```
IF DELETED, . . .
IF FAILED, . . .
IF MONITOR
```

To define the condition of an IF command more narrowly, the user can follow the category name with a character string; this string will be compared with the current program event message, provided that the message is of the specified category. The character string must be enclosed in parentheses, apostrophes, quotes, question marks or solidi. Note that if there are significant spaces in the comparison then apostrophes or quotes must be used. The following are examples of various forms of IF command with a program event category and character string as the condition:

```
IF HALTED (OK) , . . .
IF DELETED"PROGA RUN ENDED OK", . . .
IF FAILED (ILLEGAL:RESERVATION VIOLATION) , . . .
IF MONITOR (DISENGAGED,*CR0) , . . .
IF MONITOR (DISPLAY) , . . .
```

The condition will be found satisfied if the category of the program event message corresponds to the specified category and the first *n* characters of the program event message correspond to the characters in the string, where *n* is the number of characters in the string. Thus if it is probable that the first few characters of the program event message will serve to identify the event message uniquely it is only necessary to specify these first few characters. For example:

```
IF DELETED "PROGA", . . .
```

might be a satisfactory replacement for the second example given above.

A display output by DISP or DISTY extracode does not normally cause a program event but is set as the program's current display and may be examined when the job returns to command level at the end of the program run. The condition used to examine the contents of the current display is

IF DISPLAY *enclosed string*

where *enclosed string* is defined in the same way as for character strings specified in the above examples of the IF command. If, however, the user wishes to examine the text of the display as soon as it is output by the program, he must issue a MONITOR ON, DISPLAY command before ENTERing the program, so that the display will cause a program event. An IF DISPLAY command can then be included in the job description to test the contents of the current display when a program event occurs.

There is a further condition parameter for program events, which enables the user to discover which program member has caused the event. This has the form

IF MEMBER *number* , . . .

This condition may be combined with some other condition parameter to make the subsequent action depend on whether a particular event has occurred in a particular program member, for example:

```
IF MEMBER 3 AND HALTED (OK) , . . .
```

In general any number of IF conditions may be strung together to form a *compound* condition provided that they are connected by AND, OR or NOT.

The second parameter of the IF command indicates the action to be taken if the condition is satisfied. It may be any appropriate command including another IF command. The most common command parameter is the GO TO command. This causes a branch to be made to be a labelled command at the same command processor level, or, failing that, to a labelled command at some higher command processor level. The implications of multi-level jobs are dealt with in the section *Multi-level jobs* on page 31.

The following section consists of several examples of the use of the IF command to examine program event message and current displays.

Example 2/9

If a user wishes to use one set of IF commands to control a number of program runs, he must arrange his job description so that the command processor will read the same IF commands every time a program event occurs. This is done by issuing an unconditional GOTO command to branch to the same set of IF commands after each program event.

```
ENTER
GO TO 1
2 RESUME
1 IF FAILED, GO TO 3
IF DELETED, ENDJOB
IF HALTED, GO TO 2
3 ....
```

Example 2/10

A program stored in the file FRED may need a line printer. If the printer is needed the program will halt and output a message asking for the printer. If this happens the printer will be made on-line and the run will be resumed. The program has a postmortem routine which is entered if the program goes illegal.

```
LOAD FRED
ASSIGN *CR0, CARDATA
ASSIGN *CP0, RESULTS
ENTER 0
IF NOT HALTED (LINE PRINTER), GO TO 1
ONLINE *LP1, EXCEPTIONS
RESUME
1 IF FAILED, ENTER 1
ENDJOB
```

Example 2/11

A program is to operate on two files of data, one after the other. If the program reads to the end of FILEA, a FAILED program event will be generated with the message:

```
FILE *CR0 EXHAUSTED
```

If this happens FILEB will be assigned to this peripheral channel and the program will be resumed. If a program event of any other kind occurs, the job will be terminated.

```
LOAD FRED
ASSIGN *CR0, FILEA
ENTER 0
IF NOT FAIL (FILE *CR0), GO TO 1
ASSIGN *CR0, FILEB
RESUME
1
ENDJOB
```

Example 2/12

A program is organized to read from two card readers, one after the other. In this case the peripherals are to be on-line, therefore a MONITOR command must be used to detect the disengaging of the first reader.

```
LOAD FRED
ONLINE *CR0, ONLINEDATA1
MONITOR ON, DISENGAGED, *CR0
ENTER
```

```
IF NOT MONITOR (DISENGAGED, *CR0), ENDJOB
MONITOR OFF, DISENGAGED, *CR0
ONLINE *CR0, ONLINEDATA2
RESUME
ENDJOB
```

DISPLAYS

The message generated by the most recent DISP or DISTY extracode is set up as the *current display* of the job. In addition a copy of the display is sent to the OBJECT category of the monitoring file. Though these extracodes only cause program events if a MONITOR ON, DISPLAY command has been given, the contents of the current display can be used with IF or SETPARAM commands as soon as the job returns to command level.

In the same way as the current reply, or the current program event message, the current display is overwritten whenever a new display is generated. The current display is destroyed by all commands that load programs, but it is not destroyed by the ENTER and RESUME commands.

The two commands that reset the current display are DISPLAY and ANSWER.

The DISPLAY command

The message generated by a DISPLAY command differs from that generated by a DISP or DISTY extracode in that it can be directed to the operator's console as well as to the DISPLAY category of the monitoring file, and to the current display. The destination of the message is determined by a routing parameter in the DISPLAY command. The routing parameter can be set to any one of the following values:

Value	*Meaning*
0	Send to monitoring file
1	Send to monitoring file and to cluster console
2	Send to monitoring file and to central console
3	Send to monitoring file, central console, and cluster console

For jobs submitted centrally, values 1, 2 and 3 all have the same effect: the message is sent to the monitoring file and to the central operator's console. This is because the central console is implicitly defined as the cluster console for all jobs initiated centrally. For a description of peripheral clusters, see page 86.4.

Note that the output from a DISPLAY command is always sent to the job's monitoring file and set as the current display.

In background jobs the DISPLAY command is likely to be used mainly to output requests to the operator, for example

```
DISPLAY 2, PLEASE LOAD DOC FRED ON CR AND ENGAGE
```

The display message may be up to 40 characters long. In MOP, the DISPLAY command has further uses that are explained in Chapter 7.

The QUESTION and ANSWER commands

The QUESTION command may be used in a background job to send a message to the operator and to receive a reply (by means of the ANSWER command) that can be tested using the IF command.

The QUESTION command has a routing parameter which allows the same options as the DISPLAY command, but, unlike the DISPLAY command, the QUESTION command does not overwrite the current display. When a message has been output, the QUESTION command waits for an ANSWER command to be issued. The QUESTION command is not complete until an ANSWER command has been issued; the ANSWER command may thus be regarded as a facility within the QUESTION command. A user who intends to use QUESTION commands in background jobs must inform the operator beforehand of the kinds of answers that must be given to questions from his job. Thus the user can test the operator's answer, which is set up as the job's current display, by means of IF commands. In this way the user may discover such things as the state of on-line peripherals by directly communicating with the operator at run time.

Example 2/13

The user wishes to check that 2-ply line printer paper is set up before continuing his run. If it is not, the operator is asked to make available a line printer with the correct paper. Note that had 2-ply paper been a peripheral property, GEORGE would have asked the operator to make a suitable peripheral available when the command that required the peripheral was issued (a full description of peripheral properties and the use of the property system is given in Chapter 5 *Peripheral handling*).

```
1    QUESTION 1, LINE PRINTER WITH 2-PLY PAPER?
     IF DISPLAY (NO), GO TO 2
     IF DISPLAY (YES), GO TO 3
     DISPLAY 1, PLEASE ANSWER YES OR NO
     GO TO 1
2    DISPLAY 1, PLEASE PROVIDE PRINTER WITH 2-PLY
     GO TO 1
3    RESUME
```

In the above example, the command after the first QUESTION command will not be read until the operator has given an ANSWER command. If the operator does reply he should reply 'YES' or 'NO'. If he replies with something else he is told to answer 'YES' or 'NO' and the question is repeated.

Example 2/14

If no cards have been read in a job, the job is to be terminated. Otherwise the job continues at the next instruction of the program being run.

```
1 QUESTION 1, HAVE ANY CARDS BEEN READ?
IF DISPLAY (NO), GO TO 2
IF DISPLAY (YES), GO TO 3
DISPLAY 1, PLEASE ANSWER YES OR NO
GO TO 1
3 RESUME
2
ENDJOB
```

THE IF COMMAND AND THE CURRENT REPLY

As was explained in the section *The current reply,* each message that is sent to the monitoring file, except messages to the COMMANDS and ONLINE categories, is also set as the job's current reply until it is overwritten by the next message. The form of the IF command that can be used to test the current reply is

IF REPLY , ...

which is satisfied if there is a current reply, or

IF REPLY *enclosed string* . . . .

which is satisfied if the first *n* characters of the current reply are the same as the enclosed string specified in the IF command, where *n* is the number of characters in the enclosed string.

THE IF . . . ELSE COMMAND

A format of the IF command is available which allows alternative courses of action to be specified dependent on whether or not a condition is fulfilled. The format is:

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

where *command*$_1$ and *command*$_2$ may be any commands, including further IF commands.

It should be noted that many sequences of conditional commands may be expressed more concisely and executed more efficiently if the above format is used. Example 2/13 above, for instance, could be re-written as follows:

```
1       QUESTION 1,LINE PRINTER WITH 2-PLY PAPER?
        IF DISPLAY(YES),(GO TO 2)
        IF DISPLAY (NO),(DISPLAY 1, PLEASE PROVIDE PRINTER WITH 2-PLY)
        ELSE(DISPLAY 1,PLEASE ANSWER YES OR NO)
        GO TO 1
2       RESUME
```

Note:

Use of the above format may result in the command being too long to be held in one record. In order to carry a command over from one record to the next, the hyphen character (-) must be given as the last non-space character in the record (see Chapter 10). Care should be taken that the beginning of the continuation record does not duplicate a label used by the GOTO command, unless delimiters are used (see *Command delimiters,* page 5). In the example above, if 2-PLY started a new record, an error would occur.

PSEUDO PROGRAM EVENTS

The HALT, DELETE and FAIL commands are analogous to the DISPLAY command in their relation to program extracodes. They enable the user to generate pseudo program events of the categories HALTED, DELETED and FAILED, respectively. The program member number is always the current member number. The FAIL command can also be used to enter a program's own monitoring of illegals routine. The DELETE command also deletes the current core image.

The text parameter of the command is set up as the current program event message, overwriting the information stored by any previous program event or pseudo program event. This information may then be examined by an IF command, with a program event category optionally followed by a character string as its condition parameter.

Normally these commands are most used in MOP jobs, where the user might break-in on a program run and abandon the run with a DELETE, HALT or FAIL before continuing the job.

In a background job these commands are useful for changing the category of a program event to satisfy a particular condition. For example:

```
IF HALTED, FAIL TIME UP
IF FAILED (TIME UP), GO TO 1
```

EXAMINING AND ALTERING THE CORE IMAGE

The switch word

The user can alter specific bits of the switch word (word 30) of his current core image by means of the ON and OFF commands. The parameters of these commands indicate the bits (numbered 0 to 23) that are to be set 1 or 0, for example:

```
ON 1, 2, 3, 23
OFF 17
```

The switch word can be examined by means of the IF command and the conditions ON and OFF; for example:

```
IF ON 14, GO TO 1
IF OFF (2, 16), GO TO 2
```

Other facilities

Using the IF command, the ZERO, POSITIVE and NEGATIVE conditions provide a general facility for examining the value of any word in the core image. Numbers can be represented in a variety of ways in GEORGE commands, but in particular if a program location enclosed in square brackets is specified, the number in that location will be examined by GEORGE. Details of the formats of numbers are given in Chapter 10. Since the contents of program locations may be tested by IF commands the function of the IF OFF command given above could be performed by the following command:

```
IF ZERO ([30] & #10000200), GO TO 2
```

The contents of word 30 are obtained by [30] then a logical AND is performed with the specified octal number, the condition is satisfied if the result is zero.

An example of the power and flexibility of this facility is given below:

Example 2/15

The command in this example tests the value of a count in a PERI control area. It assumes that word 8 contains the address of the PERI instruction and takes no account of modification.

```
IF ZERO ([([[8]] & #77777) + 2] – 128), GO TO 4
```

[[8]] gives the PERI instruction (the contents of the contents of word 8).

& #77777 isolates the part of the instruction that contains the address of the control area.

+ 2 adds two to this address to give the address of the count word.

[......] gives the contents of this address, that is the value of the count.

– 128 subtracts 128 from the count.

The condition is satisfied if the count is 128.

The PRINT and ALTER commands

The PRINT command enables the user to output one or more regions of his core image to a named file or to the monitoring file and, in the case of MOP jobs, to the MOP terminal. The output is sent to the POSTMORT category of the monitoring file. The facility of printing to a named file is useful for listing programs. The facility of printing the whole core image is useful where the program size is unknown and the user wishes to avoid a command error.

Note that in GEORGE 4 the PRINT command will not access or print the unused areas of a sparse program.

The ALTER command enables the user to reset the value of a word of the core image. It is frequently used when the contents of a word must be reset before a program is entered or resumed.

In GEORGE 4 the ALTER command has an additional parameter, the character string PURE, which enables words in a pure area to be reset. However, GEORGE 4 will not allow the alteration of a location in a pure area which can be shared by a different activation of the program and in this case the user will have to load his own private copy of the program. This can be achieved by specifying the character string PRIVATE as a parameter to the LOAD command.

The ALTER command is most frequently used in MOP jobs. In this case the user outputs regions of his core image to the terminal by means of a PRINT command and can then make the necessary changes to the core image by issuing ALTER commands.

Saving programs

The SAVE command enables the user to preserve a core image in a *saved file.* The command creates or overwrites a basic peripheral file and writes to it a binary dump of the current core image in GRAPHIC or disc format together with any other information required to remember the state of the core image.

The format is:

SAVE *filename,device type*

where *device type* may be *CP or *DA and is optional. If it is omitted, *DA is assumed. It is recommended that disc files be used where possible, since they are more efficient. For further details see the specification of the SAVE command.

When a program is saved, any on-line peripherals or files connected to the program are disconnected and the current state of these is lost. A list of the peripheral channels involved will be sent to the monitoring file system (COMMENT category). SAVE does not delete the current core image nor alter it in any other way.

A saved program can be reloaded by the LOAD command; any peripherals may be restored by means of the ASSIGN or ONLINE commands.

Should the user wish to load and enter a saved program without restoring any peripherals, he can carry out both operations by means of a single RESUME command. In this case the RESUME command has a file name parameter as well as an optional entry point parameter.

A postmortem of a saved program can be obtained by use of the LOAD command, followed by PRINT. Note that this use of the LOAD command will destroy the current core image.

Notes:

1 If the consolidator XPCK, or any other program which relies on accumulators being carried forward from a program deleted by DELTY, LOAD or FIND is SAVEd, it will not run correctly on being run from the SAVEd file; XPCK will probably HALT LD on entry. XPCK has a special binary format and must not be dumped by the Executive dump facility.

SAVE stores a value for each word of the program being SAVEd, except for sufficiently large blocks of zeros (16 or more). Because word 8 is always non-zero, the accumulators are always stored, whether zero or non-zero.

LOAD sets the accumulators of the LOADed program to the values in the file; if there are no values in the file, then the carried-forward accumulators' values are used; if the carried-forward or file values are not present, the accumulators of the LOADed program are set to zero. Since there are always values stored in the SAVEd file for words 0 to 15 accumulators will never be carried forward if loading is carried out from a SAVEd file.

2 Because of the above fact, and also because the overlay file is disconnected if SAVE is used with overlay programs, SAVE is not recommended for use with ICL library programs. The macro DISCPROG (page 416) should be used for manipulation of these items.

Example 2/16

```
ENTER
FAILED ILLEGAL INSTRUCTION LDX 0 0(1) N(M) = 89432
SAVE SAVEFILE
LINGO SECONDPROG
output from compiler
SAVE SECONDFILE
LOAD SAVEFILE
PRINT , REGION ((0,7)), REGION ((500, 580))
```

In this example, a MOP user's program goes illegal and so he SAVEs the core image and compiles another program. During the compilation he considers the error in the first program and decides he requires more information to correct it. When the compilation has ended he therefore issues a LOAD and PRINT sequence to write the contents of the accumulators and words 500 to 580 of his failed program to the monitoring file. Note that if he had not issued a SAVE command before LOAD he would have lost the compiled version of his second program, since LOAD deletes the current core image.

Command Errors

When GEORGE detects a command error it abandons the command and sends a command error message to the COMERR category in the monitoring file of the job. The various command error messages that may be given are listed in Chapter 11 for general command error messages and Chapter 12 for messages peculiar to a particular command. The action taken by GEORGE on finding a command error is described in detail in Chapter 12.

If the user wishes GEORGE to take some specific action on finding a command error he must use the version of the WHENEVER command that monitors command errors (see below).

Any user who writes job descriptions that rely for "switching" on particular error actions or error messages should note that error actions and messages are liable to change in future marks of GEORGE.

The WHENEVER command

The user can include WHENEVER commands in his job description to tell GEORGE what action to take whenever a specified type of event occurs during the running of the job. The types of event that may be specified are: command error, break-in, job time exceeded and system close-down. The WHENEVER commands that correspond to these types of event are: WHENEVER COMMAND ERROR, WHENEVER BREAKIN, WHENEVER JOBTIME EXCEEDED and WHENEVER FINISH.

Each WHENEVER command overwrites the information stored by any previous WHENEVER command of the same type at the same command processor level. Thus if a command error occurs, GEORGE will obey the last

WHENEVER COMMAND ERROR command to be read at the current command processor level.

Once a WHENEVER condition has been satisfied it will have no effect if it is encountered again in the course of the job.

COMMAND ERROR

The format of the WHENEVER command to deal with command errors is:

WHENEVER COMMAND ERROR, *command*

or

WHENEVER COMERR, *command*

For the use of WHENEVER COMMAND ERROR commands in multi-level jobs see the section *Multi-level jobs* on page 32.

Example 2/17

```
WHENEVER COMMAND ERROR, GO TO 1
LOAD PRINTPROG
ASSIGN *CR1, MYFILE
ASSIGN *CP1, RESULTS
ENTER
WHENEVER COMMAND ERROR, ENDJOB
IF NOT DELETED, PRINT 0(30)
1 DISPLAY 0, ERROR IN SECTION 1
LOAD OTHERPROG
```

Note that the second WHENEVER command is effective only if there is no error in the four commands that follow the first WHENEVER command. If there is an error in one of these four commands, control will pass to the command labelled 1, and the second WHENEVER command will never be read.

BREAK-IN

The format of the WHENEVER command to deal with break-ins is:

WHENEVER BREAKIN, *command*

This command is provided for use in macro definition files (including job description files) that may be issued from a MOP console. If the user breaks in while the macro is being executed, the action specified by the second parameter of the WHENEVER is taken. It should be noted that control is not passed back to the user at the MOP console, nor is the context of the job changed to BREAK-IN. This means that the user cannot terminate the break-in and continue his job from the point at which he broke-in, as he can do when no WHENEVER BREAK-IN has been issued.

JOBTIME EXCEEDED

The format of the WHENEVER command to deal with jobs exceeding their jobtime is:

WHENEVER JOBTIME EXCEEDED, *command*

This command specifies the tidying up action that the job needs to take before it is abandoned. If the command is reached, the job is allowed a further 10 mill seconds for its tidying up action, and if it does not terminate within this time it will be abandoned immediately.

SYSTEM CLOSE-DOWN

The format of the WHENEVER command to deal with the system closing down is:

WHENEVER FINISH, *command*

This command specifies what action should be taken if the operator issues a FINISH command to close down the system while the job is running.

Job security

Each user is able to impose a security restriction on the running of jobs in his name by means of the SECURITY command (for a full specification of this command see Chapter 12). The levels of security possible are HIGH, NORMAL and LOW. Users are initially set up at NORMAL security. To change to HIGH security the user must issue a SECURITY HIGH command, and likewise to change to low, a SECURITY LOW command is needed.

When the user is at NORMAL security he has full security when working from a MOP terminal. That is, INPUT commands may not be issued in the NO USER context from the MOP terminal and JOB, RUNJOB and CONNECT commands issued in the NO USER AND MOP context will call for the user's password in the same way as a LOGIN command. However JOB, RUNJOB and INPUT are allowed in NO USER AND READER context.

A user who is at HIGH security has full security both for background and on-line working. Only the INPUT command is allowed in the NO USER AND READER context, and the MOP restrictions are the same as for a user at NORMAL security.

A user who is at LOW security has no security protection. INPUT, JOB and RUNJOB are all allowed in NO USER contexts for both on-line and background working. Also the LOGIN command does not ask for the user's password so there is no security check on USER context MOP working. The LOW security facility is provided for the benefit of installations which do not have security problems and therefore do not need to impose on their users the chore of typing a password every time they LOGIN.

Real time programs

Any program that requires to operate in real time must be kept permanently in core. This requirement must be specified to GEORGE so that the program is never swapped out. Since this arrangement will cause relative inefficiency in GEORGE's management of the core store, real time working is a facility which should be strictly controlled. Before any user may run a real time program GEORGE will check that the user has sufficient REALTIME budget (see page 154). Thus the installation manager may restrict the number of users who may run real time programs. The way in which a user declares that his job will from a certain point have real time requirements is to issue a

```
REALTIME ON
```

command. The user may also specify, by means of a parameter to this command, a member or members of his program that he wishes to have high, fixed priority, above that of GEORGE. When a job no longer has real time needs a

```
REALTIME OFF
```

command should be issued. For further details see the specification of the REALTIME command.

In GEORGE 4 real time programs are run with all of their pages in core and so do no page turning. When real time is switched on a program's quota will be fixed and set equal to its size. Sparse programs are not allowed to be real time.

Restrictions on the size of the core image

The term "size" or "core size" refers to the number of words of store (GEORGE 3) or virtual store (GEORGE 4) that are accessible to a core image. This is always a multiple of 64 (GEORGE 3) or 1024 (GEORGE 4).

The size of each core image must be restricted both to increase system efficiency and to stop programs that are out of control. The following limits are applied to core size requests in the LOAD, CORE and SIZE commands and the GIVE/4 instructions:

1 The installation parameter COREOBJECT. This is the maximum size that can be given to an object program. If COREOBJECT is reduced, core images already greater than the size specified by the new COREOBJECT will be allowed to continue at their original size but will not be allowed to increase. In GEORGE 3 COREOBJECT is limited by the core size of the machine. In GEORGE 4 COREOBJECT is limited by the amount of backing store reserved to hold object programs: the corresponding restriction on the amount of core an object program may occupy simultaneously is the installation parameter MAXQUOTA (see 2, below).

2 The installation parameter MAXQUOTA (GEORGE 4 only). This is the maximum quota that will be given to an object program.

3 The MAXSIZE command. This may be issued at any stage of a job and limits the size of core images within the job. Any subsequent attempt to load a program with a core size greater than that specified by MAXSIZE will be treated as an error, and if a CORE or SIZE command or GIVE/4 extracode requests a core size greater than the value in the MAXSIZE command, the MAXSIZE value will be taken as the core size requested.

4 The installation parameter SIZEDEFAULT. This specifies the maximum core size a user can request by the GIVE/4 extracode, or the commands LOAD, CORE and SIZE in a job which does not include a MAXSIZE command. If the standard High Level Scheduler is not in use (see page 149) SIZEDEFAULT only affects GIVE/4 extracodes. The SIZEDEFAULT parameter cannot be set to a higher value than COREOBJECT.

5 The MAXQUOTA command (GEORGE 4 only). This may be issued at any stage of a job and limits the size of a core image's quota within a job (see MAXQUOTA in Chapter 12, page 187).

Note that if the user wishes to determine the value of one of these installation parameters or of his current core size he may do so by means of the SETPARAM command (see page 364).

MULTI-LEVEL JOBS

All background jobs are initiated by the name of the file holding the job description being issued as a macro command, thereby causing all the commands in the file to be obeyed. Thus all background jobs are, in effect, macros. (Note that job names must therefore conform to the same restrictions as macro names, that is they must be local names that do not contain spaces.) However, in all of the preceding examples of job descriptions every item in the job description file was completely specified when the file was set up. This section describes the facility of writing macros in the GEORGE command language that have only formal parameters assigned to some items of the job description when the job is stored. The actual parameters are then inserted at run time, so that the same basic job description may be used for several different jobs simply by varying the parameters given at run time. System macros are written in this way, but in this section emphasis is given to user macros and program issued commands.

Writing user macros

PARAMETER SUBSTITUTION

A job description file is simply one case of a macro definition file. Since job description files have been described earlier in the chapter they are used here to explain parameter substitution which is a common technique for writing all macros.

When the user writes a job description, he does not have to specify all the commands that constitute the job description in full. He may wish to leave certain commands or parts of commands undefined, so that he will be able to use the job description with a number of different sets of values. To do this he must write the parts that are to be replaced as *parameter identifiers*. The standard notation for representing a parameter identifier is the percentage sign (%) followed by a letter of the alphabet between A and X (inclusive). For a more detailed description, see *Parameter identifiers,* page 160.

When the user initiates a job with a JOB or RUNJOB command, he can give actual values to the parameter identifiers in the job description by means of special parameters of the JOB or RUNJOB command, known as *job parameters*.

These are optional parameters; if they are included they must follow the normal command parameters and must be preceded by character string PARAM. When a parameter identifier is read by the command processor it will be replaced by the appropriate job parameter in the command at the level above (JOB or RUNJOB). %A will be replaced by the first job parameter, %B by the second, and so on. Thus the command:

```
JOB MYJOB,:SMITH,PARAM(OBJECT,8,12000)
```

or

```
RUNJOB MYJOB,:SMITH,JDFILE,PARAM(OBJECT,8,12000)
```

will cause %A in the job description to be replaced by OBJECT, %B by 8 and %C by 12000. If the job description contains the commands

```
LOAD %A
CORE %C
ENTER %B
```

the JOB or RUNJOB command will implement the commands

```
LOAD OBJECT
CORE12000
ENTER 8
```

%Y and %Z are reserved parameter identifiers which are replaced by the current job name and the proper user name (minus the prefix, colon) respectively.

There are two alternative notations for representing a parameter identifier which enable the user to include part of a parameter at the level above within a command in his job description. These are:

1 *% enclosed string*

2 *%n enclosed string*

where *n* is a decimal number less than 25 (see Chapter 10 for details of the formats of parameter identifiers and enclosed strings).

The parameter identifiers identify the first (format 1) or the *n*th (format 2) job parameter at the level above beginning with the specified character string. When the command processor reads an identifier with one of these formats it finds the parameter that is being referred to at the level above and replaces the identifier by the part of this parameter that follows the specified string. Thus, if a stored job description contains the commands

```
LOAD %(PROG)
CORE %(CORE)
ENTER %(ENT)
```

then the command

```
RUNJOB   MYJOB,:SMITH,JDFILE,PARAM(PROGOBJECT,ENT8, CORE12000)
```

or

```
RUNJOB  MYJOB,:SMITH,JDFILE,PARAM(CORE12000,PROGOBJECT,ENT8)
```

will implement the commands

```
LOAD     OBJECT
CORE     12000
ENTER    8
```

Note that the character string given in the parameter identifier is not included in the actual parameter.

There is a further form of parameter identifier which enables the user to include in his job description any of the values which may be specified as the second parameter to a SETPARAM command. A full list of these values is given in the specification SETPARAM on page 364. They include the current program event message, the current display, the current value of any installation parameter, and the date and time. The format of this type of parameter identifier is:

%; *second parameter of SETPARAM*;

Thus if the user issues the command:

```
DISPLAY 0, %;TIME;
```

the time in the form HH.MM.SS is sent to the monitoring file and set as the job's current display.

For further details see *Parameter identifiers*, page 160.

These facilities enable the user to run a number of different jobs using only one job description. All the user has to do to initiate a different job is issue a JOB or RUNJOB command with different parameters. It should be noted that, in the case of JOB, the facility can only be used if the JOB command is actually replaced in a stack of cards that is being used to input the job description to the system. Normally a user who wishes to vary certain values in a job description will file it in a permanent file and use a series of RUNJOB commands to initiate the different jobs. There would be little point in using the job description to run a series of once-only jobs initiated by different JOB commands, unless the user was short of space in the filestore.

The facility to substitute values in a job description file is one instance of a general facility whereby commands at any command processor level other than level zero may substitute parameters of the command at the level above within themselves. The special parameter identifiers %Y and %Z are also allowed at level zero, but %Y is forbidden in the NO USER context since in this case there is no job name. If %Z is used in NO USER context it must be within an embedded INPUT command; in this case the user name parameter of the embedded INPUT is substituted.

A parameter identifier may represent a single character of a parameter, a whole parameter, a command verb or a whole command, subject to the following restrictions:

1 An identifier is not allowed in place of a label at the beginning of a command, though it is allowed as the label parameter to a GO TO command, for example

```
IF FAILED, GO TO %A
```

This is because GOTO, for reasons of efficiency, does not do parameter substitution when searching the job description for a label.

2 An identifier may stand for a whole command only if the command contains no commas. Thus, if %A is an identifier standing for a command, it may be replaced by

```
TIME 2SECS
```

but not by

```
ASSIGN *CR0,DATAFILE
```

The reason for this restriction is that the comma would be regarded as a normal separator in the command at the higher level, so that %A would be replaced by ASSIGN *CR0 and %B by DATAFILE. (It is, however, possible to avoid this restriction by use of the SETPARAM command. See page 36.)

3 If a parameter is null or absent, substitution still takes place. The identifier is erased from the command and a null parameter is put in its place.

4 %% is treated as %.

THE MACDEF COMMAND

The facility described above shows a job description file to be simply one type of macro definition file. A job description file differs from the standard macro definition file only in that the macro command in question is issued internally by JOB or RUNJOB and is always issued at command processor level zero. Standard macro commands are normally issued at some lower level unless they come form a MOP terminal.

A macro definition file is normally created by means of a MACDEF command issued from a job description file, from another macro definition file or from a MOP terminal. The format of the command is

MACDEF *filename*, *terminator*

where the *file name* must be a local name without internal spaces (so that it can be issued as a macro command). The *terminator* parameter is optional, as in the INPUT command. If this parameter is omitted, the terminator ENDMAC (or ENDM) is assumed. The commands that follow MACDEF up to the terminator are stored as a file in CARDS mode.

Example 2/18

```
MACDEF SUBJOB
WHENEVER COMMAND ERROR, EXIT
LINGO *CR%A, BIN%B, ER1
LOAD %B
%C %D, %E
%C %F, %G
TIME %(TIME)
ENTER %H
IF FAILED, SUBSUBJOB
SAVE %I
1
ENDM
```

The user can implement this macro by issuing the name of the macro definition file as a command, with the values that are to replace %A, %B etc. as its parameters:

```
SUBJOB SOURCE1, OBJECT1, ASSIGN, *CR0, DATA1, *CP0, OUT1,, SAVEDFILE1, TIME 2SECS
```

This command will implement the following series of commands:

```
WHENEVER COMMAND ERROR, EXIT
LINGO *CR SOURCE1, BIN OBJECT1, ER1
LOAD OBJECT1
ASSIGN *CR0,DATA1
ASSIGN *CP0,OUT1
TIME 2SECS
ENTER
IF FAILED, SUBSUBJOB
SAVE SAVEDFILE1
1
```

Note: the IF FAILED command specifies that in the event of a program failure, the macro command SUBSUBJOB is to be obeyed.

This means that the example contains three possible command processor levels:

1 The level at which SUBJOB is issued,

2 The level at which the commands that constitute SUBJOB are obeyed, and at which SUBSUBJOB is issued if a program failure occurs,

3 The level at which the commands that constitute SUBSUBJOB will be obeyed if the program failure occurs.

If there is a program failure, the macro SUBSUBJOB will be obeyed. When this has been completed, the command processor will return to the macro level above, to the command after 'IF FAILED, SUBSUBJOB'. When the commands at this level have been carried out, it will return to the level above, to the command after SUBJOB. When the commands at any level have been obeyed, that level will be erased. Note that the command source is not erased when the command processor level at which commands from this source are being issued is erased.

Note on the INPUT command

Though the MACDEF command is normally used to create macro definition files, the INPUT command will perform the same function. The user can create a file of commands by means of an INPUT command. He can then issue the name of the file as a macro, with the required macro parameters.

THE SETPARAM COMMAND

When a new command processor level is created, a block of 24 variable length parameters is passed down from the level above. These parameter locations are referred to in parameter substitution by the identifiers %A to %X. When the parameter block is passed down some of the parameters may not be set, but all the locations are available for parameter substitution.

Any of these locations may be reset from within the macro by means of a SETPARAM command. When the command processor level at which the SETPARAM command was issued is destroyed, the block of parameters available at that level is also destroyed. Thus with one form of SETPARAM command only parameters within the macro are reset. There is, however, an alternative method of using SETPARAM; in this case the location referred to at the current level itself contains a parameter identifier which refers to a location one level above. Thus by chaining such identifiers it is possible for a macro to return a value to any command processor level above it.

The general format of the SETPARAM command is

SETPARAM *parameter identifier, new value*

There are four formats for the parameter identifier, two each for call by name and for call by value techniques. The *new value* may be a fixed value, a value taken from the current display, a program event message, or message to the monitoring file system, information from a directory entry, or a value known to the system, such as the time or date, the current jobname or username, the current size of the core image, or the value of a specified installation parameter. For details of the formats see the SETPARAM Command in Chapter 12.

Example 2/19

In the simplest case, the SETPARAM command can be used to change the value in an existing location and then re-run a series of commands. The previous example can be rewritten so that in the event of a program failure, different input and output files will be assigned to the program and it will then be run again.

```
MACDEF SUBJOB
LINGO *CR %A, BIN %B, ER 1SUB
LOAD %B
2SUB ASSIGN %C, %D
ASSIGN %E, %F
TIME % (TIME)
ENTER %G
IF NOT FAILED, EXIT
IF STRING (%D) = (DATA2), GO TO 1SUB
SETPARAM D, "DATA2"
SETPARAM F, "OUT2"
GO TO 2SUB
1SUB
ENDM
```

This macro can be run with the command

```
SUBJOB SOURCE1, OBJECT1, *CR0, DATA1, *CP0, OUT1,,TIME 2SECS
```

Example 2/20

The SETPARAM command can be used in conjunction with IF to make a parameter value dependent on the current program event message or the current display. For example, the user might wish to run a program which involves the use of a card reader and a paper tape reader. When the program has finished using either of these peripherals it will obey a SUSTY extracode, which will generate the message

```
IVE FINISHED WITH *CR0
```

or

```
IVE FINISHED WITH *TR0
```

When this happens the user wishes to release the peripheral in question and then resume the program. By means of SETPARAM he can name a parameter identifier (%A, %B, %C, etc.) and set its value as part of the SUSTY message. He can then issue a RELEASE command with this identifier as part or all of the peripheral name specified in RELEASE.

```
  ENTER
  GO TO 1
3 RESUME
1 IF FAILED OR DELETED, GO TO 2
  IF HALTED (IVE), SETPARAM A, MESSAGE (20)
  RELEASE *%AR0
  GO TO 3
2 ..........
```

The SETPARAM command sets the twentieth character of the program event message as the value of %A.

Example 2/21

A program stored in a file FRED is to be run. It may halt with a message starting with the characters LOAD. In this case the next eight characters are the name of a file containing another program to be run.

```
LOAD FRED
ENTER 0
IF NOT HALTED "LOAD", GO TO 1
SETPARAM A, MESSAGE (5,12)
LOAD %A
ENTER 0
1
ENDJOB
```

The SETPARAM command sets the fifth to twelfth characters inclusive of the current program event message as the value of %A.

Example 2/22

A program contains a number of DISTY extracodes. It is required that display messages beginning with OP be sent to the operator's console.

```
LOAD JOHN
MONITOR ON, DISPLAY
ENTER 0
GO TO 1
2 IF NOT DISPLAY 'OP', GO TO 3
SETPARAM A, DISPLAY
```

```
DISPLAY 1, %A
3 RESUME
1 IF DISPLAY, GO TO 2
ENDJOB
```

The SETPARAM command sets the whole of the current display as the value of %A.

Example 2/23

This example shows how SETPARAM may be used to overcome the second restriction on parameter substitution mentioned on page 32, that is that a parameter identifier may not represent a command containing commas.

A user has a macro BERYL in which a command is represented by %A. In order to substitute the command ASSIGN *CR0,DATAFILE for %A, the user must call the macro as follows:

```
BERYL (ASSIGN *CR0,DATAFILE)
```

This will cause the character string (ASSIGN *CR0,DATAFILE) to be substituted for %A in the macro. However, the brackets must be removed before the command can be obeyed. This can be done by including a SETPARAM command before %A in the macro definition. BERYL then contains:

```
SETPARAM A,%A
%A
```

The SETPARAM command expands to SETPARAM A, (ASSIGN *CR0, DATAFILE). The brackets are taken as the delimiters of an enclosed string and so %A is set to the character string ASSIGN *CR0, DATAFILE. When %A is issued, the ASSIGN command is obeyed.

LABELS IN MULTI-LEVEL JOB DESCRIPTIONS

When the command processor reads a label in a GOTO command, it searches through the job description file or macro definition file from which the GOTO was issued, starting from the command after GOTO. If it reaches the end of the file without finding a matching label, it starts searching again from the beginning of the file and continues up to the point at which the GOTO was issued. If it does not succeed in finding the label, it will search the file (if any) at the command processor level above, in the same way, starting from the command after the macro command. This process may need to be repeated for each macro level up to level one of a background job (this will be the job description file). If in its search the command processor encounters a program that has issued a command, or a built-in GEORGE command that has issued a command, the search is terminated since the program or built-in command cannot logically be searched for a label. If the matching label is found, the job will continue from the command bearing the label; otherwise a command error is reported. The action of GEORGE on finding a command error is described in Chapter 11.

It is important to bear in mind that the lowest command processor level in existence at any given time is always the level at which commands are currently being issued. A new command processor level is created only when a command is issued at a lower level, and as soon as the command processor returns to a higher level, the lower level is erased. This point is of particular relevance to branches within a job. A GO TO command can never cause the command processor to branch to a labelled command at a lower command processor level than the GO TO command since no lower command processor level is in existence when the GO TO command is issued.

Because of the way GO TO commands are obeyed, it is essential for the user to use labels in commands in a disciplined way.

Labels within a macro should conform to a consistent naming system, so that the danger of unintentional branches outside the macro is kept to a minimum. For example, the user might decide that all labels within a macro should consist of a digit followed by the name of the macro, for example 1SUBJOB, 2SUBJOB, 3SUBJOB etc. When editing a job description file or macro definition file, the user must take care not to alter labels that may be referred to in macros at lower command processor levels.

THE WHENEVER COMMAND ERROR COMMAND IN MULTI-LEVEL JOBS

The macro definition in Example 2/18 on page 33 contains a WHENEVER COMMAND ERROR command. This will be effective for command errors in commands that follow it in the macro definition file, but will not be effective for a command error detected later at some higher command processor level.

In general, a WHENEVER COMMAND ERROR command will be obeyed if a command error is detected in a command that follows it at the same level, though it will be superseded by a subsequent WHENEVER COMMAND

ERROR command at the same level. A WHENEVER COMMAND ERROR command will also apply to command errors detected at a lower level, though it will be superseded by a WHENEVER command that is issued at a lower level. Consider the Example 2/18 on page 33 where there is a macro SUBSUBJOB issued within the macro SUBJOB. If there is no WHENEVER COMMAND ERROR command in the macro definition file SUBSUBJOB, the WHENEVER COMMAND ERROR command in SUBJOB will apply to errors in SUBSUBJOB. If, however, the macro SUBSUBJOB did contain a WHENEVER COMMAND ERROR command this would take precedence in the event of errors in SUBSUBJOB. In this case when the macro SUBSUBJOB was completed, the job would return to the level above and the level above and the lower level would be erased, thus any WHENEVER COMMAND ERROR commands within SUBSUBJOB could not affect commands outside this macro.

A WHENEVER COMMAND ERROR command is initially analysed by the command processor when it is set as the current WHENEVER for command errors in the job. If a command error occurs and the command processor goes to the WHENEVER COMMAND ERROR command, it analyses the command a second time before obeying it. This peculiarity of the WHENEVER command makes it possible to use a parameter identifier in a WHENEVER command to call a parameter location by name rather than by value. In this case the identifier must be preceded by two percentage signs (instead of none as in the SETPARAM command), for example:

```
WHENEVER COMMAND ERROR, GO TO %%A
```

As was explained earlier, %% is treated by the command processor as %, so on the first pass the command processor analyses the WHENEVER command and reduces it to

```
WHENEVER COMMAND ERROR, GO TO %A
```

Subsequently, if a command error occurs, the command will be re-analysed, %A will be treated as a call by value and will be replaced by the current value of %A before the command is obeyed.

Example 2/24

```
MACDEF FRED
WHENEVER COMERR, GO TO %%A
--------
```
Further job description commands
```
--------
SETPARAM A, "2"
--------
```
Further job description commands
```
--------
SETPARAM A, "3"
--------
ENDMAC
```

On the first pass the command processor analyses %%A in the WHENEVER and reduces it to %A. If an error occurs between WHENEVER and the first SETPARAM, the command processor branches to the WHENEVER, re-analyses the command, substituting the first parameter of FRED, and obeys the resulting GO TO. Similarly with commands between the two SETPARAMS and commands after the second SETPARAM, except that in each case the label will have a different value because of the effect of SETPARAM.

EXAMINING THE PARAMETERS OF A JOB DESCRIPTION OR MACRO

There are three conditions of the IF command that can be used to make the action to be taken in a job description or macro dependent on the value of a parameter. These are the STRING, PRESENT and ABSENT conditions. The STRING condition enables the user to compare a character string with a parameter given by the level above.

The PRESENT and ABSENT conditions enable him to test whether a parameter with a particular value was given by the level above.

Simple switching

It is possible to write a macro or job description so that it allows one of two courses of action to be taken, depending on the value of a switch parameter.

Example 2/25

```
MACDEF FRED
IF STRING (%A) = (SIMPLE), GO TO 1FRED
IF STRING (%A) = (COMPLEX), GO TO 2FRED
GO TO 3FRED
1FRED
------------
------------
GO TO 3FRED
2FRED
------------
------------
3FRED
ENDM
```

The first parameter of the macro FRED may be either SIMPLE or COMPLEX. For each value of the parameter, a different series of commands will be obeyed.

Default parameters

By using the STRING condition, it is possible to write a macro or job description so that a default value will be assigned to a parameter that is given as null or absent by the level above.

Example 2/26

The third parameter of the macro FRED is to be a number giving the mill time in seconds for which a program will be allowed to run. If the parameter is null or absent, the program will be allowed to run for five seconds. This result can be obtained in either of the following ways.

```
MACDEF FRED
------------
IF NOT STRING (%C) = ( ), TIME %CSECS
IF STRING (%C) = ( ) , TIME 5SECS
------------
ENDM
```

or

```
MACDEF FRED
------------
IF STRING (%C) = ( ) , SETPARAM C, (5)
TIME∇%CSECS
------------
ENDM
```

Alternatively, the time might be specified by an optional parameter of the form

TIME *n*SECS

The parameters of the macro could then be given in any order.

```
MACDEF FRED
------------
IF ABSENT (TIME), TIME 5SECS
IF PRESENT (TIME), TIME %(TIME)
```

Altering the core image

A parameter of a macro or job description can be used in a straightforward way to set bits of a switch word, for example

```
ON %A
```

or

```
ALTER 100, %A
```

A parameter might also be used to test which bits of a switch word are already set, for example

```
IF ZERO(([100] & %A) – %A) , . . . . . .
```

Loops in job descriptions

It may be necessary to go through part of a job description a set number of times. This can be done by means of a *recursive macro*, that is, a macro that is issued *within* itself a number of times. Each time the macro is issued, a new command processor level is created. When the macro has been obeyed the specified number of times or some condition has been satisfied, the macro will be terminated at that level, the level will be erased and the command processor will return to the macro at the next higher level. When this macro is terminated, its level will in turn be erased, and so on up to the job description file.

Example 2/27

The macro below performs some operation until a certain condition is satisfied, up to a maximum number of times. This number is given by the number of Z's in the first parameter of the macro. DISPLAY commands are used to record whether the condition is satisfied or not.

```
MACDEF FRED
IF STRING (%A) > (Z), GO TO 1FRED
DISPLAY 0, ERROR
EXIT
1FRED
------------
IF condition, GO TO 2FRED
FRED %(Z)
EXIT
2FRED DISPLAY 0, OK
ENDM
```

Note that the command FRED %(Z) uses the facility whereby the character string specified in the parameter identifier is not included in the actual parameter that replaces it. This means that each time the command FRED %(Z) is obeyed, the parameter of FRED will contain one less Z. Eventually it will consist of a null string and the condition of the IF STRING command will not be satisfied.

(Note that '>' in the IF STRING command means 'if the strings match up to the end of the second string'. Thus the command

```
FRED ZZZZZ
```

will allow the operation to be performed up to five times.

The operation described above could be performed equally well without the use of a recursive macro. In the macro below, the first parameter is the maximum number of times the operation may be performed. The optional second parameter is a label to jump to if the condition is not satisfied.

```
MACDEF FRED
1FRED IF ZERO %A, GO TO 3FRED
```

```
            ------------
            ------------
            IF condition, EXIT
            SETPARAM A, VALUE (%A – 1)
            GO TO 1FRED
            3FRED IF NOT STRING (%B) = ( ), GO TO %B
            ENDM
```

If the first parameter of FRED is 5, the SETPARAM command will reset it to 4 at the first pass, 3 at the second pass and so on.

This second method must be used if it is desired to repeat the operation more than *n*–1 times, where *n* is the maximum number of command processor levels permitted. Initially *n* will be set to 25.

GENERATING JOBS WITHIN JOBS

It has been shown that the user can organise his job as a hierarchy of macros. It has also been explained that the JOB and RUNJOB commands issue macros internally at command processor level zero. Since RUNJOB (but not JOB) can be issued in any context and at any command processor level, it follows that instead of organising a complex operations as a number of macros within a single job, the user can arrange the operation as a hierarchy of jobs. Each job in the hierarchy, apart from the master job, can be initiated from within another job by a RUNJOB command; and, as with macros, parameter values can be passed down through the job hierarchy by means of the job parameters of the RUNJOB command.

The significant difference between this method of operation and the normal one is that as soon as a job has been initiated it runs independently of other jobs in the system. Thus, scheduling considerations permitting, a number of interrelated jobs can be run at the same time by means of this technique. Within a single job, on the other hand, only one operation can in general be performed at any one time, so the macros of a job are obeyed in sequence.

This system should not be used if the system is heavily loaded with jobs, since new jobs can only be accepted up to the limit set by the JOBLIMIT installation parameter (see page 150).

See also the section *Command issuer, PERI modes 11, 12 and 13,* page 43.

THE IF ... ELSE COMMAND

A format of the IF command is available which allows alternative courses of action to be specified dependent on whether or not a condition is fulfilled. The format is:

IF *condition, (*$command_1$*)* ELSE *(*$command_2$*)*

where $command_1$ and $command_2$ may be any commands including further IF commands.

It should be noted that many sequences of conditional commands may be expressed more concisely and executed more efficiently if the above format is used. This applies to many of the examples in this section.

Command issuing programs

Programs can issue commands in two distinct ways by issuing PERI type 60 (command issuer) instructions of various modes.

The first method, using modes 1, 2 and 3, allows commands to be issued and obeyed as part of the current job, thus the commands issued are subject to certain contextual restrictions.

The second method, using modes 11, 12 and 13 allows the program to issue commands which are obeyed as a separate job. By using a combination of these modes, a program can simulate a MOP terminal and therefore the only restrictions that apply to the commands issued are the usual ones for MOP jobs.

More than one command issuer may be connected to a program but the two sets of modes may not be mixed on one channel.

Command issuer, PERI type 60 modes 1, 2 and 3

These modes may only be issued if the command issuer has not previously been connected to the program by an ONLINE command, otherwise they are illegal.

Mode 1 of the PERI type 60 instruction writes characters to an area accessible to the command processor and returns control to the command processor. The characters are interpreted by the command processor as a command of the job; at the command processor level below that at which the entering command (ENTER or RESUME) was obeyed. Details about PERI control areas are given in Chapter 14.

A command that is issued by a program in this way is issued in the PROGRAM context. Any command that can be issued in the CORE IMAGE context may also be issued in the PROGRAM context, provided that it does not suspend, re-enter or delete the program. This restriction is necessary because the command processor must be able to return control to the instruction after the PERI Type 60, when the program-issued command has been obeyed and its command processor level has been erased. Thus, the commands HALT, FAIL, RESUME and LOAD are forbidden in the PROGRAM context. A macro command may be issued by a program and creates a further command processor level in the normal way, but at no level below the program-issued command may the program be suspended, re-entered or deleted.

Parameter identifiers within a command issued by a program will be replaced by special *object program parameters* in the ENTER or RESUME command. These parameters are analogous to the job parameters of the JOB and RUNJOB commands. They follow the normal parameters of ENTER and RESUME, and are preceded by the character string PARAM. Internal spaces in object program parameters are treated by the command processor as significant.

Example 2/28

It is required to run a long program that requires one pair of on-line peripherals for the first half of the run (*CR0 and *LP0) and a second pair of on-line peripherals for the second half of the run (*CR1 and *LP1). The batches of output are to be headed by the same document name. The user wishes to release the first pair of peripherals as soon as the program has finished using them and at the same time to connect the second pair of peripherals to the program. He has two macro definitions to organise the operation, A and B. Macro A will load the program, ONLINE the first pair of peripherals and enter the program. Macro B will be issued by the program by means of a PERI Type 60. It will release the first pair of peripherals and make the second pair on-line before returning the job to program level.

Macro A
```
MACDEF FRED
LOAD PROG
ONLINE %A, %B
ONLINE %C, %D
ENTER 0, PARAM(%A, %C, %E, %F, %G, %D)
IF NOT DELETED, SAVEPROG
ENDM
```

Macro B
```
MACDEF NEWPERIS
RELEASE %A
RELEASE %B
ONLINE %C, %D
ONLINE %E, %F
```

Macro A is issued by the command

```
FRED *CR0, INDOC1, *LP0, OUTDOC, *CR1, INDOCZ, *LP1
```

Macro B is issued by a PERI Type 60 mode 1, which writes the characters

```
NEWPERIS %A, %B, %C, %D, %E, %F
```

When the command processor reads the ENTER command it substitutes parameters of the command at the level above (FRED) for the parameter identifiers in ENTER and issues the command as

```
ENTER , *CR0, *LP0, *CR1 INDOCZ, *LP1, OUTDOC
```

When the command processor reads the NEWPERIS command issued by the program, it again substitutes parameters of the command at the level above (in this case ENTER) and issues the command as

```
NEWPERIS *CR0, *LP0, *CR1, INDOCZ, *LP1, OUTDOC
```

This command will cause the commands in the file NEWPERIS to be obeyed as follows:

```
RELEASE *CR0
RELEASE *LP0
ONLINE *CR1, INDOCZ
ONLINE *LP1, OUTDOC
```

The command processor level structure of this example is illustrated below in the form of a table. Each column contains the commands issued at a particular command processor level. It is asumed that the macro FRED is issued from the job description file, that is, at level one.

Command Processor levels			
1	2	3	4
FRED	LOAD		
	ONLINE		
	ONLINE		
	ENTER		
	.		
	.		
	.		
	program		
	instructions		
	.		
	.		
	PERI type 60		
		NEWPERIS	
			RELEASE
			RELEASE
			ONLINE
			ONLINE
	Instructions after		
	PERI type 60		
	IF		
Command after			
FRED			

Object program parameters as program data

The object program parameters of an ENTER or RESUME command have a second function. They can be read directly by the program and then treated as data. This is done by mode 2 of the PERI Type 60 instruction. This mode reads the parameters in sequence. All internal spaces in a parameter are treated as significant, as with mode 1 of the instruction. Once a parameter has been read by a mode 2 PERI Type 60, it cannot be read again unless X is replaced (bit 0 of word 0 of the control area is set); in this case the number of the parameter required will be given by bits 15 to 23 of X. Further details about the control area are given in Chapter 14.

Effecting a break-in

If a PERI type 60 mode 3 is issued from a program in a MOP job, the command processor simulates a break-in and control returns to the MOP console at command level in break-in context. A CONTINUE command will return control to the instruction after the PERI type 60 mode 3. If a PERI type 60 mode 3 is issued from a background job, the program that issues it will fail.

COMMAND ISSUER, PERI MODES 11, 12 AND 13

The basic purpose of these modes is to allow a program to simulate a MOP terminal, and they are therefore most likely to be used to provide MOP facilities on non-standard devices. A full description and example of this use of command issuers is given under *On-line control of exotics*, page 88.

BREAKING IN ON PROGRAM ISSUED COMMANDS

If a user breaks in on a program issued command before it is obeyed, the message:

BROKEN IN DURING ENTER IN. . .

will be output (assuming ENTER was the command by which the program was entered). Word 8 of the program will be stepped back so that the PERI type 60 will be obeyed again when the user issues a CONTINUE command. Thus if a user breaks in on a program issued macro, when he CONTINUES the macro will be obeyed from the beginning again. This may be avoided by setting a WHENEVER BREAKIN command in the macro to deal with a break-in.

How GEORGE processes a command

There are three categories of command verb:

1 Built-in commands

These are recorded in a dictionary of built-in commands together with their two-letter abbreviated forms.

2 System macro commands

These are the local names of macro definition files entered in the directory :SYSTEM.MACROS.

3 User macro commands

(a) Normally these are the local names of macro definition files entered in the current directory of a job.

(b) Less commonly they are the local names of macro definition files entered in directories superior to the current directory of a job. It is thus possible for a user to make use of macro definition files belonging to his superiors, without changing the current directory of the job.

When a user issues a command, GEORGE tries to identify the verb as belonging to one of the above three categories. The order of its search is as follows:

1 It examines the dictionary of built-in commands to see if the verb coincides with the full or abbreviated form of any of these commands

2 It sees if the verb coincides with the local name of any files entered in the current directory of the job

3 It examines :SYSTEM.MACROS to see if the verb coincides with the local name of any file entered in this directory

4 It examines the directories superior to the current directory of the job, searching in ascending order from the directory immediately superior to the current directory up to and including :MANAGER

Note that with a command issued in the NO USER context, stages 2 and 4 of the search are omitted as there is no current directory.

If GEORGE fails to identify a command verb, the command error message

THIS IS NOT A COMMAND

is issued. Note that this message will be given both when a verb is unknown to the system and when a user macro is issued in the NO USER context.

As can be seen from the order of GEORGE's search for a command verb, care must be taken in the choice of names for macro definition files. Two points in particular are worth noting:

1 The local name of a macro definition file should never coincide with the full or abbreviated form of a built-in command.

2 A user should never give a user macro definition file the same local name as a system macro definition file, unless he does not wish to make use of the system macro.

If the user wishes to ensure that the names of his own macros will never clash with GEORGE commands or macros issued in future by ICL, he should begin his macro names with Y or Z. ICL will not issue any commands or macros beginning with these two letters.