CLM-PDN 1/74

PRELIMINARY GCL USER MANUAL

by

R J Dakin

ABSTRACT

GCL (General Control Language) has been designed to provide a common job control interface to a variety of computing systems. This manual is intended to serve both as a general introduction to GCL and as a guide to its use on the ICL 4-70 at Culham. GCL provides a convenient means of accessing the facilities of the Multijob operating system, together with enhanced facilities such as file substitution. The manual is "preliminary" since it is hoped that increasing usage of GCL will give rise to improvements and extensions to GCL facilities.

U.K.A.E.A. Research Group
Culham Laboratory
Abingdon
Berkshire

February 1974

CONTENTS PAGE

CONTENTS PAGE

APPENDIX - HOW TO WRITE GCL FUNCTIONS

CHAPTER 1

INTRODUCTION

GCL is a General Control Language which is being developed to provide a common user interface to the facilities of a variety of large computer systems. This manual is intended to provide a guide to GCL in general [and, in particular, to the version implemented on and for the Multijob operating system for ICL System 4 computers]. [To help distinguish these two aims, anything specific to the Multijob version is enclosed in square brackets]. It will be necessary to use some terms in a fairly specific way; to assist the reader, each term is underlined when it is first introduced.

GCL has been designed with a view to its use in satellite computers and this imposes some discipline in the way things are organised. While this should not prevent the application of GCL in other environments, an understanding of the satellite environment does shed some light on the rest of the manual.

1.1 The Satellite Environment

Relevant features of a typical satellite environment for GCL are shown in Fig. 1.

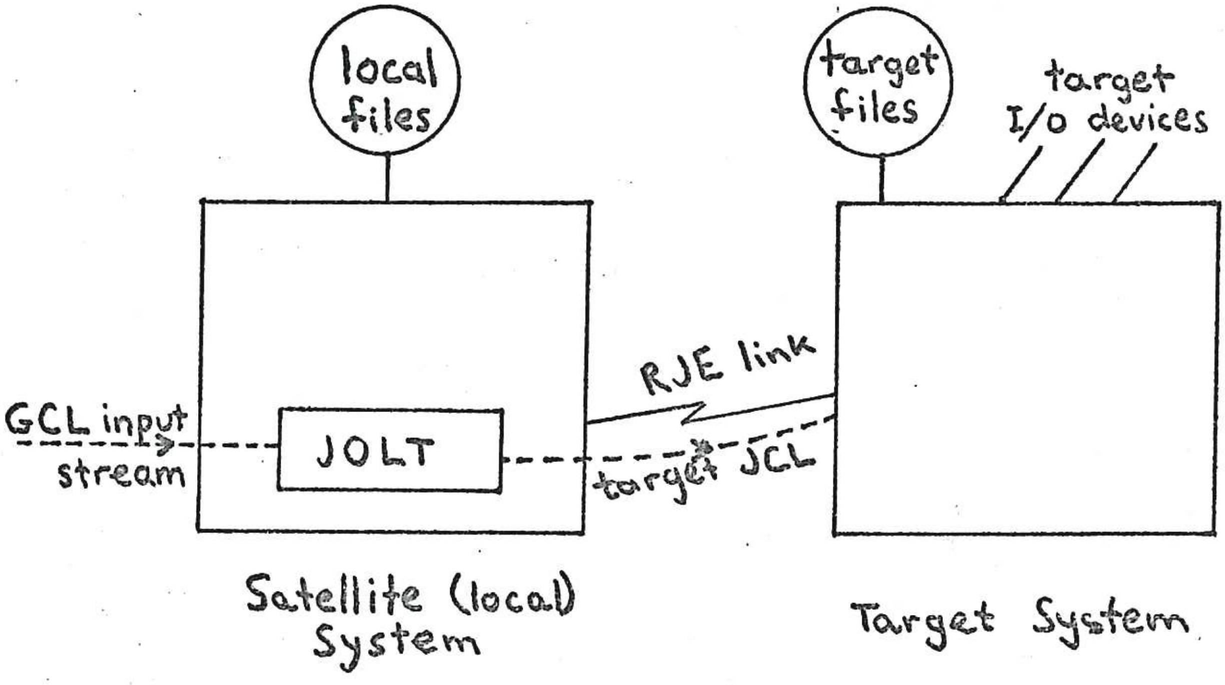


Fig. 1 Satellite and Target Systems

GCL comes into the satellite via an input stream (which may be, say, from a card reader, paper tape reader or local file) and is translated by a Job Language Translator program (JOLT) into the job control language (JCL) of the target main frame system which is to perform the required processing. It is then transmitted to the target system via a remote job entry link. In due course the JCL will be obeyed by the target system, causing activity by the target filing system and I/O devices and the transmission of results back to the satellite.

Note the functional distinction between local and target files : only local files are accessible during translation from GCL to JCL, while only target files are accessible when obeying JCL on the target system. The distinction between local and target files is maintained in the rest of the manual, although it disappears if JOLT is run on the target system [as in the Multijob implementation].

1.2 On the Rest of This Manual

GCL is based on a simple regular structure. It will save a lot of repetition when describing particular GCL facilities if we consider the structure separately, which we do in Chapter 2.

Further essential groundwork is provided in Chapter 3 which describes how GCL deals with input and output devices.

The main facilities currently implemented for GCL are introduced in Chapter 4 via a series of examples. This is intended simply to convey an impression of how actions are expressed in GCL and may leave some questions unanswered; it shoul however, throw light on Chapters 5 and 6 which present the facilities in a more complete and systematic fashion.

[Finally, Chapter 7 describes how to use the Multijob version of JOLT.]

A major design aim of GCL has been to make it open ended and easy to change. This is a preliminary manual and is not intended to be definitive as far as the GCL user image is concerned. Any suggestions for improving it will be most welcome.

CHAPTER 2

GCL STRUCTURE

2.1 Main Features

The GCL unit corresponding to a program in normal programming languages is a job which specifies a sequence of actions to be performed by the target system on behalf of a single user. A session is comprised of a number of statements, possibly interspersed with source program or data text.

A typical GCL statement looks like this:

```
RUN(P,<I>,<PRINTER>,STORE=100,LANGUAGE=ALGOL);
```

which means apply the function RUN to the parameters P, <I> and <PRINTER> with the options STORE and LANGUAGE set to 100 and ALGOL respectively.

The significance of a parameter depends on its position; thus the first parameter of RUN specifies the program to be run, the second - <I> - specifies inputs to the program and the third - <PRINTER> - specifies outputs. A function normally requires a fixed number of parameters which must be provided in any call on the function.

Option settings, on the other hand, can be written in any order or left out altogether. They must come after the parameters and can, optionally, be separated from them by a colon instead of a comma to highlight the transition. Thus the example could be written:

```
RUN(P,<I>,<PRINTER>:STORE=100,LANGUAGE=ALGOL);
```

With no option settings it would reduce to:

```
RUN(P,<I>,<PRINTER>);
```

When option settings are omitted, default values apply. The system provides defaults, but if these don't fit your requirements you can set your own by writing statements like

```
LANGUAGE=ALGOL;
```

which sets the LANGUAGE default to ALGOL for the rest of the job (or until a similar statement sets it to some other value). If you habitually use defaults which differ from the system ones then you can arrange for your defaults to be automatically set up whenever you start a job by putting default setting statements in a local file. [In the Multijob version this is normally the file UTPROG.SETGCL(S) under your own username.]

Some functions require no parameters, but the parameter brackets must always be present : RUN refers to the function itself - it will only do anything if it is applied to a parameter list. A call on a parameterless function may, however, include option settings. Examples of parameterless functions are:

```
RUNJOB();
TEXT(TERM='END');
```

The symbols <> are simply an alternative form of the bracket pair () and mean exactly the same thing. Thus, our second example could equally well be written

```
RUN (P,(I), (PRINTER));
```

The original version is a little more readable, which is why GCL allows alternative brackets. Anything enclosed in brackets is called a list. Thus <I> is a list with the single element I while (P,<I>, <PRINTER>)is also a list with three elements P, <I> and <PRINTER>.

RUNJOB () and TEXT (TERM='END')
(in fact, most of the above examples with the semicolon terminator removed) are function calls. A function call usually specifies some action and returns a value. The point of this is that it allows us to define an expression which is

i) an integer)
ii) an identifier)
iii) a string) discussed in the following section
iv) the refer-back symbol*)
v) a list, or
vi) a function call

(This definition can be extended, but the extensions are not applicable to most users.) The apex of this pyramid is that any expression is a valid form of parameter, option setting or list element. Thus a considerable nesting of brackets in statements is possible, though excessive nesting can be (and should be) avoided.

2.2 Integers, Identifiers, Strings and Refer-Back

Let us now consider some of the more elementary GCL constructs, most of which appeared in the previous section.

2.2.1 An integer is written as an unsigned sequence of digits terminated by anything that is not a digit and has its normal numerical significance.

2.2.2 RUN, P, I, PRINTER, STORE, ALGOL etc. are all identifiers which take the usual alphanumeric form (initial letter). You will sometimes need to introduce your own identifiers to represent items to which you make repeated reference. Your own identifiers should either be a single letter or contain the ampersand character (&) (which is treated as a letter) to distinguish them from system identifiers. Thus P and I in the previous examples were both user identifiers. A new identifier does not need a separate declaration statement but must be preceded by an exclamation mark (!) the first time it occurs. An identifier can be of any length but the first 12 characters must be distinct; as each occurrence of an identifier longer than this generates a warning message it is best to keep to 12 or less characters. An identifier is terminated by any character other than a letter, digit or ampersand.

You can use identifiers to effectively shorten function names. For example:

```
!C= COMPILE:/ASSIGN COMPILE FUNCTION TO C
  C(X);      /EQUIVALENT TO COMPILE(X)
```

Other examples of identifier usage will arise in later sections.

2.2.3 'XYZ' is a string, which is a sequence of characters enclosed between primes. Any printable character can appear in a string, but the characters /*'; must be represented by character pairs as follows:

*/	represents	/
*;	"	;
*'	"	'
**	"	*
*N	"	new line

2.2.4 The refer-back symbol asterisk (*) is used in a statement to access the value (if any) generated by the preceding statement. For example

```
RUN(LINK<A,B>,<I>, <PRINTER>);
```

is equivalent to

```
LINK(A,B); RUN (*, <I>,<PRINTER>);
```

2.3 Layout and Commenting

Our original statement example, which was

```
RUN(P,<I>,<PRINTER>:STORE=100,LANGUAGE=ALGOL);
```

could equally well be written:

```
RUN(P, / (PROGRAM TO CALCULATE REGRESSION COEFFTS)
   <I>, / FROM EXPERIMENT NO.18
<PRINTER>:/
        STORE=100,
        LANGUAGE=ALGOL);
```

This obeys the GCL format rules, which are:-

i) extra spaces are ignored,

ii) end of line is ignored, and

iii) slash (/) and anything that follows it, is ignored.

The use of "extra" in rule (i) needs some explanation: a space character will terminate an identifier or integer; also spaces are meaningful inside strings - but any other spaces are "extra" and can be used freely to improve layout.

Note that slash terminates a line even inside a string unless preceded by an asterisk. Thus

```
'THE RAIN IN /
SPAIN'
```

is equivalent to

```
'THE RAIN IN SPAIN'
```

2.4 ++ Statements

It is usually obvious both to JOLT and to the user which lines contain GCL statements and which ones contain program or data text. We shall, however, encounter circumstances where we wish to obey a GCL statement in the middle of text - for example we may wish to specify insertion of the contents of a local file at some point in the text. To cater for this, GCL uses the two characters ++, which must occur at the beginning of a line, as a special marker. When JOLT encounters a line marked in this way, it

i) suspends whatever it was doing when it read in the line,

ii) reads and obeys one GCL statement following the ++ (like any other GCL statement, this one can span more than one line),

iii) throws away the rest of the line on which the statement terminates, and

iv) continues with the activity suspended in (i).

Only one statement is read and obeyed as the result of a ++ marker. If you wanted to obey several statements in the middle of text you would have to precede each of them by a ++ marker.

If a genuine line of text starts with ++ then you must modify it to +++ (the first + being discarded) to prevent it being treated as a ++ marker. This last facility is not implemented at the time of writing, so it would be as well to check availability before using it.

CHAPTER 3

INPUT AND OUTPUT

3.1 Conceptual Framework

Activities on the target system send and receive information to and from a variety of places, many of which need to be specified explicitly by GCL. The situation for a given target system may be complicated by intermediate buffering (spooling) between a slow I/O device and a running program; this buffering may be more or less explicitly specified by the target JCL.

In GCL the specification of all sources and sinks of information is treated within a single conceptual framework illustrated in Fig.2.

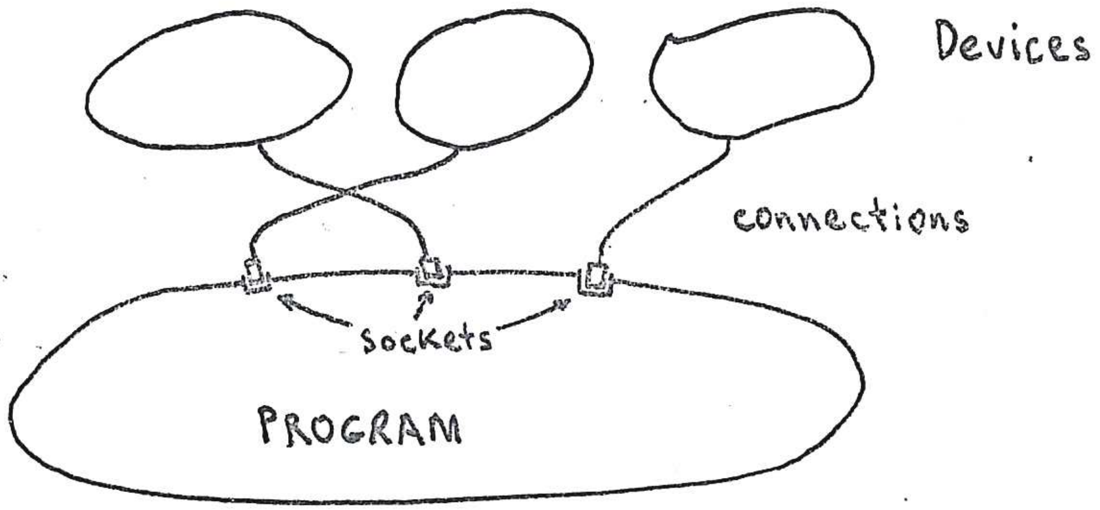


Fig. 2. How GCL Pictures I/O

An information source or sink (which we call a device) may be connected to a program run by connecting it to a socket which is an identifiable connection point. In different target systems sockets correspond to symbolic filenames, channel numbers, DD names etc. As well as specifying connections between devices and sockets we may specify other I/O operations such as listing information read from a device. GCL always pictures connections as being direct : any complications in the target JCL arising from intermediate buffering are automatically generated by JOLT and need not concern the user.

The simplest device category is physical devices such as card readers and line printers on the target system, extending naturally to include, say, a printer on the local satellite system. We regard the target filing system as simulating a large number of GCL devices (variously known as files, documents or data sets) on a small number of physical devices such as disc drives. We shall call these GCL devices <u>files</u>. Bodies of program or data <u>text</u>, supplied in the GCL stream constitute a third and final device category.

Physical devices are represented in GCL by suitable identifiers. The only one available at the time of writing is PRINTER, which refers to the normal output printer.

Text and file devices must be defined by the user himself by calling a suitable function. Once defined the device can be conveniently assigned to a user's identifier for later reference. We will now look at device definitions in detail.

3.2 In-Line Text

A body of text is defined by calling the parameterless function TEXT, which:

i) throws away the rest of the current input line, and
ii) accepts, as part of the defined text body, subsequent lines up to, but not including, a line starting with terminator specified by the TERM option (default is the string '*/**' specifying a /* terminator).

After obeying TEXT, JOLT will continue input just after the terminator on the same line.

Example:

```
!A=TEXT(); /ASSIGN FOLLOWING TEXT TO A
---
---
---
/*!B= TEXT(TERM='=END');/END A'S TEXT-B's FOLLOWS
---
---
---
=END / TERMINATES TEXT ASSIGNED TO B
```

3.3 Target System Files

A file device is defined by one of four functions, depending on the nature of the information it contains. Each function requires a single parameter which is either the identifier NONAME, for unnamed temporary files, or a string of alphanumeric characters for a named file. In the latter case the string forms part of the full target system file identifier; for maximum generality it should start with a letter and be no longer than five characters, although not all target systems are so restrictive. Other components of the target system identifier and any other information affecting format, access, choice of physical device etc. are provided by option settings when the file is defined. Such options and their settings inevitably reflect target system characteristics which need not, however, concern the user who keeps to defaults.

A named file is always permanent and must be explicitly deleted if required. Unnamed files are always temporary and disappear on completion of the job.

Note that the same file should not be defined more than once within a job - otherwise invalid target JCL can, in some circumstances, result. Repeated file definitions with the NONAME parameter always define distinct files. A file definition does not specify any action by the target system - in particular, it does not cause the file to be created.

The four file definition functions are:

SOURCEFILE for files containing program source text,

OBJECTFILE for files containing object code - i.e. compile code which needs to be combined with other modules and subroutine libraries to produce a program,

PROGFILE for files containing loadable program, and

DATAFILE for files with unspecified content.

Examples:

```
!&TABLE=DATAFILE ('TABLE');NAMED DATA FILE
!S=SOURCEFILE('SETUP');/NAMED SOURCE FILE
!P=PROGFILE(NONAME);UNNAMED PROGRAM FILE
```

CHAPTER 4

INTRODUCTION TO GCL FACILITIES

This chapter provides an informal introduction to the facilities of GCL. The intention is to illustrate rather than specify, the latter being reserved for later chapters.

4.1 Simple Compile-and-Go

```
++JOB(SMITH); /START JOB FOR SMITH
RUNJOB(); /COMPILE AND GO
---
---  } source program
---
/*
---
---  } data
---
/*
ENDJOB();/END OF JOB
```

The effect of the above is to compile and execute the source program, using the supplied data as input and sending its output to the PRINTER, on behalf of a user whose GCL user name is SMITH. (The target system user name and any other accessing information are automatically generated by JOLT.)

The start and end of the job are defined by the JOB and END statements. It is not strictly necessary that JOB be a ++ statement; this is simply a safety measure to prevent any error in a preceding job from swallowing SMITH's job as data or causing other consequential errors.

The RUNJOB function invokes two calls on the TEXT function to read the program and data. The language in which the program is written is specified by the LANGUAGE option (default FORTRAN).

"Normal" input and output sockets are assumed for data and PRINTER. [For the Multijob version using Fortran, these would be the symbolic filenames DSET97 and DSET99 respectively.]

4.2 Text Substitution

The job:

```
++ JOB(SMITH);
RUNJOB();
++ SUBST('PART1');
       F=X**1.7
++SUBST('PART2');
/*
--- )
      data
--- )
/*   ENDJOB();
```

is equivalent to:

```
++JOB(SMITH)
RUNJOB();
---   )
---   ) contents of local file PART1
---
         F=X**1.7
---
---  )
     ) contents of local file PART2
---
/*
---  )  data
---
/* ENDJOB();
```

In this way the Fortran statement F=X**1.7 has been imbedded in a program which might, for example, graph F as a function of X. [In the Multijob version the "local" file PART1 is the file PART1(S) under the user and group names specified by the options USER and GROUP.]

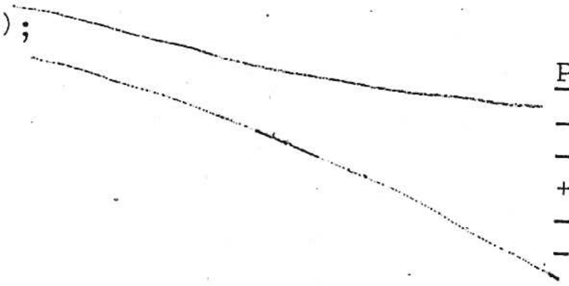
Substitution can be nested to a depth of three levels (an implementation restriction which can easily be relaxed).
A frequent use of two level substitution is the insertion of Fortran COMMON blocks - especially convenient in programs with a large number of modules. For example, suppose the local file COMM1 contains COMMON declarations and the local file PROG contains the line
++SUBST ('COMM1');
then this line will be replaced by the COMMON declarations if PROG is substituted as in the following example:

```
++JOB(SMITH);
RUNJOB();                                  PROG
++SUBST('PROG');
/*                                         ---
                                           ---
---                                      ++SUBST('COMM1');
    ) data                                 ---
---                                        ---
/* ENDJOB();
```

The use of an identifier as a SUBST parameter can be very powerful; for example:

```
++JOB(SMITH);
!C='COMM1';
RUNJOB();
++SUBST('PROG');

 /*
 ----
 ----    (data)
 ----
 /*ENDJOB ();
```



```
PROG
----
----
++SUBST(C);
----
----
```

is equivalent to the previous example, but can easily be made to select a different COMMON block - contained in the local file COMM2, say - by changing the second GCL statement to

```
!C='COMM2';
```

4.3 Multi Module Programs

A call on the RUNJOB function introduced above is equivalent to the following call on the RUN function:

```
    RUN(<TEXT()>, <TEXT()>, <PRINTER>);
```

A more general use of RUN is illustrated below.

```
++ JOB(JONES);
!&TABLE=DATAFILE('TABLE');/FILE ALREADY SET UP
!&MOD1=OBJECTFILE ('MOD1');/COMPILED 3/12/73
!&DATA=TEXT();
---
---
---
/*
!X=TEXT();/PROGRAM MODULE
---
---
/*
RUN(<X,&MOD1>,<&DATA,&TABLE>,<PRINTER>);
ENDJOB();
```

This will give rise to the following actions by the target system:

i) the module X will be compiled and linked with the previously compiled module &MOD1 to form a program;

ii) This program will be run with its first and second input sockets connection to &DATA and &TABLE respectively and its first output socket connected to PRINTER.

We assume that there is some natural ordering of input and output sockets pertaining to a particular source language and target system. [Thus for the Multijob version using Fortran the first and second input sockets are assumed to be DSET97 and DSET5.]

4.4 Running an Existing Program

Example:

```
++JOB(DAKIN);
!P=PROGFILE('ANAL');/(ALREADY EXISTS)
!&DATA=TEXT();
----
----
/*
RUN(P,<&DATA>,<PRINTER,PRINTER>);
ENDJOB();
```

Note that in this case the first parameter is not in brackets. The use of <P> would imply that P is a component of the program to be run rather than a complete loadable program.

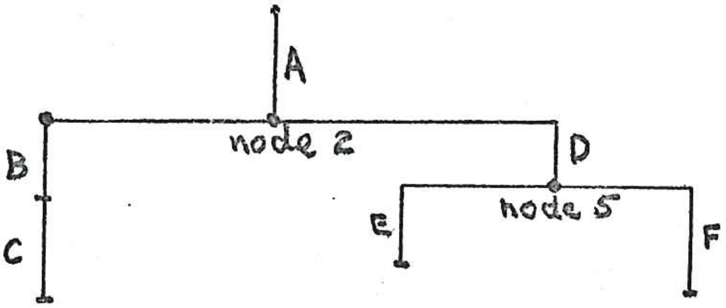
The connection of PRINTER to two output sockets is allowed; the two lots of output come out separately.

The assumed ordering of sockets in the previous section still applies. One can change this order via the INLIST and OUTLIST options. For more general sockets you will need to use the RUNB function described in Chapter 5.

4.5 Overlay Programs With Tree Segmentation

In this example we will suppose that A,B,C -- F are already defined as source or object files or text. The statement:

RUN(<A,2,B,C,2,D,5,E,5,F>, ---
specifies the formation and execution of an overlayed program with a tree overlay structure as follows



Thus integers as program components represent nodes in the overlay tree. In the absence of node numbers modules are placed end-to-end. The first occurrence of a particular node number effectively labels the current store position as the start of an overlay area. Each set of modules to be overlayed in this area is preceded by the node number. Node numbers for tree segmentation must not exceed 99.

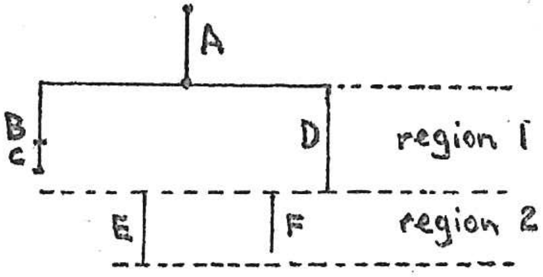
Any modules extracted from subroutine libraries are inserted in the root segment.

4.6 Overlay Region Specification

The statement:

RUN(<A,101,B,C,101,D,102,E,102,F>,----

specifies a program with the following overlay structure



Thus, each integer greater than 100 (100+N,say) defines a new overlay region number N which follows all preceding regions in store. Again N should not exceed 99.

Some target systems will allow only one type of overlay structure; if the target system allows both you are free to mix them, but all node numbers and region numbers must be distinct (for example, you should not include both 6 and 106 in the program component list). [Multijob does allow both types of overlay.]

4.7 Separate Compilation and Link Editing

The LINK function allows you to specify the link editing of a program (with prior compilation if necessary) without subsequent program execution. Its parameter list is of the same form as the program component list which is the first parameter of the RUN function. Thus the program specified in the example in section 4.6 will be formed by the statement:

```
LINK(A,101,B,C,101,D,102,E,102,F);
```

Similarly the COMPILE function is used to specify separate compilation of a module. If you wish to retain a compiled module in an object file on the target system then COMPILE must be used; LINK automatically discards any object files which it forms.

4.8 Use of Private Subroutine Libraries

Two functions are available in connection with subroutine libraries: LIBRARY, which simply defines a library and CATALOGUE which creates a new library or updates an existing one. The LIBLIST option is used to specify which private libraries are to be used when link editing. The CATALOGUE function requires two parameters: the first specifying object modules to be deleted from the library and the second specifying modules to be inserted..

Examples:

```
 /(&SQRT, &NSQRT, &EXP, A,B AND C ARE ASSUMED
/TO BE ALREADY SET UP AS OBJECT FILES)
```

```
/(1) LIBRARY UPDATE:
CATALOGUE(<&SQRT>,/(DELETED FROM LIBRARY)
      <&NSQRT,&EXP>); /(ADDED TO LIBRARY)
/(2) LIBRARY DEFINITION
! L = LIBRARY (&NSQRT,&EXP,'LOG','SIN');/PARAMETERS
/    SPECIFY ENTRY NAMES
/(3) USE OF LIBRARY IN LINK EDIT
!P=LINK(A,B,C : LIBLIST = <L>);
```

4.9 Transparent Insertion of JCL

Occasionally you may encounter requirements which are not catered for by GCL; in such cases it is possible to use GCL for everything which GCL can handle and directly insert JCL only where absolutely necessary. The parameterless function TRANSPARENT is used to define sections of inserted JCL in much the same way as the TEXT function. [The Multijob version has the additional functions RTP and NOWTRIALS for insertion of run time parameters and Trials statements respectively at appropriate insertion points.]

[4.10 Notes on the Multijob Implementation]

The Culham Multijob version includes the option SYSTEM which allows you to specify the particular Culham stream configuration (A,B or C) on which the job is to run. The setting of SYSTEM should remain constant throughout the job; its effect is to tailor generated JCL to the configuration and ensure that stream store limits etc. are not violated.

In most circumstances all print output for a job is queued at the end of the job and so should be contiguous. The JOURNAL option allows you to control whether or not journal files are printed and/or deleted.

When two or more JOLT jobs under the same user name are concurrent there are likely to be clashes in journal and other file names. Clashes can be avoided if each job initialises the option defaults for TRNO, CBRUN, LSDRUNNO and PRUN to a different integer, reasonably well separated (by

10 or so) from all others. Thus one job might use the default values of 0 while another might include the statements

```
     TRNO=10; CBRUN=10; LSDRUNNO=10; PRUN=10;
```

after the ++ JOB statement.

Each use of the PRINTER device causes output to be buffered in a file called PRINT(S) with run number PRUN, which is incremented on each occasion to make the file identifiers distinct.

CHAPTER 5

FUNCTION DETAILS

This chapter describes each of the available GCL functions. Since one option may apply to several functions with much the same effects, details of options, settings and defaults are dealt with separately in Chapter 6.

5.1 Job Definition

5.1.1 JOB

Purpose: to initiate a job.

Value Returned: none.

Parameters: one parameter which is an identifier allotted to you as your GCL user name. [If none has been allotted then a string comprising the Multijob username can be used instead.]

Action: the job is initialised. If the user has a GCL initialisation file this is automatically inserted in the input stream following the line containing the JOB statement; this file can contain your own option default settings, file definitions etc. [This is an S file in your own file space whose name depends on the GROUP and SETGCL options - UTPROG.SETGCL(S) if you use defaults.] If this file does not exist no harm will result, but a warning message will be generated.

Options: SETGCL[,GROUP]

Note: JOB should be a ++ statement.

Examples:

```
/(1) USING GCL USER NAME
++JOB(SMITH);
/[(2) NO GCL USERNAME ALLOCATED:
++JOB('JRSCLU':GROUP='FIELD');
RUN(P,<&DATA>,<PRINTER>;/DEFINITION OF
/P AND &DATA IN JRSCLU:FIELD.SETGCL(S)]
```

5.1.2 ENDJOB

Purpose: to terminate a job.

Value returned: not applicable.

Parameters: none.

Action: if no failures were detected then generated JCL is submitted for execution. If failures were detected the

JCL is diverted to the error stream with all text bodies contracted to the single line

```
:::::::::: TEXT :::::::::
```

[In the Multijob version JCL goes to the file UTPROG.JCLOUT(S) (modifiable by the JCLOUT option) and the error stream to the file UTPROG.GCLERR(S). Both files replace any previous file of the same name.]

Options: [JCLOUT]

Examples:

```
    ENDJOB ();/[JCL TO UTPROG.JCLOUT(S)]
    ENDJOB  (JCLOUT='MYJCL');/[JCL TO UTPROG.MYJCL(S)]
```

5.1.3 LINECOUNT

Purpose: to reset the line number count used in failure messages. This is of value when a GCL setup file is used and is known to be error free.

Parameters: a single integer parameter which is to be the new line number corresponding to the line on which the LINECOUNT statement terminates.

Value Returned: the old line count.

Options: none.

Example: if the last line of a setup file (invoked by JOB) is

```
          LINECOUNT(1);/RESET LINE COUNT
```

then this has the effect of excluding lines in the setup file from line numbers given in failure messages which will, then, correspond to the actual numbers in the GCL file.

5.2 Device Definition Functions

5.2.1 SOURCEFILE, OBJECTFILE, PROGFILE and DATAFILE

Purpose: to define a file containing source program, a compiled module, a loadable program or unspecified content respectively.

Value returned: the file so defined.

Action: no action on the target system is specified by these functions.

Parameters: a single parameter which is NONAME for an unnamed temporary file or a string containing an alphanumeric identifier of up to 5 characters [6 are allowed in the Multijob version] for a named permanent file. Unnamed files disappear at the end of the job.

[Options: for all functions: GROUP,USER,RUNNO,VSPEC and VOL for DATAFILE only: FTYPE,TRSPEC and TRCYL. Note: TYPEZ files must be named in the present implementation.]

Note: the same named file should not be defined more than once in a job.

Examples:

```
/(1) SYSTEM INDEPENDENT EXAMPLES
 !P=PROGFILE('DOIT');
 !&WORK=DATAFILE(NONAME);
/[(2)-EXAMPLES SPECIFIC TO MULTIJOB
 !&MOD1=OBJECTFILE('MOD1':RUNNO=100,GROUP='XYZ');
 !F=SOURCEFILE('ANALYS');/(6CHAR.FILE NAME)
/]
```

5.2.2 SCRATCH

Purpose: to define a scratch (unnamed temporary) file. SCRATCH() is exactly equivalent to DATAFILE(NONAME).

Parameters: none.

[5.2.3 SYSMFLE and SYSUFLE]

Purpose: to concisely define Multijob system program files.

Value returned: the file so defined.

Parameters: one string parameter giving the name of the file.

Action: both define program
files with no run number; SYSUFLE define a
file under SYSTEM:UTPROG and SYSMFLE defines
a file under SYSTEM:MJPROG.

Examples:

```
!N=SYSUFLE('NDFHK');/DEFINES SYSTEM:UTPROG.NDFHK(P)
!T=SYSMFLE('TRIALS');/DEFINES SYSTEM:MJPROG.TRIALS(P)
```

5.2.4 TEXT

Purpose: to define a body of text in the GCL input stream.

Value returned: the text so defined.

Parameters: none.

Options: TERM

Example:

```
!&DIST=TEXT();
      FUNCTION DIST(X,Y)
      DIST=SQRT(X*X + Y*Y)
      RETURN
      END
/*    /(TEXT TERMINATOR)
COMPILE(&DIST);/SUBSEQUENT REFERENCE TO TEXT
```

5.3 Device Manipulation Functions

5.3.1 PRINT

Purpose: to print the information read from a device.

Parameters: one parameter, which is a device.

Value returned: if the device is a target file - the device itself; if the parameter is text - an unnamed target file containing the text.

Options: NUMBERED, BIGFILE, [PRINTQ]

Examples:

```
!F=SOURCEFILE('F'); PRINT(F);/FILE LISTING
!T=PRINT(TEXT<>);/LIST TEXT =>T
----
----
----
/*
```

[Multijob Sequencing Limitation: at execution time printing will always follow previously specified activities but will not necessarily precede subsequent ones (but PRINT followed by DELETE is always correctly sequenced). Thus

```
PRINT(F); /PRINT OLD F
RUN(P,<I>,<F>);/RUN REPLACES FILE F
```

may result in printing the new version of F rather than the old version as specified. Such situations rarely occur in practice.]

5.3.2 DELETE

Purpose: to delete a file on the target system.

Parameters: a single parameter which is the file on the target system.

Value returned: the file which is deleted.

Options: None

[Multijob Restriction: Dedicated files cannot be deleted at present.].

Example:

```
DELETE(&F1),
```

5.3.3 DISPLAY

Purpose: to print and then delete a file.

Example:

```
   DISPLAY(F);/EQUIVALENT TO:
   /  DELETE(PRINT<F>);, OR
   /  PRINT(F); DELETE(F);   ,OR
   /  PRINT(F); DELETE(*);
```

5.3.4 COPY

Purpose: to copy information from an input device to an output device.

Parameters: two parameters, which are the input and output devices.

Action: copying takes place until the end of the information is reached. Since the only currently available input devices are text or files, "end of information" is either end of text or end of file.

Value returned: none.

Options: none.

Note: if the output device is a target file it must not exist already; if it does exist it should be DELETE'd first.

Example:

```
    !A=DATAFILE('DATA'); !B=DATAFILE('FRED');
    COPY(TEXT<>,A);/TEXT TO TARGET FILE
    ----
    ---- } text copied
    ----
    /* COPY(A,B); /MAKE A FURTHER COPY
```

5.4 Running Programs

5.4.1 RUN

Purpose: To run a program written in a higher level language, with or without prior compilation and link editing.

Parameters: Three parameters, specifying the program to be run, input devices and output devices respectively.

1st Parameter: either a program file, or a list of source files and/or object files and/or node numbers for segmentation. An element of this list can be:

i) an object file for inclusion in the program,

ii) a source file for inclusion in the program,

iii) a body of source text for inclusion in the program,

iv) an integer N in the range $0 \leq N \leq 99$ which is interpreted as a node number for a segmentation tree, or

v) an integer 100+N in the range $0 \leq N \leq 99$ which is interpreted as a region number N for segmentation.

Program components are assumed to be laid end-to-end until a node number is reached; the first occurrence of a node number defines a position in the code storage area which immediately follows the preceding component in the case of (iv) or the end of the longest segment in the previous overlay area in the case of (v).

2nd Parameter: a list of input devices which are connected to corresponding entries in a list of input sockets appropriate to the target system and the source language. [See specifications of the FORTRANIN, ALGOLIN etc. options for details.]

3rd Parameter: a list of output devices which are connected to the output sockets appropriate to the target system and source language. [See FORTRANOUT, ALGOLOUT, etc options.]

Value returned: none.

Options: LANGUAGE, CLTIME, CLSTORE, SOURCELIST, OBJECTLIST, DEBUG, REFERENCES, PROGMAP, MAPLEVEL, LET, LIBLIST, RUNTIME STORE, INTERACTIVE, [JOURNAL, SYSTEM, STREAM], plus options peculiar to the source languages. [Note that the JOURNAL option allows one to print, discard, or retain job journals.]

Examples:

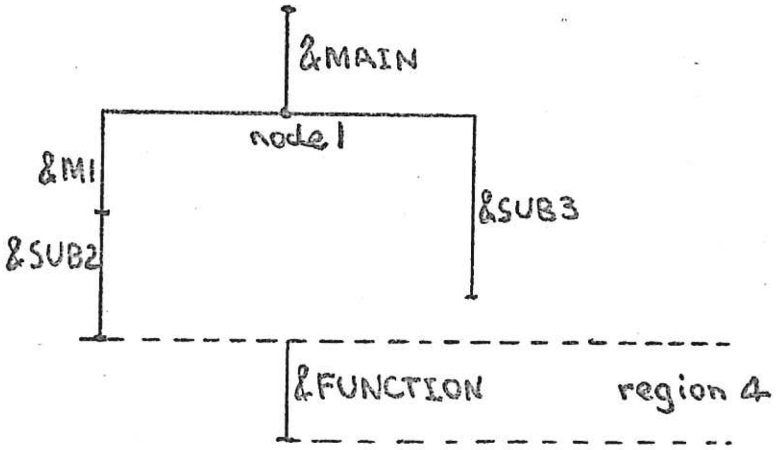
i) Run a previously compiled program using in-line text input plus data held on the target filing system, retaining the output on the target system:

```
!P=PROGFILE('ADDUP'); !I=DATAFILE('TABLE');
!&OUT=DATAFILE('OUTPUT');
!T=TEXT();
----
----
/*
RUN(P, <T,I>, <&OUT>);
```

ii) Form a segmented program and run it, printing the output.

```
/ DEVICE DEFINITIONS:
!&M1=OBJECTFILE('MOD1'); !&MAIN=SOURCEFILE('MAIN');
!&SUB2=OBJECTFILE('SUB2'); !&SUB3=SOURCEFILE('SUB3');
!&FUNCTION=TEXT();
-----
-----
-----
/*
!&DATA=TEXT();
-----
-----
/*
RUN(<&MAIN,1,&M1,&SUB2,1,&SUB3,104,&FUNCTION>,
       <&DATA>, <PRINTER>);
```

This forms a segmented program with overlay structure



5.4.2 RUNJOB

Purpose: To compile and run a single module program supplied as source text, with one input device supplied as text and a single output device which is the printer.

Parameters: None.

Action: RUNJOB(); is precisely equivalent to

```
        RUN(<TEXT()>, <TEXT()>, <PRINTER>);
```

Example:

```
RUNJOB();
----
----        (source program text)
----
/*
----        (data text)
----
/*
```

5.4.3 RUNB

Purpose: to run a program with devices connected to sockets specified in a general (system dependent) manner.

Value returned: none.

Parameters: there are three parameters:

1st parameter is a program file

2nd parameter is a list of sockets. [In the Multijob version a socket takes the form of a string which constitutes a Multijob symbolic filename.]

3rd parameter is a list of devices to be connected to corresponding sockets in the socket list.

Example:

```
RUNB(P,           / PROGRAM P
    <'READ','WRITE'>, /SFN'S READ AND WRITE
    <&DATA,           /(CONNECTED TO READ)
           PRINTER>); /(CONNECTED TO WRITE)
```

5.5 Separate Compilation and Link Editing

5.5.1 LINK

Purpose: to form a program from source and/or object modules without executing it.

Value returned: a program file containing the program; this file is defined by the PROGSAVE option (default is an unnamed file).

Parameters: source files, object files, text and node numbers as described for the first parameter of RUN. Any object files formed by compilation of components are temporary files.

Options: LANGUAGE, CLTIME, CLSTORE, SOURCELIST, OBJECTLIST, DEBUG, REFERENCES, PROGMAP, MAPLEVEL, LET, PROGSAVE, LIBLIST, plus options peculiar to the source language.

Example: to form the program which is specified in the second RUN example in Section 5.4.1:

```
!P=LINK(&MAIN,1,&M1,&SUB2,1,&SUB3,104,&FUNCTION);
```

5.5.2 COMPILE

Purpose: Separate compilation of source code modules, allowing retention of compiled code in permanent files.

Parameters: One parameter which is a device from which source code is to be read.

Value returned: an object file containing the compiled code. This file is a named (permanent) file if the parameter is a named file - otherwise it is an unnamed (temporary) file.

Options: LANGUAGE, CLTIME, CLSTORE, SOURCELIST, OBJECTFILE, DEBUG, REFERENCES, plus source language options.

Examples:

```
!A=COMPILE(TEXT<>);/RESULT IS UNNAMED FILE
----
----
----
/*
!B=SOURCEFILE('JIM');
!C=COMPILE(B);/RESULT IS OBJECTFILE('JIM')
```

5.6 Private Subroutine Library Facilities

In their present form these facilities are probably unduly influenced by the form of Multijob library facilities and may need to be revised in the light of experience with other target systems.

5.6.1 LIBRARY

Purpose: to define a private library.

Parameters: a variable number (up to 127) of parameters which are either strings or named source or object files which define entry names that are satisfied by this library.

Action: the function simply defines a library and does not specify any action by the target system.

Value returned: the library so defined.

Options: [USER, GROUP - which name the library]

Example:

```
  !A=SOURCFILE('M1'); !B=OBJECTFILE ('M2');
!L=LIBRARY(A,B,'M3','M4':USER='JRSMPI');
/EQUIVALENT TO -
/!L = LIBRARY ('M1','M2','M3','M4' : USER='JRSMPI');
!P=LINK(X,Y,Z : LIBLIST = <L,K>); / EXAMPLE
/OF USE OF LIBRARIES IN PROGRAM LINKAGE
/(K HAVING ALSO BEEN ASSIGNED A LIBRARY).
```

5.6.2 CATALOGUE

Purpose: to form a new subroutine library or update an existing one.

Parameters: there are two parameters - the first is a list of object files to be deleted from the library and the second is a list of object files to be added. All files must be named.

Action: if NEW=YES a new library is formed (in which case the first parameter must be an empty list); otherwise an existing library is updated.

Value returned: none.

Options: [GROUP - which names the library; the user name is always taken to be that of the user running the job].

Examples:

```
/(A,B,--- ARE NAMED OBJECT FILES)
CATALOGUE (<>, <A,B,C,D,E>:NEW=YES);/FORM
/          NEW LIBRARY IN DEFAULT GROUP
CATALOGUE (<G,H>,<I,J,K>:GROUP='FRED');
/          UPDATE OF FRED LIBRARY
```

5.7 List Manipulation

This section describes some general utility functions which are particularly useful when manipulating large program suites.

5.7.1 APPLY

Purpose: to apply a one parameter function to each element of a list, replacing the list elements by the results of the function evaluations.

Parameters: there are two parameters : the first is a single parameter function and the second is a list.

Value returned: a list whose elements are the result of evaluatir the function with each element of the list, in turn, as parameter

Options: those applicable to the function.

Example:

```
!X=APPLY(COMPILE,<A,B,C>);/EQUIVALENT TO:
/  !X=(COMPILE<A>, COMPILE<B>,COMPILE<C>);
```

5.7.2 JOIN

Purpose: to form a single list comprised of the elements of two or more lists (i.e. concatenate the lists).

Parameters: two or more parameters which are lists.

Value returned: The resulting list.

Example:

```
!X=JOIN(<A,B,C>,<D,E,F>); /EQUIVALENT TO:
   /!X=(A,B,C, D,E,F);
```

[5.7.3 Multijob Examples using APPLY and JOIN]

(a) Suppose a user wishes to form a program from:

(1) source files under group FIELD

(2) object files under group FIELD

(3) object files under group UTIL

(4) object files under user JIMKLD, group FIELD

for the sake of brevity, single character filenames will be used.

```
GROUP='FIELD';
!&MODULES=JOIN(        /LIST OF MODULES:
    APPLY<SOURCEFILE,('A','B')>, / (1)
    APPLY<OBJECTFILE,('C','D','E')>, / (2)
    APPLY<OBJECTFILE,('F','G'):GROUP='UTIL'>,/(3)
    APPLY<OBJECTFILE,('H','K'):USER='JIMKLD'>);/(4)
!&PROG=LINK &MODULES;  /
```

(b) As a second example, suppose we wish to form the modules in the previous example into a new subroutine library, named FIELD, where the entry names are the file names of the modules (i.e. A,B,---,K).

```
    GROUP='FIELD';
!&ENTRIES=JOIN(      /LIST OF OBJECT MODULES TO BE
                     /ENTERED IN THE LIBRARY
    APPLY<COMPILE,(APPLY<SOURCEFILE,('A','B')>)>, /(1)
    APPLY<OBJECTFILE,('C','D','E')>,/ (2)
    APPLY<OBJECTFILE,('F','G'):GROUP='UTIL'>,(3)
    APPLY<OBJECTFILE,('H','K'):USER='JIMKLD'>);/(4)
CATALOGUE(<>, &ENTRIES:NEW=YES);/CREATE LIBRARY
LIBRARY &ENTRIES; /AND DEFINE IT
```

5.8 Transparent Insertion of JCL

5.8.1 TRANSPARENT

Purpose: to insert target JCL directly, allowing a job to make use of target facilities not accessible via GCL.

Parameters: none.

Value returned: none.

Action: following lines in the input stream are inserted at the current position in the GCL-generated JCL until a line starting with the characters specified by the TERM option (default '*/**' specifying a /* terminator) is encountered. The next characters input by JOLT will be those immediately following the terminator.

Options: TERM.

Caution: transparently inserted JCL is completely unchecked; its use requires a good understanding of the target JCL. If in doubt it is as well to examine generated JCL to check that the combined effect of transparent and GCL-generated JCL is what you require.

[Example: to overcome the current omission of paper tape punch facilities in Multijob JCL:

```
TRANSPARENT ();
// SCHEDULE MICCSS:UTPROG.SOPTP,5
// CONFG STORE=3,RSP=2E10,PP=1
// FILE PTP,PP,*
// FILE READ,RA,XXX(S)
// EXEC
/*  /  ]
```

[5.8.2 RTP]

Purpose: to allow transparent insertion of Multijob run time parameters for a job scheduled with GCL.

Parameters: none.

Value returned: none.

Action: RTP affects the program most recently scheduled via RUN, RUNJOB or RUNB; its effect is to insert, at the appropriate place in the generated JCL, a // PARAM line followed by lines read from the GCL input stream as for TRANSPARENT.

Options: TERM.

Example:

```
RUN(P, <I>, <PRINTER>);
RTP();/ RUN TIME PARAMETERS FOR P
FIRST PARAMETER
SECOND PARAMETER
/*
```

[5.8.3 NOWTRIALS]

Purpose: to allow the transparent insertion of Multijob Trials statements.

Parameters: none.

Value returned: none.

Action: if a Trials run has been initiated by GCL then text defined by the NOWTRIALS statement (terminated, as usual, by TERM) is inserted following Trialsstatements inserted by GCL. If no Trials run is current then a Trials run is initiated and the text is inserted between bracketing // TRIALS and // ENDTRIALS statements.

Options: TERM and, if a Trials run has to be initiated, CLTIME, CLSTORE.

Example: to make use of the Usercode *DUMP option (not available in GCL):

```
NOWTRIALS(); /TRIALS STATEMENTS FOLLOW
// UCODE COMP.ANAL
// OPTION *DUMP
/*  / (TERMINATOR)
```

CHAPTER 6

[MULTIJOB OPTIONS, AVAILABLE SETTINGS AND SYSTEM DEFAULTS]

GCL option defaults for the Multijob implementation at Culham are set up from GCL statements in the file

RJDSUI:GCL.OPTION(S)

which includes comment explaining the significance of each option and available settings. The file also contains some syste limits and standard settings which are applied to generated JCL in accordance with Culham stream configurations and operating conventions.

This file has public read access to allow users to keep themselves informed; a listing of the current version is given below.

```
    LK GCL.OPTION,,,U

/++++++++++++++++OPTIONS - SETTINGS AND DEFAULTS+++++++++++++++++
/
/
/
/========== (1) - COMPILE OPTIONS
/
!LANGUAGE = FORTRAN;/                   SOURCE CODE LANGUAGE- ALSO :
/                                         =ALGOL, COBOL, UCODE OR LSD
!CLTIME  =20;/                          TIME LIMIT FOR COMPILE & LINK
/                                         IN ETU (1 ETU =3.5 SEC. CPU)
!CLSTORE =180;/                         STORE FOR COMPILE & LINK EDIT
/                                         IN 512 BYTE UNITS
!SOURCELIST =YES;/                      COMPILER SOURCE LISTING
!OBJECTLIST =NO;/                       COMPILER OBJECT CODE LISTING
!DEBUG    =NO;/                         COMPILER ETC. DIAGNOSTICS
!REFERENCES =NO;/                       =YES FOR COMPILER SYMBOL TABLES
/
/=====(1.1)- FORTRAN OPTIONS
!MANYNAMES =NO;/                        =YES FOR LARGE TABLE COMPILER
/
/=====(1.2)- ALGOL OPTIONS
!ENTRY    =@$1 4;/                      ENTRY NAME(DEFAULT IS FILENAME)
!ALGBUG   ='ROUTE,ASSIGN';/             ALGOL DEBUG FACILITIES
/                                        (INVOKED IF DEBUG= YES )
/
/=====(1.3)- COBOL OPTIONS
!COBMAP   ='MAP,XREF';/                 COBOL REFERENCES SPECIFICATION
/
/=====(1.4)- USERCODE OPTIONS
!MACLIB   ='SYSTEM';/                   MACRO LIBRARY
/
/=====(1.5)- LSD OPTIONS
!GENTIME =50;/                          MACROGENERATION TIME LIMIT (ETU)
!LSDD     ='DECL';/                     LSD GLOBALS IN USER:GROUP.LSDD(S)
!LSDE     ='EXTS';/                     EXTPROCS IN USER:GROUP.LSDE(S)
!LSDSTACK=100;/                         SIZE OF LSD STACK
!MAIN     =YES;/                        =NO IF NOT MAIN MODULE
!LSDU     ='MICCSS';/                   USER NAME FOR LSD ROUTINES
!LSDG     ='NEWLSD';/                   GROUP FOR LSD ROUTINES
!LSDP     ='PROG';/                     LSD PROGRAM NAME
/
/
/========== (2) - LINK EDIT OPTIONS
/
!PROGMAP =YES;/                         PROGRAM MAP (SEE MAPLEVEL)
!MAPLEVEL ='MAP';/                      THIS GIVES MODULE MAP-
/                                         ='XREF' ADDS CROSS REFS.
/                                        (NO EFFECT IF PROGMAP =NO)
```

```
!LET      =YES;/                      =NO FAILS LINK EDIT IF ANY
/                                       UNSATISFIED REFERENCES
!ERREX    =NO;/                       =YES FOR ERROR EXIT FACILITY
!EXTO     ='DUMMY';/                  ERROR EXIT LABEL (LINKED TO
/                                        UNSATISFIED REFERENCES)
!PROGSAVE =@!SCRATCH();/              FILE TO HOLD LINKED PROGRAM
!LIBLIST =();/                        LIST OF SUBROUTINE LIBRARIES
/                                       TO BE USED IN LINK EDIT
/
/
/========== (3) - PROGRAM EXECUTE OPTIONS
/ .
!RUNTIME =20;/                        TIME LIMIT- ELAPSED TIME UNITS
/                                       (1 E.T.U. = APPROX. 3.5 SECS.)
!STORE    =130;/                      STORE REQUIRED (512 BYTE UNITS)
!INTERACTIVE = NO;/                   =YES IF INT'VE OR VERY SHORT
!JOURNAL =!DISPLAY;/                  ACTION TAKEN ON PROGRAM JOURNALS
/                                       ALSO: =DELETE, RETAIN, PRINT
/
/=====(3.1)- SOCKET LISTS FOR RUN FUNCTION
!FORTRANIN=(97,5,8);/                 FORTRAN INPUT DATA SETS
!FORTRANOUT=(99,98,6,7,9);/                   OUTPUT
!ALGOLIN =(220,221,222);/             ALGOL INPUT CHANNELS
!ALGOLOUT =(230,231,232);/                  OUTPUT
!COBOLIN =(1);/                       COBOL INPUT
!COBOLOUT =(3);/                            OUTPUT
!UCODIN =(0,2,4);/                    USERCODE INPUT- FILE0,FILE2,..
!UCODOUT =(1,3,5);/                            OUTPUT
!LSDIN    =(5,6);/                    LSD INPUT- READ,MERGE
!LSDOUT   =(10,0);/                        OUTPUT- PRINT,PUNCH
!INLIST =@(FORTRANIN,ALGOLIN,COBOLIN,UCODIN,LSDIN)LANGUAGE;/ I/P SOCKETS
!OUTLIST =@(FORTRANOUT,ALGOLOUT,COBOLOUT,UCODOUT,LSDOUT)LANGUAGE;/ O/P
/
/
/========== (4) - SUBROUTINE LIBRARY (CATALOGUE FUNCTION) OPTIONS
/
!NEW      =NO;/                       =YES IF NEW LIBRARY BEING FORMED
/
/
/========== (5) - IN-LINE TEXT OPTIONS
/
!TERM     ='*/**';/                   LINE STARTING WITH TERM IS
/                                       TEXT TERMINATOR (DEFAULT /*)
/
/
/========== (6) - PRINTER LISTING OPTIONS
/
!NUMBERED =NO;/                       =YES FOR LINE NOS. ON LISTINGS
!BIGFILE =NO;/                        =YES IF IT WILL EXCEED
/                                       NORMAL PRINT LIMIT
!PRINTQ   = @IF<BIGFILE,'*/2','*/1'>;/  ='*/3' ETC. FOR PRINT Q 3 ETC.
```

```
/
/
/========== (7) - JCL GENERATION OPTIONS
/
!JCLOUT   ='JCLOUT';/                     JCL SENT TO FILE-
/                                           USER:GROUP.JCLOUT(S)
/                                           JCLOUT IS UP TO 6 CHARS.
!SETGCL   ='SETGCL';/                     SETUP FILE (INSERTED BY JOB):
/                                            USER:GROUP.SETGCL(S)
/                                            (UP TO 6 CHARS.)
!SYSTEM    =MJA;/                         =MJB, MJC FOR CULHAM SYSTEMS B, C
/
/
/========== (8) - MULTIJOB FILE SPECIFICATION OPTIONS
/
!RUNNO     =0;/                           RUN NO. FOR FILES & PROGRAM RUNS
/                                           (0 TO 999, OR =NORUNNO IF NONE)
USER      =NULLSTR;/                      USER NAME (6 CHARS.)
/                                           DEFAULT IS SET BY JOB FN.
GROUP     =UTPROG;/                       GROUP NAME (UP TO 6 CHARS.)
!FTYPE     =TYPES;/                       TYPE CODE FOR DATAFILE & SCRATCH
/                                           =TYPES FOR DATA OR LSD SOURCE,
/                                           =TYPEF FOR FORTRAN SOURCE CODE
/                                           =TYPEA FOR ALGOL SOURCE,
/                                           =TYPEC FOR COBOL SOURCE,
/                                           =TYPEU FOR USERCODE SOURCE,
/                                           =TYPEY FOR COMPILED MODULE,
/                                           =TYPEP FOR LOADABLE PROGRAM,
/                                           =TYPEN FOR PARTITIONED ACCESS,
/                                           =TYPEZ FOR DEDICATED FILES
!VSPEC    =NO;/                           =SPVOL IF VOLUME SPECIFIED,
/                                          =VSEQ IF SEQUENCE NO. OF
/                                            MULTI-VOLUME FILE SPECIFIED
!VOL      =8;/                            VOLUME OR VOLUME SEQ. NO.
/                                           (N/A IF VSPEC=NO)
!TRSPEC   =NO;/                           =TRACK/CYL IF TRACK/CYLINDER
/                                           INCR. SPECIFIED (TYPEZ ONLY)
!TRCYL    =4;/                            TRK/CYL INCREMENT (N/A IF TRSPEC=
          NO)
/
/
/========== (9) - LOWER LEVEL MULTIJOB OPTIONS
/
!TRIALS   = NO;/                          =YES IF TRIALS RUN CURRENT
/                                           (GENERALLY SET AUTOMATICALLY)
!STREAM    =@(IF<TRIALS,1,IF(INTERACTIVE,2,5)>,/ MULTIJOB STREAM:
/                                           SYSTEM A - TRIALS IN A,
/                                           INTERACTIVE IN B, ELSE E
          IF<INTERACTIVE,2,1>,/            SYSTEM B- INT'VE IN B, ELSE A
          1)SYSTEM;/                       SYSTEM C- ALL IN STREAM A
/                                            =1,2,... FOR STREAMS A,B,...
!TRNO     =0;/                            INITIAL RUN NO. FOR TRIALS
```

```
!CBRUN     =0;/                          INITIAL RUN NO. FOR CALLBACK
!DEVICE   =DISC;/                        DEVICE TYPE
!LSDRUNNO =0;/                           INITIAL RUN NO. FOR STAGE2
/                                          & LSDPP
!PRUN     =0;/                           INITIAL RUN NO FOR PRINT FILES
/
/
/
/++++++++++++++++++++SYSTEM LIMITS & STANDARD SETTINGS++++++++++++++++
/
!MAXRUNTIME =CONST @< /                  MAXIMUM TIME LIMITS (ETU) -
          (30,20,100,100,250,100),/        STREAMS A TO F (SYSTEM A)
          (1000,20,100),/                     A TO C (SYSTEM B)
          (1000,100)>SYSTEM STREAM;/          A & B (SYSTEM C)
!MAXSTORE=CONST @< /                     MAXIMUM STORE SIZES -
          (180,200,36,32,400,68),/         STREAMS A TO F (SYSTEM A)
          (684,138,128),/                         A TO C (SYSTEM B)
          (996,68)>SYSTEM STREAM;/                A & B (SYSTEM C)
!RANK      =CONST 2;/
!PRIORITY =CONST@< /                     PRIORITIES - STREAMS A,B,...
          (12,13,10,10,6,10),/             SYSTEM A
          (12,13,10),/                     SYSTEM B
          (12,13)>SYSTEM STREAM;/          SYSTEM C
/
***END
?//
```

CHAPTER 7

[HOW TO USE THE MULTIJOB VERSION OF JOLT]

7.1 Operation from a Terminal

An interactive JOLT run is initiated by the console command JOLT. The form of the JOLT command is:

```
?// JOLT filename[,,,run]
```

where filename is a file containing the GCL statements to be translated. Further parameters can follow, the specification of these being identical to those for the RUN command (see Multijob Remote Terminals manual); the only parameter which you should ever need to use is run number (to avoid clashes with other people under the same username).

On completion of the translation JOLT types the message

```
GCL OK
or
TRANSLATION FAILED
```

as appropriate. In the latter case the failure messages and generated JCL are available for inspection in the file UTPROG.GCLERR(S).

If translation is successful the generated JCL can be inspected in the JCL output file (see Section 5.1.2). It can then be submitted to the system for execution via a RUN or REMJOB command. This is a temporary arrangement - a more automatic mode of operation will be implemented when more experience has been built up.

Non-interactive execution of JOLT can be initiated via the REMJOLT command, which has the same parameters as JOLT, but sends termination messages to the JOLT journal.

7.2 Failure Messages

Any error which is detected in a GCL job causes the generation of a message of the form:

```
FAILURE f LINE n LAST IDENTIFIER READ: ident
 LAST 4 IDENTIFIERS ACCESSED: ident ident ident ident
```

where f is an integer identifying the reason for the failure, which was detected on the nth line read by JOLT after reading the specified identifier (denoted by ident). The line count normally includes all lines read from substituted files; Section 5.1.3 shows how lines from the setup file, automatically inserted by the JOB statement, can be excluded from this count. The second line of the failure message is an implementation diagnostic aid which is of little significance to users.

There are some "suspicious circumstances" (such as identifiers longer than 12 characters) which also give rise to error listings but do not cause the translation to fail. In such a case WARNING replaces FAILURE in a message of the above form.

All failure and warning messages are sent to the file UTPROG.GCLERR(S).

Reasons for failures or warnings are given (at Culham) in the file RJDSUI:GCL.FAIL(S) in which line numbers correspond to failure or warning numbers. Thus, if UTPROG.GCLERR(S) contains a line starting

FAILURE 12 ---

then the terminal command

LOOK RJDSUI:GCL.FAIL,12

will tell you why.

7.3 Keeping Up to Date

GCL has been implemented in a way that makes it relatively easy to extend its facilities and change their form. Every attempt will be made to react appropriately and promptly to user suggestions. It is to be hoped, therefore, that corrections, improvements and extensions will occur at a rate that revisions of this manual could not hope to match in the early stages.

The resulting information problem is to be tackled on three fronts:

i) where practicable, changes will not invalidate the specifications in this manual,

ii) up-to-date information on options, defaults and system constants, in the same form as Chapter 6, will be available at Culham in the file RJDSUI:GCL.OPTION(S). Since the information in this file is actually used to set up JOLT, its authenticity is guaranteed.

iii) other information will be given in the file RJDSUI:GCL.NEWS(S).

7.4 How to Scream

Users are invited to participate in the interactive development of GCL. Culham users should refer all questions, comments, suggestions and bouquets to:

Bob Dakin
Ext. 6133
Room G13,E6

For users outside Culham Laboratory the full address is:

Dr R J Dakin
UKAEA
Culham Laboratory
Nr Abingdon
Berks

Telephone Abingdon 1840, Ext. 6133

APPENDIX 1

How to Write GCL Functions

It is possible for you to write your own GCL functions to perform recurring tasks. Function definitions can be included in your setup file, invoked automatically by JOB, in which case they become, in effect, private extensions to GCL. A group of users can share a set of file definitions maintained in a single file by including appropriate SUBST statements in their setup files.

A1.1 The Form of a Function

A function is a sequence of GCL statements bracketed by @ and # symbols. [In Multijob either the teletype # or the punched card/line printer #, which corresponds to the teletype character\(shift L) can be used.] Statements in the function are separated by semicolons as usual, but the terminating semicolon for the final statement (immediately before the terminating #) can be omitted.

A function constitutes a single syntactic unit. It usually forms the right hand side of an assignment statement in which case it must be followed by a semicolon to terminate the statement.

Execution of a function automatically terminates at the end of the function; no special RETURN statement is required. The value returned by the function is the value returned by the final statement in the function.

Example: the RUNJOB function is defined by the statement:

```
! RUNJOB = @ RUN (<TEXT()>, <TEXT()>, <PRINTER>)#;
```

A 1.2 Reference to Call Parameters

The first, second --- parameters of a call on a function are referred to inside the function as Ø1, Ø2, ---.
Call parameters are evaluated once only, before the function is entered.

[Example: Suppose a user wants to define a number of object files under the Multijob user name CLIXXX (not his own), group

ROUTE and run number 237. This could be coveniently handled by creating a function called &CLFILE as follows:

```
! &CLFILE = @OBJECTFILE($1:USER = 'CLIXXX',
                  GROUP='ROUTE', RUNNO=237)#;
/ EXAMPLE OF CALL ON THIS FUNCTION:
!A= &CLFILE('ANAL'); / DEFINES A TO BE THE FILE
/           CLIXXX:ROUTE.ANAL(Y2370)   ]
```

A 1.3 Assignment Scope

The scope of GCL assignments is governed by the following rule:

> an assignment is only effective for the duration of the statement or function in which it occurs; thereafter the identifier concerned reverts to its original setting.

This rule provides the option default override behaviour already described, and can be used in a number of other ways. For example an identifier, used to hold intermediate values in one function, can be used for the same purpose in other functions without any risk of mutual interference, since the above rule implies that assignments inside a function have no external effects. The two identifiers TEMP and TEMPA are used by system functions to hold intermediate results and are available to users for the same purpose.

Another implication of the assignment scope rule is that you cannot write a function that uses ordinary assignment to initialise variables - since all assignments are nullified on exit from the function.

Since all identifiers obey the assignment scope rule there is no essential difference between options and other identifiers - your own identifiers can be referred to inside your own functions and make use of the normal default setting (ie. assignment outside a function) and override facilities.

A 1.4 Other facilities

A number of other facilities such as loops and conditionals are available in GCL but are outside the scope of this manual.