source program
****
JOB COMP1, :JOHN
LINGO *TRPTPROG,BIN,ER1FAIL
SAVE OBJECT
1FAIL
ENDJOB
****
INPUT :JOHN,PROGRUN
RESTORE OBJECT
ASSIGN *TR0, TAPEDATA
ASSIGN *TP0, TAPERESULTS
ENTER
LISTFILE TAPERESULTS,*TP
ERASE TAPERESULTS
ERASE TAPEDATA
ENDJOB
****
```

```
INPUT :JOHN,TAPEDATA,T####
lines of data
####
RUNJOB JOB1, :JOHN,PROGRUN
```

The first INPUT command inputs the lines that follow it, up to the standard terminator, and stores them in the file PTPROG. The JOB command that follows, introduces and initiates a once-only job to compile the program; when the object program is loaded into core a SAVE command is issued, which stores the current state of the core image in the file OBJECT (the SAVE command will be fully described later in this chapter). The INPUT command that follows inputs the lines that follow it, up to the standard terminator, and stores them in the job description file PROGRUN. The second INPUT command stores the lines of data that follow it in the file TAPEDATA. The RUNJOB command initiates the job stored in the file PROGRUN. The RESTORE command in the job description loads the object program that is SAVEd in the file OBJECT, the remainder of the job description connects the peripheral channels off-line, then enters the program, lists the results and erases the off-line files. To run the same program with a new set of data the job description required is:

```
INPUT :JOHN,TAPEDATA,T####
lines of new data
####
RUNJOB JOB2, :JOHN,PROGRUN
```

Other input/output techniques

In the previous sections peripheral transfers were controlled by ASSIGN commands that connected the peripheral channels of the object program to permanent data files in the filestore. There are three other methods of handling input and output operations:

1 Embedded INPUT commands

2 Embedded data

3 On-line peripherals

EMBEDDED INPUT COMMANDS

The user may embed his input in his job description preceded by INPUT commands in the normal way. In this case the INPUT command and the data introduced by it are read into the job description file with the rest of the job description. Since the INPUT command is issued in USER context, the command has no user name parameter; apart from this the command has the usual format. The INPUT command is obeyed only when the job is run; the action taken is to copy the data from the job description file into the file specified by the INPUT command. The type of the file created by the INPUT command will be the same as the type of the job description file. It is essential that the terminator of the embedded INPUT command be different from the terminator of the JOB or INPUT command that reads the job into the filestore. If a standard terminator is given for an embedded INPUT command and the JOB command has not specified a different terminator for the job itself, GEORGE will treat the embedded INPUT terminator as the terminator for the job description and will then attempt to obey the command following the asterisks in the NO USER context. This may cause a context error.

Disadvantages of embedded INPUT commands

Since two data transfers are needed when embedded INPUT commands are used, it is generally more efficient to use INPUT commands in the NO USER context. If embedded INPUT commands are used in permanently filed jobs then the same data transfer must be carried out each time the job is run. If the user wishes to run the same job several times with the same data, this data should be INPUT to a file before the first run of the job.

Advantages of embedded INPUT commands

INPUT commands in USER context are more commonly used from MOP terminals. In this case the only data transfer needed is from the MOP terminal to a filestore file.

The other main use of embedded INPUT commands is to allow users with once-only input data to store the data in temporary work files in the job's work file stack. The workfile stack is a collection of temporary files that exist for the duration of the job. They are not dumped for security purposes and are referred to by special *work file names.* Work files must be created before they are used. The command used is

```
CREATE !
```

This sets up a work file (denoted by !) at the top of the stack for the job. For a full description of work files and work file names see Chapter 3. When the work file has been set up, an INPUT command to send data to this file has the form:

INPUT !, *terminator*

There is no point in specifying modes for the file since the file will have the same modes as the file from which the command is issued. Since workfiles are organized in a stack, if another workfile is created then the CREATE command should again have the format

```
CREATE !
```

but if the user should now wish to refer to the first created work file he must specify it as !1 since it is now one from the top of the stack.

If in *Example 2/3* the user wanted to use temporary files he could have used work files instead of using and erasing permanent files.

Example 2/7

The following is an example of a once-only paper tape job using work files.

```
JOB PAPJOB, :JOHN
CREATE !
INPUT !,T????
lines of input data
????
LOAD OBJECT
ASSIGN *TR0,!
CREATE !
ASSIGN *TP0,!
ENTER
LISTFILE !,*TP
ENDJOB
****
```

In this job two workfiles are used to hold the input and output data. Since the first created workfile is not referred to in the job description after the creation of the output workfile, the name ! may always be used. If the second CREATE command had immediately followed the first in the job description, then all subsequent references to the first created work file would have had to refer to !1. Both workfiles are erased when the job ends or, for the output file, when the listing of the file is completed if this is after the ENDJOB command has been obeyed.

EMBEDDED DATA

A single stream of input data for a program may be handled by embedding the data in the job description file and connecting the program's input channel to the job source. Thus there is no need to transfer the data to another file. The ONLINE command or the ASSIGN command may be used to perform the connection. Either command must be issued in CORE IMAGE context; the formats are

ONLINE *peripheral name*

or

ASSIGN *peripheral name*

where *peripheral name* is the name of the peripheral channel that is to be connected to the job description file. If such a command is issued in a job initiated from a MOP console, it will connect the peripheral channel to the job source for a MOP job, that is the MOP terminal. When a job source is made on-line to a program this is an example of a *pseudo on-line peripheral.*

Normally the embedded data and the command which is to access it will be enclosed in command delimiters (see page 5).

Example 2/8

This could be used in a job description file but never in a macro.

```
ONLINE *CR0
((((
ENTER
Embedded data
))))
IF FAILED . . .
```

The use of delimiters ensures that should a program event occur before all the embedded data has been read, the next command to be obeyed is the one following the stopper. Note, however, that should a program read the stopper as embedded data, command processor action will be as if that stopper was omitted. The use of delimiters also ensures that a GOTO command will not accidentally satisfy a label search with a line of embedded data.

Notes:

1 Embedded data can conveniently serve only one input channel of a program.

2 The following commands all have the same effect:

```
ONLINE    *LP
ASSIGN    *LP
ASSIGN    *LP,
ASSIGN    *LP, %(*LP) where %(*LP) is null
```

The last example is of particular use in stored job descriptions and macros, since it is equivalent to the more cumbersome command

```
IF STR(%(*LP)) = (), (ONLINE * LP0) ELSE (ASSIGN *LP0, %(*LP))
```

(see also *Parameter substitution*, page 32).

Advantages of embedded data

The principal advantage of embedded data over embedded INPUT commands is that no transfer of data in the file-store is necessary. This method of handling data is very useful for MOP jobs. In this case single items of data are input from the ONLINEd terminal direct to the program, thus if the results are sent to the terminal as well it is possible to use the terminal in a fully interactive mode, supplying data items, receiving the results of the computation and inputting more data selected because of the previously received results.

ON-LINE PERIPHERALS

Input and output operations can be handled via on-line peripherals in almost the same way as in a conventional system. These on-line peripherals can be basic peripherals, secure or insecure magnetic tapes or exofiles. GEORGE performs certain special checks that are not made in an Executive-controlled system, and GEORGE also supervises certain other operations such as the rewinding of magnetic tapes.

For a full description of using basic peripherals on-line see Chapter 5 *Peripheral handling.* For details of the use of magnetic tapes and exofiles see Chapter 4 *Using entrants outside the filestore.*

ADVANTAGES OF OFF-LINE INPUT AND OUTPUT

GEORGE provides facilities to hold in filestore files any file which has a serial basic peripheral format, or is organised as a direct access file, or which has a magnetic tape file format. Thus basic input and output may be off-lined via the filestore, and direct access and magnetic tape filestore files grant the user all the filestore's organizational controls over these types of file. The specific advantages of using each type of filestore file are discussed below:

1 BASIC PERIPHERALS

(a) Peripheral transfers between filestore files and object programs are faster than using on-line basic peripherals.

(b) Since on-line peripheral transfers are slower, the object program has to be kept in core for longer periods of time.

(c) On-line basic peripherals tend to be kept in use by one user far longer than when the peripheral is used for an input to or output from the filestore. If many users handle their input and output with on-line basic peripherals it is possible for serious peripheral jams to occur.

2 MAGNETIC TAPE

(a) Tape decks are often used to hold tapes that are on-line to programs that are being developed and are also often used to hold tapes which contain only a small amount of data. Thus there is often wastage in tape utilisation and in deck utilisation.

Also, if an installation has many tapes that hold a small amount of information this will inevitably mean that the installation's tape library is larger than the amount of data in it warrants. Using magnetic tape filestore files will greatly ease these problems by economising on tape and deck usage.

(b) If worktapes are often used, valuable deck space is taken up and frequent operator intervention is necessary. If magnetic tape work files are used, worktapes are freely available with no need for operator action.

(c) Since magnetic tape files are usually held on direct access devices access to them is usually faster and skips and rewinds take much less time. Also magnetic tape files do not need to be initially loaded and thus reduce operator intervention.

(d) Magnetic tapes files are automatically dumped giving file security with tape economy.

(e) Magnetic tape files may be read by several jobs simultaneously.

3 DIRECT ACCESS

Direct access files are automatically dumped giving file security.

Summary

JOB DESCRIPTIONS

There are two kinds:

1 Permanently stored job descriptions, filed by INPUT and initiated by RUNJOB.

2 Once-only job descriptions, filed and run by JOB.

PERIPHERALS

With both kinds of job description program input/output operations can be handled in the following ways.

Input

1 Basic peripheral files and basic peripheral documents

(a) Data is stored in a permanent file by INPUT (NO USER context). The file is connected to the program by ASSIGN.

(b) Data is stored in a permanent or temporary file by an INPUT command (after CREATE for a temporary file) embedded in the job description. The file is connected to the program by ASSIGN.

(c) Data is embedded in the job description. The job description file is connected to the program by ONLINE or ASSIGN.

2 Magnetic tape files and magnetic tape

(a) The file is connected to the program by ASSIGN.

(b) The magnetic tape is connected to the program by ONLINE.

(c) The magnetic tape is connected to the program by unanticipated open mode PERI.

3 Direct access files and exofiles

(a) The file is connected to the program by ASSIGN.

(b) The exofile is connected to the program by ONLINE.

(c) The exofile is connected to the program by unanticipated open mode PERI.

Output

1 Basic peripheral files and basic peripheral output

(a) Data is written to a permanent file, created (if necessary) and connected to the program by ASSIGN.

(b) Data is written to a temporary file created by CREATE and connected to the program by ASSIGN.

(c) Data is written to the job's monitoring file, which is connected to the program by ONLINE or ASSIGN.

(d) Data is output on a basic peripheral, which is connected to the program by ONLINE.

2 Magnetic tape files and magnetic tape

(a) The file is connected to the program by ASSIGN.

(b) The magnetic tape is connected to the program by ONLINE.

(c) The magnetic tape is connected to the program by unanticipated open mode PERI.

3 Direct access files and exofiles

(a) The file is connected to the program by ASSIGN.

(b) The exofile is connected to the program by ONLINE.

(c) The exofile is connected to the program by unanticipated open mode PERI.

THE MONITORING FILE SYSTEM

When a job is initiated by a JOB, RUNJOB or LOGIN command, a file, called the monitoring file, is created to hold monitoring information produced by the job. The file has the same name as the job, and a language code B1B0. Information that is sent to a job's monitoring file in the course of the job is stored in a number of *categories;* each category is described below. In the event of a job's monitoring file becoming full in the course of the job, the job will be abandoned, as by an ABANDON command.

Monitoring file categories

BROADCAS

This category contains messages output to the monitoring file by the BROADCAST command.

CENTRAL

This category contains messages sent to the central operator's console and is a subset of the OPERATOR category.

CLUSTER

This category contains messages sent to the cluster console and is a subset of the OPERATOR category.

COMERR

This category consists of command error messages. The command error reporting system has been designed to give the user as much information as possible about errors. To make error reporting as immediately informative as possible, error messages are written in standard English, and each message gives details of both the reason for the error and the command processor level at which the error occurred.

If there is no monitoring file in existence, because a job has not yet been initiated, command error messages are output at the operator's console or, in the case of commands issued in the NO USER and MOP context, at the

MOP console from which the command in error was issued. In a MOP job all categories of monitoring information may also be output at the MOP terminal (for details see the REPORT command in Chapter 12 and Chapter 7 *Using MOP).*

A description of the command error reporting system is given in Chapter 11. Error messages that are common to a number of commands are listed in this chapter. Error messages that occur only with a specific command are listed under the appropriate command.

COMMANDS

This category contains a copy of each command, including macros, that has been read by the command processor. It also contains lines of comments introduced by #.

COMMENT

A number of commands generate special replies, which are stored in the COMMENT category. Replies generated before the creation of a job's monitoring file are treated in the same way as command error messages (see above).

DISPLAY

Messages are sent to this category by the DISPLAY, QUESTION, and ANSWER commands. Note that the output of the PLAN extracodes DISP and DISTY is held in the OBJECT category (see below) rather than in this category.

ENGINEER

This category is a subset of JOURNAL (see below) and contains messages relating to peripheral transfers and peripheral failures. ENGINEER messages are not documented in this manual.

FILES

This category contains information about filestore changes that occur in the course of a job.

JOURNAL

This category contains all messages relating to the job that have been sent to the System Journal. The latter is a system file in which GEORGE collects a variety of information that is of interest to the installation manager. In particular the information is used by the manager in the following areas:

1 ACCOUNTING All the information required by the installations logging and accounting systems is collected in the System Journal (see also *LOGGING,* below).

2 MONITORING HARDWARE RELIABILITY Information relating to peripheral failures is stored in the System Journal (see also *ENGINEER,* above).

LISTING

The LISTFILE and LISTDIR commands either output a file on a basic peripheral or send the file to the monitoring file system. In the latter case the file is stored in the monitoring file in the LISTING category. The LISTFILE and LISTDIR commands are described in Chapter 3.

LOGGING

The information used by the log analysis program to calculate the charge for running a job is held in this category. This information consists of details of the amount of mill time and core used by the job. Logging messages generated by a command are listed in the specification of the command that produces them (see Chapter 12 for the specifications of the commands).

OBJECT

This category consists of information generated by object programs, such as program output directed to the monitoring file system, extracode messages and reports of program failures. The output of the HALT, FAIL and DELETE commands is treated as extracode output and is stored in this category. Details of how GEORGE handles events in object programs are given under *Program Events,* page 22.2.

ONLINE

This category is a subset of the OBJECT category. It holds information produced by object programs as a result of issuing an ONLINE command for a basic peripheral without specifying an output document name. The action is to divert the output of the ONLINEd peripheral channel to the monitoring file.

OPERATOR

This category contains all messages sent to the operators. It is the combination of the CENTRAL and CLUSTER categories.

POSTMORT

This category holds the output of the POSTMORTEM and PRINT commands, which are used to print regions of core image.

The TRACE system

The number of categories which are output to a job's monitoring file is initially set by installation parameters and may be varied using the TRACE command. When a background job is started the categories output to its monitoring file are determined by the installation parameter JOBTRACE (for a MOP job the relevant parameter is MOPTRACE). The format of the TRACE command is

TRACE *action on monitoring file*

where action on monitoring file consists of a string of parameters separated by commas, that specify the monitoring file categories which the job's monitoring file is to contain. The minimum TRACEing level is set by the installation parameter MINTRACE and categories specified by this installation parameter will always be included in the categories set by a TRACE command. Examples of TRACE commands when MINTRACE is set to LOGGING are:

TRACE ALLBUT, COMMANDS	In this case all categories other than COMMANDS will be included in the monitoring file.
TRACE LISTING, COMERR	In this case the categories included will be LISTING, COMERR and LOGGING.
TRACE FULL	In this case all categories other than CENTRAL, CLUSTER, ENGINEER and JOURNAL will be included (see also *Output from the monitoring file,* page 176)
TRACE FULLBUT, COMMANDS, COMERR	In this case all categories will be included except CENTRAL, CLUSTER, ENGINEER, JOURNAL, COMMANDS and COMERR

The user should be certain that he will not require information from any category he suppresses, as such information will not be available at the end of the job.

Any TRACE command in a job will supersede a previous TRACE command at the same command processor level. A TRACE command issued in a macro can only include categories which have not been excluded by a TRACE command at a higher level. If a TRACE command is given in a macro, when the macro ends the TRACE level reverts to that set by a TRACE command at a level that called the macro, or if no TRACE command was issued at that level to the set of categories which were included when the macro was called. For a full specification of the TRACE command see Chapter 12.

If a FULLTRACE command is issued in a job then all monitoring file categories are included in the job's monitoring file, and no subsequent TRACE commands in the job will have any effect. The format of the FULLTRACE command is simply:

FULLTRACE

For the MOP user, there is also the REPORT command which enables him to specify which monitoring file categories should be output to the MOP console during the job. For further details see page 136.

Action on the monitoring file

When the user terminates a job (normally with an ENDJOB or LOGOUT command) he can specify which categories of monitoring information are to be listed, by means of parameters of the terminating command. For example:

ENDJOB COMMANDS, COMERR

ENDJOB ALL

```
ENDJOB ALLBUT, FILES, COMMENT
ENDJOB NONE
```

The terms ALL, ALLBUT and NONE allow users to indicate groups of monitoring file categories. The terminating command (such as ENDJOB) deletes any core image and calls in the built in log analysis routine if this form of accounting is in use. For details of how log analysis affects the user see Chapter 9, page 153.

RETAINING THE MONITORING FILE

After the required monitoring file categories have been listed, the monitoring file is erased. However it is possible to retain the monitoring file by copying it into a named filestore file. This is done by including a RETAIN (*local name*) parameter in the terminating command of the job. The format of ENDJOB might then be:

```
ENDJOB ALL, RETAIN(FRED)
```

In this case all the monitoring file is listed and the file is retained by being copied to a file FRED.

Note that since the monitoring file is copied, the standard rules for creating terminal files apply (see Chapter 3).

If the user subsequently wishes to erase the copied monitoring file he should issue an ERASE command with the file name as its parameter, in the normal way.

The current reply

Each message that is sent to any category of the monitoring file system, except COMMANDS or ONLINE, is set as the job's current reply. When the next message is sent this will become the current reply. The job's current reply may be used by the IF command and the SETPARAM command, but users should note that the exact wording of these messages is not guaranteed to remain constant in all future versions of GEORGE. For further details see the specifications of these commands in Chapter 12 and the descriptions of the use of these commands later in this chapter.

STANDARD FACILITIES FOR RUNNING JOBS

This section describes the facilities that GEORGE provides to deal with organizational problems involved in running a job, which under a conventional environment would require operator intervention.

Controlling job and program run times

In order to prevent a job as a whole or an object program from wasting an excessive amount of mill time if it behaves in an unexpected manner (by looping for example), two timers are provided, the Job Timer and the Program Timer. These timers are decremented by the amount of mill time used to perform commands and run programs. If they run out special action is taken by GEORGE.

The Job Timer is used up by running the job. It includes the amount of mill time measured by the Program Timer, plus an estimated amount of mill time used by GEORGE in performing commands.

The Program Timer is used up by running programs within a job. It includes the mill time used in executing a program, plus an estimate for the mill time used by GEORGE in performing off-line peripheral transfers, program-issued commands and in servicing program events and illegals.

THE JOB TIMER

The length of time permitted to a job can be set either by the JOBTIME command or the installation parameter JOBTIME. If a JOBTIME command is not given, the maximum time allowed will be sent to the current value of the JOBTIME installation parameter. If the user wishes to set a different limit for a particular job he must issue a JOBTIME command. (Time used before the JOBTIME command is issued, is subtracted from the value given by the command). An example of the JOBTIME command is:

```
JOBTIME     2MINS
```

If a background job is completed before the time limit has been reached, it is terminated in the normal way. If, however, the time limit is reached before the job has been terminated naturally, the message

```
JOB ABANDONED:  JOBTIME EXCEEDED
```

is sent to the monitoring file system and an ENDJOB command with no parameters is obeyed. Once a user has set a milltime limit on his job with a JOBTIME command, he cannot change it. If he tries to issue another JOBTIME command, a command error will result.

When a user at a MOP terminal has reached the job limit his job is not abandoned. A message:

```
JOB TIME EXCEEDED:  MORE TIME HAS BEEN ALLOCATED
```

is sent to the monitoring file in the COMMENT category, and according to whether the user has switched off the REPORTing of comments, the message is also sent to the MOP terminal. The amount of extra time added for the job at this point is that equal to the value of the Installation Parameter JOBTIME, irrespective of whether the MOP user has issued a JOBTIME command of a different value. This means that the MOP user will be sent the JOB TIME EXCEEDED: MORE TIME HAS BEEN ALLOCATED message every time he has used an amount of time equal to the Installation Parameter JOBTIME, except in the first instance when he will receive the message after he has used the amount of time specified by his own JOBTIME command, if he has issued this command.

THE PROGRAM TIMER

The maximum time for which an object program is allowed to run is set by the installation parameter PROGTIME, but this may be over-ridden by a TIME command in the job description. When a LOAD command is issued the program timer is set to the value of PROGTIME, and so if the user wishes to alter the time limit, a TIME command must be given between the LOAD and ENTER commands for the program. Note that any RESUME or ENTER command (see the following section) will, if the time is not positive, reset it to the value of PROGTIME. An example of a TIME command is:

```
TIME 5SECS
```

If the program is completed before the time limit has been reached, it is terminated in the normal way. If, however, the time limit is reached before the program has been terminated, the message

```
TIME UP
```

is sent to the monitoring file (OBJECT category), the program is stopped and a program event of the FAILED (TIME UP) category is generated (see *Program events,* page 22.2) which may be tested by an IF command.

LOGGING INFORMATION

Whenever a program is deleted, a message of the following format is sent to the monitoring file:

time *jobtime* DELETED, CLOCKED *progtime*

The *time* is the clock time when the program was deleted. The *jobtime* is the amount of mill time used by the job since it started. It is equal to the amount by which the Job Timer has been decremented.

The *progtime* is the amount of mill time used by the program since it was loaded. It does *not* include the estimated mill time used by GEORGE in performing functions for the program, and does not therefore correspond exactly with the amount of time measured by the Program Timer.

Compiling and running a program

To compile a program the user should issue the appropriate compiler system macro command, which will call in a compiler and, in most cases, a consolidator and perform the compilation automatically. These macros provide the user with a full range of compilation options for each language, and are described in the relevant compiler manuals.

Loading and entering programs

INTRODUCTION

This introduction gives a brief outline of the process of loading and running programs under GEORGE. It is followed by a more detailed explanation of the loading process.

Programs are stored in filestore files or entrants outside the filestore in binary program format. Programs may be stored by compilers, library updating software or, in the case of a filestore file, by a SAVE command.

Before a program can be run it must be converted to a form suitable for being held in the computer's core store. This process is termed loading the program and is initiated by commands such as LOAD or FIND. It should be noted, however, that the process of loading may not proceed as far as placing the program into core store until the program is able to run. This is to improve the efficiency of use of core store, the latter being an expensive resource.

After the loading process has been initiated several things may be done, the most important being to connect the program's peripheral channels to filestore files or on-line devices by means of commands such as ASSIGN or ONLINE. When the program is ready to be run this is indicated by a command such as ENTER or RESUME. When such a command is obeyed, the program is entered on a list of programs waiting to be run when system resources permit. The decision to start the program run is made by the GEORGE routine called the low level scheduler which then completes the loading process by copying the program into core and allowing it to run. When the program has run for an interval decided by the low level scheduler, or if it halts for some reason, it will be 'swapped out' of core, on to a special area of backing store, to allow other programs to run. When it is again decided by the low level scheduler that the swapped out program should be allowed to run, the program will be automatically 'swapped in' to core. The process of scheduling by the low level scheduler is designed to allow GEORGE to run more programs than there is room for in core at the same time, and as a result to optimize the use of costly system resources such as central processor time.

When a program is no longer required it may either delete itself by an instruction issued within the program, or be deleted by a DELETE command in the job description. A third method of deleting a program is for the job to initiate the loading of another program – a job is allowed to have only one program loaded at a time.

A successful LOAD command puts the job into CORE IMAGE context. The job remains in this context until the program is deleted. During this period the program is often referred to as a core image. The term core image originates from the fact that the program retains a format suitable for being in core even if it becomes swapped out.

INITIATING THE LOADING PROCESS

The GEORGE commands that initiate the loading process are LOAD, FIND, LOADENTER and RESUME. The LOAD command is used to initiate the loading of a binary object program held in a filestore file in card, paper tape, magnetic tape, or direct access format. An example of a LOAD command is

```
LOAD    OBJECTPROG
```

where OBJECTPROG is the name of a filestore file with a suitable format. If the file is in magnetic tape or direct access format, the program must be the first program in the file or preceded only by a General Purpose Loader or #TAPE seek program. If the program is in a file produced by a magnetic tape compiling system, the first program in the file may be:

1 The program required. This will be dealt with by LOAD.

2 A seek program, for example #TAPE. In this case the required program will occur later in the file and the FIND command (see below) should be used. This is because the LOAD command would only successfully load the seek program and the required program would never be loaded.

3 A General Purpose Loader (GPL). In this case the required program will follow the GPL and will be in consolidated semi-compiled form. The LOAD command will load the GPL and run it before returning to command level, with the consequence that the required program is loaded after completion of the LOAD command.

4 An overlay loader. The overlay loader can be dealt with by LOAD, but it will require a worktape to use in the process of converting the required program from consolidated semi-compiled form to binary. The overlay loader will not automatically be run by GEORGE so in this case it would again be preferable to use the FIND command.

When the LOAD command is successfully executed, the program becomes the job's current core image. If the job already had a core image this will be destroyed and replaced by the new one.

There is a facility that permits the user to override the core or size requirement stated in the program's request slip. This is achieved by including a CORE or SIZE parameter in the LOAD command. For example if the program to be loaded from MYPROG has a core or, in GEORGE 4, size requirement of 7,000 words specified in its request slip, but the user knows that only 6,000 words will be required on this run, he can issue a LOAD command of the form:

```
LOAD MYPROG, CORE 6000
```

It should be noted that the CORE and SIZE parameters are interchangeable, but that GEORGE 3 and GEORGE 4 execute them differently.

There are also CORE and SIZE commands that can be issued at any time that the job is in CORE IMAGE context. However, users are advised to restrict their use of the CORE and SIZE commands to when it is required to modify the core allocation of a program that has begun to run. At the time of initiating the loading process it is more efficient to use the CORE or SIZE parameter in the LOAD command. Note that the core size specified by the user in either a CORE or SIZE parameter or a CORE or SIZE command is rounded, in GEORGE 3 up to the nearest multiple of 64, and in GEORGE 4 up to the nearest multiple of 1024. In the example above, the actual core size allocated would therefore be 6016 under GEORGE 3, or 6144 under GEORGE 4.

If the program loaded has either of the 'retain load peripheral' bits set (bits 0 or 1 of the third word of the request slip), for example in an overlay program, and the filename parameter to the LOAD command is not null, then LOAD issues an ASSIGN command to connect the file to the program. In this case the user must have both READ and EXECUTE traps set for the file. If the user is not allowed READ access, the ASSIGN command will fail. However, the loading process will succeed and when the program is entered and attempts to access the load peripheral a program event will be caused.

If a LOAD command is issued with a null file description parameter, whether or not the 'retain load peripheral' bits are set it is assumed that the binary dump immediately follows the LOAD command and is completely contained within the file from which the LOAD command was issued. The 'retain load peripheral' bits are ignored and no traps are required on this file apart from those that were required to allow the LOAD command to be issued. Note that the LOAD command and binary dump may be enclosed within command delimiters (see *Command delimiters,* page 5). With this format of the LOAD command, the loading process is completed immediately.

Several cases were noted above where it would be preferable to use a FIND command. There are also two cases where it is essential. These cases are when a user wishes to load a program from an on-line magnetic tape or from an exofile. The format of FIND is:

FIND *program name,device type,entrant description,*CORE*number*,OVERLAYS *magnetic tape description*

Note that CORE may be replaced by SIZE.

A full description of the FIND command is given in Chapter 12. A simple example of loading a program with a FIND command is given below:

```
FIND #AL01,MT,PROGRAM∇TAPE(45)
```

#AL01 will be loaded from the tape with the tape name PROGRAM∇TAPE with generation number 45. Since the FIND command is a macro that runs a bootstrap program, the two phases of initiating and completing the loading process do not apply. FIND completes the loading process leaving a core image in core under the control of the low level scheduler.

The LOAD command can be used to initiate the loading of a SAVEd core image held in a filestore file. (The action of the RESTORE command is the same as a LOAD command, and all the conditions specified above also apply to RESTORE; however LOAD should be used in preference to RESTORE).

Errors in the LOAD command will cause the initiation of the loading process to fail and the appropriate error message is sent to the monitoring file. No core image will be created.

COMPLETING THE LOADING PROCESS

In the above description it was noted that the loading process was immediately completed in the following cases:

1 If the FIND command was used

2 If a LOAD command with a null file name was used

3 If the program to be loaded was preceded by a GPL.

In all other cases of the load command, it is not necessary to complete the loading process until either the program is ready to run or a command is issued that makes it essential that the program is in fact loaded into core. Commands that cause the loading process to be completed are as follows:

1 ENTER, RESUME, LOADENTER

2 SAVE

3 CORE (or SIZE)

4 IF commands with ON or OFF condition parameters

5 PRINT

6 ON, OFF, ALTER

7 Any command that has a legal [*number*] parameter (see *Numbers*, page 163)

ENTER, RESUME and LOADENTER are all commands that indicate to GEORGE that the current core image is now in a state ready to be run. SAVE copies the current core image to a filestore file named in the command. It is necessary for GEORGE to have completed the loading of the current core image before it can be written away. CORE (or SIZE), IF ON, IF OFF and PRINT commands all require GEORGE to read part of the current core image. Thus loading must be completed before these commands can be executed. ON, OFF and ALTER commands, on the other hand, write to the current core image. Thus again loading must be completed before these commands can be executed. Commands with a [*number*] parameter access the contents of the location *number* and thus require loading to have been completed. All these commands that cause the loading process to be completed should be delayed as long as possible in the job description. In particular the ON, OFF and ALTER commands should, where possible, be left till just before the program is entered.

If the size of the program that is to be loaded is in excess of the object program quota OBJECTQUOTA (an installation parameter) the following message is output to OBJECT category in the monitoring file, and may also be sent to the MOP terminal:

```
LOADING OF YOUR PROGRAM MAY BE DELAYED AS ITS CORE SIZE
EXCEEDS THE PROGRAM QUOTA
```

When the loading process is completed the following logging message is output

time milltime CORE GIVEN *number*

where number is the size of the program in words.

It was noted above that if the program to be loaded by GEORGE was preceded by a GPL, then GEORGE would load and run the GPL, resulting in the required program being left in core, and then halt LD thus returning to command level. It should be noted that GEORGE recognizes a program as a GPL by the fact that the GPL bit (bit 0 of the third word of the request slip) is set. Setting of the GPL bit under GEORGE is restricted to General Purpose Loaders and attempts to set this bit for other programs will have indeterminate results and should thus be avoided.

The LOADENTER and RESUME *file description* (that is first LOAD then RESUME) commands cannot be used to load a program with the GPL bit set. If an attempt is made to use these commands in this way, they report the error:

z IS A GENERAL PURPOSE LOADER

If the program to be loaded has any of bits 2 to 23 of the overlay directory word, in its request slip, non zero, then the overlay directory will be set up from the subfile description in the program file. Note also that some programs which have been overlaid from direct access files have the overlay directory word set to zero and a GO entry block with the address of a routine to set up the overlay directory. In this case the LOAD command leaves the program ready to be RESUMEd at this address : this must be done before entering the program in the normal way. Note that the RESUME command should be issued as late as possible in the job description since it will force the loading process to be completed.

Errors

Errors during the completion of the loading process will be rare: if such an error occurs it is treated as an error in the command that forced the completion of the load. All error messages are listed in Chapter 11, and include the name of the file from which loading was attempted. The job will be returned to NO CORE IMAGE context and any peripherals ONLINEd or ASSIGNed will be lost.

ENTERING A LOADED PROGRAM

It was mentioned above that between a LOAD command and initiating the run of a program various commands such as ASSIGN may be issued. When the loading of a program is completed, the program is in a state ready to be run. The command that initiates the run is ENTER (which also causes the completion of an uncompleted load). An example of an ENTER command is

```
ENTER 1,PARAM(128,29)
```

In this command the first parameter indicates the entry point for the program, in this case word 21. The second parameter provides 2 run time parameters that will be passed to the program when it asks for them by means of a special mode of PERI extracode (see *Command issuing programs,* page 40).

If it is not required to issue any command between the LOAD command that initiates the program loading and the ENTER command that completes the load and initiates the program run, then the functions of the LOAD and ENTER command may be concatenated by means of a LOADENTER command. A LOADENTER command to load a program from the file MYPROG and enter it in the same way as in the ENTER command example above would be:

```
LOADENTER MYPROG,1,PARAM(128,29)
```

If the entry point parameter in an ENTER or LOADENTER command is null, then the program will be entered at word 20.

A RESUME command may also be used to initiate a program run. This command can either nominate a file from which a core image is to be loaded before being run, or can initiate a run of the current core image. An important point to note is that though RESUME has parameters to indicate the entry point and pass run time parameters to the program when required in the same way as ENTER, it differs from ENTER in that the entry point is specified relative to word 0 of the program rather than word 20. Thus 0 to 9 is the permitted range for the entry point parameter to the ENTER command whereas the size of the program determines the limit for RESUME

For example

```
ENTER 9
```

is equivalent to

```
RESUME 29
```

The other important difference between ENTER and RESUME is that if the entry point parameter is null, RESUME will (as its name implies) enter the program at the address held in word 8 of the program, which holds the address of the next instruction to be obeyed in the sequence that was under way when the core image was last activated.

FURTHER FACILITIES FOR GEORGE 4

Appendix 1 introduces the 1900 paging scheme and discusses briefly the concepts of virtual store, sparse and dense programs, size and shareability. It also introduces the idea of a program's quota and examines the way GEORGE 4 controls the running of programs on a paged machine.

GEORGE 3 is fully compatible with GEORGE 4 and all programs, macros and job descriptions that run under GEORGE 3 will run under GEORGE 4. This means that dense programs may be loaded and entered in GEORGE 4 in exactly the same way as they would in GEORGE 3. However sparse programs may be run under GEORGE 4 only; certain commands have additional parameters in GEORGE 4 to enable sparse programs to be loaded and entered and there are new commands to enable the sizes and quotas of both sparse and dense programs to be specified or varied. Quotas have no meaning in GEORGE 3, which ignores such parameters; the same applies to shareable areas (mentioned below).

Quotas and areas left in core

The quota for a program may be obtained by GEORGE 4 either from the supplementary request slip, or from the LOAD command or a QUOTA command, described below. In the absence of the quota being specified by any of these means, the initial quota to a program on loading is deduced by GEORGE 4 from the size of the program. GEORGE will assume that pages at the low address end of a program will be required in core first and try to arrange that these will be in core on completion of the LOAD command. For a dense program it is assumed that the lowest pages referred to in the binary program are those that will be required, and for a sparse program the lowest pages other than 1, 2 and 3.

In the case of a sparse program the initial quota given to a program on loading may be specified in the (supplementary) request slip. However, if the quota so specified is not the quota required, the user can override the supplementary request slip by including a QUOTA perameter in the LOAD command. For example, if the program contained in PTOBJECT has a quota of 2,000 words specified in its supplementary request slip but the user knows that on this particular run the program will require a quota of 4,000 words to run efficiently, he can issue a LOAD command of the form

```
LOAD    PTOBJECT,QUOTA 4000
```

The quota specified will be rounded up to the nearest multiple of 1024. In the example above, the actual quota given would be 4096.

There is also a QUOTA command which can be issued at any time after a program has been loaded. It performs the same function as the QUOTA parameter of LOAD andcan be used to alter the quota of a program (sparse or dense). Thus, in the example above the same effect is achieved by issuing the two commands:

```
LOAD PTOBJECT

QUOTA 4000
```

It is however recommended that, where possible, the QUOTA parameter of the LOAD command should be used in preference to the QUOTA command.

GEORGE 4 is normally free to adjust a program's quota whilst the program is running. However, a quota may be fixed by specifying a FIX parameter to the QUOTA command (see the QUOTA command).

Loading and entering shareable programs

In GEORGE 4 there is an additional parameter to the LOAD command, the character string PRIVATE which enables a user to load his own private copy of a program that has been marked as shareable by the GEORGE 4 compiling systems (see also *The PRINT and ALTER commands,* page 28).

PROGRAM EVENTS

GEORGE provides powerful facilities for monitoring object program runs. Various commands can be given after the ENTER command in the job description to tell GEORGE what action to take if a program event occurs. A program event is an occurrence in an object program run that causes GEORGE to terminate the current run and read the next command at the current command processor level. Normally a program event is caused either by obeying a control extracode (SUSWT, SUSTY, DEL and DELTY) within the object program or else when the program goes illegal in some way. It is also possible for the user to specify that program events are to be generated in circumstances in which GEORGE would normally take some other action. This is done by issuing a MONITOR command, the significance of this facility being that it enables the user to regulate or supplement the standard program events and thus increases the user's scope in monitoring a test program. Finally, the user can bring about the effects of program events of various kinds by issuing HALT, FAIL and DELETE commands. These commands generate *pseudo program events.*

Associated with each program event there is a *category,* a *message* and a *member number.* When a program event occurs a message is sent to a special area of core reserved to hold the current *program event message.* Together with the message, the category of the event and the number of the program member in which the event occurred are stored in this reserved area. A job can have only one program event message at a time; the message is unset at the beginning of the job and is set and unset by various commands and extracodes during the job. Note that the message is unset by all commands that load or enter a program, and it is set or reset by all program events or pseudo program events.

The current program event message can be examined by the IF command to determine the action to be taken after a program event has occurred. It can also be used by a SETPARAM command to make a parameter of a command dependent on the contents of the message. Both these facilities are described in full later in this chapter.

When a program event occurs the category, message and member number are also sent to the OBJECT category of the job's monitoring file. In the case of a program failure, additional information about the failure is also sent to the monitoring file. This information, which includes the address of the failing instruction, is immediately available to a MOP user who may then take whatever action his job requires. A background user may examine this information when the job is terminated and the monitoring file is listed. The effects of program events and the messages produced by them are described in detail in the section *Description of program events* in Chapter 14.

A program can indicate that its illegals are to be monitored by itself rather than by GEORGE. In this case if the program goes illegal, instead of causing a program event, GEORGE will cause the program to enter its own monitoring routine.

CATEGORIES OF PROGRAM EVENT

Depending on the cause of the event, a program event will belong to one of four categories. The four categories are as follows:

1 HALTED events

These are caused by SUSWT and SUSTY extracodes. The message associated with the event consists of the message generated by the extracode.

2 DELETED events

These are caused by DEL and DELTY extracodes. Again the message consists of the message generated by the extracode.

3 FAILED events

These are caused by program failures, for example an illegal instruction, a peripheral failure or the program running out of time. The message associated with the event gives details of the cause of the event whenever this is possible (see Chapter 11, page 179 for details). If the event is caused by an illegal instruction, no guarantee can be given that the instruction has had no effect. In this case the event message will begin with

```
ILLEGAL
```

This will be followed by as accurate a description as possible of the cause of the illegality, for example:

```
ILLEGAL:RESERVATION VIOLATION
```

In some cases GEORGE will not attempt to identify the cause of the illegality and will give the general message:

```
ILLEGAL INSTRUCTION
```

4 MONITOR events

These are events that have been specified in a MONITOR command issued between loading and entering the program. There are four types of MONITOR command that cause specified incidents in a program to bring about program events of the MONITOR category:

(a) MONITOR ON, DISENGAGED, *peripheral name*

This causes the job to return to command level if a PERI for a named on-line peripheral is encountered in the program, and the peripheral is disengaged. The message associated with the event is

DISENGAGED, *peripheral name*

(b) MONITOR ON, DISPLAY

This causes DISP and DISTY extracodes to generate program events. Normally DISP and DISTY (and the DISPLAY command) do not cause program events; the message generated is sent to a special area of core reserved for the current display of the job (analogous to the current program event message or the current reply), and the program run continues without interruption. When the job has returned to command level because the program run is over or an event has occurred, the current display can be tested by an IF command.

If, however, a MONITOR ON, DISPLAY command has been given, a DISP or DISTY extracode will be set as the current display, and the message DISPLAY will be set as the current program event message. The job will return to command level and the command processor will read the next command at the current level.

(c) MONITOR ON, DELETE

This causes DEL and DELTY extracodes to be treated as SUSWT and SUSTY respectively. If this command is issued, a DEL or DELTY extracode will not delete the current core image, but will cause a program event of the HALTED category. The program event message is as for normal HALTED events.

(d) MONITOR ON, *monitoring expression*

This causes the job to return to command level when any of the types of extracode specified in *monitoring expression* are encountered. The extracodes that can be monitored in this way are: REL, DIS, CONT, certain types of ALLOT, and open, close, extend or rename PERIs (including write after rewind). The format of the *monitoring expression* is described in the specification of the MONITOR command in Chapter 12. If an illegal extracode is encountered after a MONITOR ON command has been issued for extracodes of the same type as the illegal one, the effect is indeterminate. The program event generated will be of either FAILED or MONITOR category.

The program event message will include as many of the following fields as are relevant:

(i) Type of extracode monitored

(ii) Device involved

(iii) A single character Y or N (for YES or NO) to indicate whether or not the device has been allocated to the program

(iv) Address of the control area (PERI extracodes only)

(v) Mode of PERI

(vi) Filename, if any, associated with the PERI

The event message has a fixed format to enable its individual fields to be accessed by the SETPARAM command or the % ; ; substitution facility (see page 32.1). This format is given in the description of program events (monitor category) on page 505.

A monitor event is also generated by the MONRESUME command when the first parameter of that command is STOP (see the MONRESUME command). A previously monitored instruction is obeyed again with monitoring suppressed and a MONITOR event is generated with the message.

MONRESUME COMPLETED

Monitoring can be turned off for any one of the above event types by a command of the form:

MONITOR OFF, *event type*

or MONITOR OFF, *monitoring expression*

MEMBER NUMBERS

Normally a program event is generated as a result of a single instruction obeyed by a particular member of a program, and so a unique member number can be associated with the event. When an event is not the direct result of a single instruction, for example:

1 When a program exceeds the mill time allowed to it

2 When a peripheral that is on-line to a program fails

3 When a pseudo program event is generated by a HALT, FAIL or DELETE command

the member number of the current program member is associated with the event.

The IF command can be used to discover which program member has generated the most recent program event.

THE STATE OF THE CORE IMAGE

The state of the core image after a program event has occurred depends on the cause of the event. There are the following possibilities:

1 The core image will have been deleted.

 This will be the case if the event is of the DELETED category.

2 The core image is left ready to obey the instruction after the instruction that caused the event.

 This will be the case if the event is of the HALTED category or if it is of the MONITOR category and has not been generated by a PERI for a disengaged on-line peripheral.

3 The core image is left ready to obey the instruction that caused the event. This will be the case if the event is generated by an illegal instruction or by a PERI for a disengaged on-line peripheral when monitoring is set for this occurrence. Illegal instructions include PERIs that are not accepted, but not PERIs that are accepted and fail during transfers.

 Note: No guarantee can be given of the state of the core image after an instruction has been detected as illegal by hardware. In particular the instruction may have been partly obeyed. This danger does not apply in the case of extracodes other than CONT.

4 The core image is left ready to obey the next instruction in the natural sequence.

 This will be the case when the event is not generated by a particular instruction, as when the program's mill time allowance is exhausted, and also when an on-line peripheral fails after the PERI instruction has been accepted. In the latter case a number of instructions may be completed after the PERI is accepted and before the failure occurs, as in the case of a transfer failure.

When a program event has occurred, the address of the instruction that the core image is left ready to obey is held in word 8. The user can restart the program at this instruction by issuing a RESUME command with no parameters. If he wishes to restart the program at some other point this can be done by issuing a RESUME command with the address of the required restart point (relative to 0 rather than 20 as in the ENTER command) as the first parameter.

After certain MONITOR events, the user can either restart his program at the instruction after the monitored one by means of the RESUME command, or he can restart at the monitored instruction itself by means of the MONRESUME command. For example, after having monitored an attempt to rename a magnetic tape, the user could either resume without renaming the tape or carry on and rename it, perhaps after altering the PERI control area to specify a different name. The MONRESUME command inhibits monitoring of the first object program instruction obeyed after it has been issued.

THE IF COMMAND AND PROGRAM EVENTS

The IF command may have any of the following formats:

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

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

3 IF *condition, command*$_1$

where *command*$_1$ and *command*$_2$ may be any commands, including further IF commands, but may not be labelled, and where the condition parameter can be the name of one of the program event categories described above, for example:

 IF HALTED, . . .