Edinburgh Regional Computing Centre

2980 User's Guide

A description of the user facilities available on the Regional Computing Organisation's ICL 2980 at the Bush Estate, Edinburgh.

First Edition
April 1977

EDINBURGH REGIONAL COMPUTING CENTRE

2980 User's Guide

A description of the user facilities available on the Regional Computing Organisation's ICL 2980 at the Bush Estate, Edinburgh.

CONTENTS

FIGURES

PREFACE

This version of the Guide is a complete revision of the original Preliminary Edition, issued in August 1976, and contains a large amount of new material. Numerous people have assisted in the preparation of this version; in particular Section F relies very heavily on a document produced by G.E. Millard. The first five sections received fairly wide circulation on a draft form, and have benefitted from much helpful criticism; the remainder of the Guide, to save time, has received much less checking and no doubt contains a number of obscurities, if nothing worse. Any errors, omissions and lack of clarity are of course my responsibility, and I should welcome notification of these, and any other comments. A page for this purpose is included at the end of the Guide.

Thanks to the efforts of the ERCC Documentation Officer, J.M. Murison, and his staff, the full text of the Guide is held as a file on EMAS from which formatted and paginated listings are obtained for reproduction. This system will make the production of updates far simpler and more effective than for the original version.

This version has been revised to reflect changes resulting from the introduction of the SV21 release of VME/B.

P.E. Williams
September 1977

A. GENERAL INFORMATION

This Guide describes the user service available on the ICL 2980 computer at the Bush Estate, just outside Edinburgh. The information is liable to change and it is expected that a considerable number of updates to this Guide will be issued. The service uses a standard ICL operating system, VME/B, which is itself in the course of rapid development, and users should not assume the existence of any facilities other than those described in this Guide.

The 2980 is a large machine, potentially as powerful as the IBM 370/168 at NUMAC, and possessing the same types of disk and magnetic tape units. Like the IBM machine, the 2980 is byte-oriented but the machine architecture is very different and the facilities available to users are presented in an entirely different way.

ACCESS

Access is by means of remote job entry terminals in each university, details of which are available from the local Computing Service.

USER REGISTRATION

All users of the 2980 must be registered in the catalogue of the operating system. The procedure is as follows.

* Apply to your Computing Service for a user number, if you do not already have one.

* If you have a user number, apply to your Computing Service to be registered as a user of the 2980. They will advise you when you have been registered with the 2980 and can start to submit jobs.

DOCUMENTATION

It is intended to provide the average user with all the documentation necessary to use the 2980 service, with the exception of certain language manuals. Some users, however, may require more advanced facilities and will need to consult their local Advisory Service. The Computing Service has a set of ICL manuals pertaining to the 2980 operating system; these may be of interest but sometimes describe facilities which have not yet been implemented or which are not supported locally.

SUBMISSION OF JOBS

All jobs for the 2980 must be accompanied by the 2980 Job Request Card, shown in Figure A1. The information on the card is for the benefit of the Job Reception staff involved: it is not read by the Computer Operators. The card accompanies the job through the card reader: please see that it is in good condition.

ACCOUNTING

At the end of the job a number of measures of the job's use of resources is printed. These measures are

* OCP time (in milliseconds) used by the job, including any time spent in supervisor on behalf of the job

* Virtual store interrupt transfers to and from the drums or discs. These are equivalent to page turns on other systems and occur every time a page of store associated with the job is written out to drum or disc because it has not been accessed recently, or whenever a page is brought back into store because the job has referred to it. Normally transfers are between store and drum; transfers to disc occur when the drums are full or when the page is part of a program segment larger than 64K bytes.

* Filestore Transfers - data transfers specified by the job. These include any transfers associated with program loading, etc.

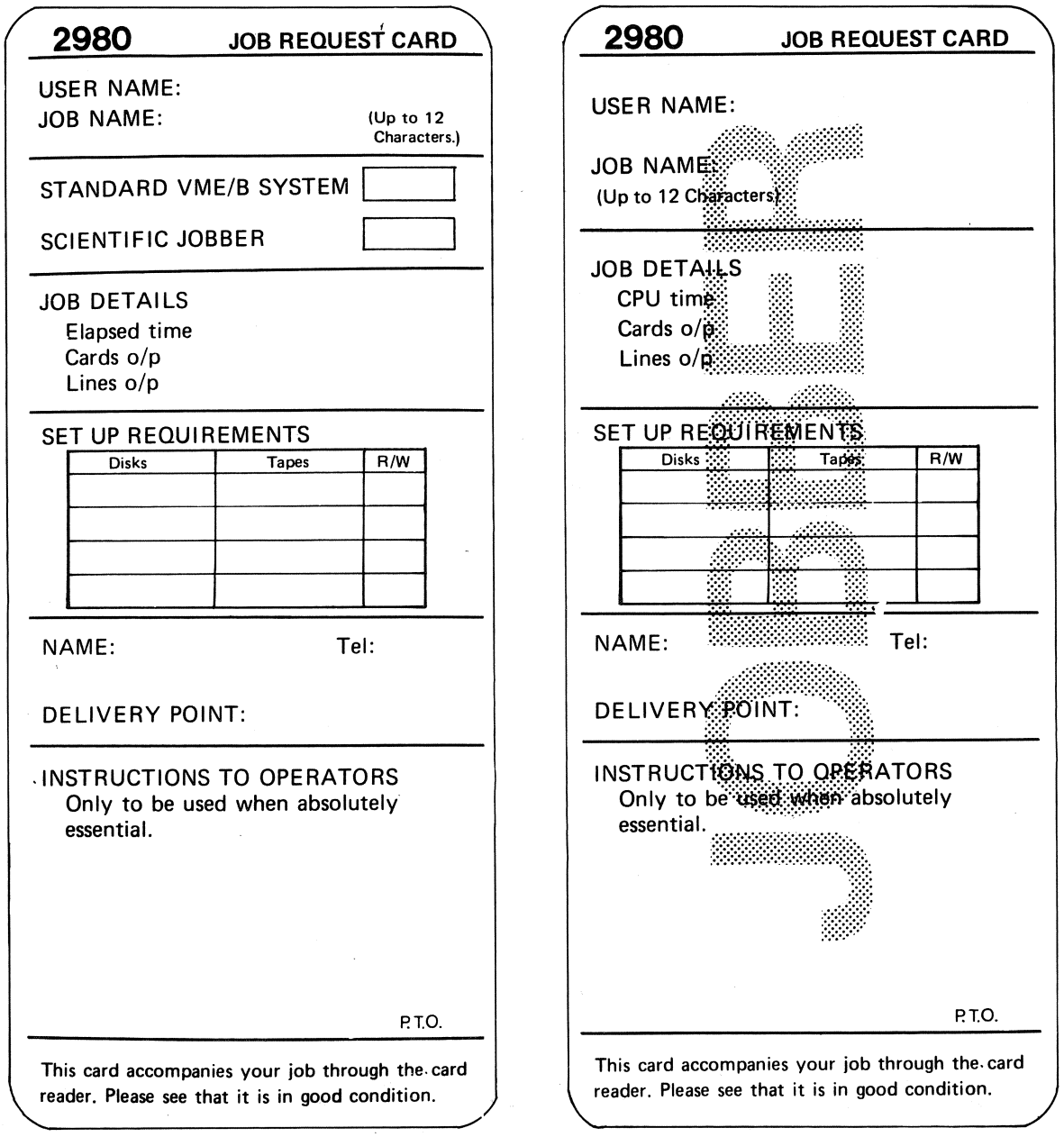
* Main store occupancy is in kilobyte-seconds and, if divided by the elapsed time for the job, gives a measure of the average store used by the job. The elapsed time for a job is of course very variable.

From these a charge, "MONEY", is calculated. The units are pence. Other measures (TRUS, etc.) should be ignored.

ALERT INFORMATION

A file is kept on EMAS containing information on recently discovered faults and restrictions in the operating system and compilers, and any important errors in documentation. The file can be listed using the commands ALERT2980 or ALERT2980(.LP), which are contained in the library CONLIB.GENERAL.

The file is also held on the 2980 and may be listed to a line printer with a LIST or LIST_FILE command, described in Section E. The file name is :ERCI22.ALERT. There are, however, terminals in all three universities which can connect to EMAS.



2980 JOB REQUEST CARD

USER NAME:

JOB NAME: (Up to 12 Characters.)

STANDARD VME/B SYSTEM ☐

SCIENTIFIC JOBBER ☐

JOB DETAILS
Elapsed time
Cards o/p
Lines o/p

SET UP REQUIREMENTS

Disks	Tapes	R/W

NAME: Tel:

DELIVERY POINT:

INSTRUCTIONS TO OPERATORS
Only to be used when absolutely essential.

P.T.O.

This card accompanies your job through the card reader. Please see that it is in good condition.

2980 JOB REQUEST CARD

USER NAME:

JOB NAME:
(Up to 12 Characters)

JOB DETAILS
CPU time
Cards o/p
Lines o/p

SET UP REQUIREMENTS

Disks	Tapes	R/W

NAME: Tel:

DELIVERY POINT:

INSTRUCTIONS TO OPERATORS
Only to be used when absolutely essential.

P.T.O.

This card accompanies your job through the card reader. Please see that it is in good condition.

Figure A1 2980 Job Request Card

B. GENERAL DESCRIPTION OF SYSTEM

The hardware configuration of the 2980 is shown in Figure B1. The following is a brief description of some of the major components.

CENTRAL PROCESSOR AND STORE

The central processor (or more properly the Order Code Processor - OCP) accesses 2M bytes of store via a Store Access Controller (SAC), the store being divided into 4 blocks of 512K bytes. In raw power the 2980 is comparable with the 370/168 at NUMAC but the operating system and other software are very different and at present no generally reliable guide to relative throughputs can be given. Until experience has been gained it is probably sensible to assume a throughput similar to that of the 370/158 - rather more than on the 360/65. The store is paged and program size is less of a restriction than on the IBM operating system at NUMAC. The fixed head disks are used exclusively by the operating system for paging.

DISC STORAGE

The 2980 has 16 EDS-100 disc drives, similar to those at NUMAC, with an effective capacity of about 95M bytes each. These drives are used to hold the operating system and any spooling space it requires, the rest of the space being available for user files. Initially only permanently mounted files will be available.

Data is recorded on a disc in blocks of a fixed size, the block size being 2048 bytes. Data is read and written by programs in units of logical records of a size determined by the user, the physical layout of that data on disc being dealt with by the operating system. At present the maximum record size may not exceed 2036 bytes and users whose record size would necessarily exceed that should contact their Advisory Service for assistance.

MAGNETIC TAPES

The system has 8 nine track 1/2" magnetic tape drives operating at 1600 byte/inch in phase encoded mode only. They will accept tapes with or without self-loading collars. Initially it will not be possible for data on tape to be read or written by a user program if its block size exceeds 2048 bytes.

How to get tapes

Users can obtain tapes through the local Computing Service, either by purchase or on loan or rental. Users may purchase tapes direct from a manufacturer; if they wish to do so they should obtain a specification of the type of tape required by the 2980 drives from their Computing Service.

Tapes must have clean uncrumpled ends with the beginning-of-tape marker between 10-30 feet from the start. The end-of-tape marker should be at least 25 feet from the end. Tapes should normally be 2400 feet long since these can be loaded automatically. A 2400 foot tape will hold about 30M bytes of data.

A new tape must be labelled by the operating system before use. This will be done by the 2980 operators who will give it a label of the form aannnn. 'nnnn' is the storage rack position and will be assigned; 'aa' are two alphabetic characters of the user's choice which provide some security against the selection of the wrong tape. Users will be advised when their tape has been labelled and can be used.

CARD READER AND PUNCH

The 2980 has a card reader and punch for standard 80 column cards using the EBCDIC code shown in Figure B2. This code is similar to that used on IBM machines, i.e., a particular set of column punches is represented in hexadecimal in the same way on both types of machines, but the relationship between column punches and a particular graphic differs somewhat. Letters and digits are represented in the same way; the relationship between the internal representation, the card code and the graphic in the case of special characters is shown in Figure B3.

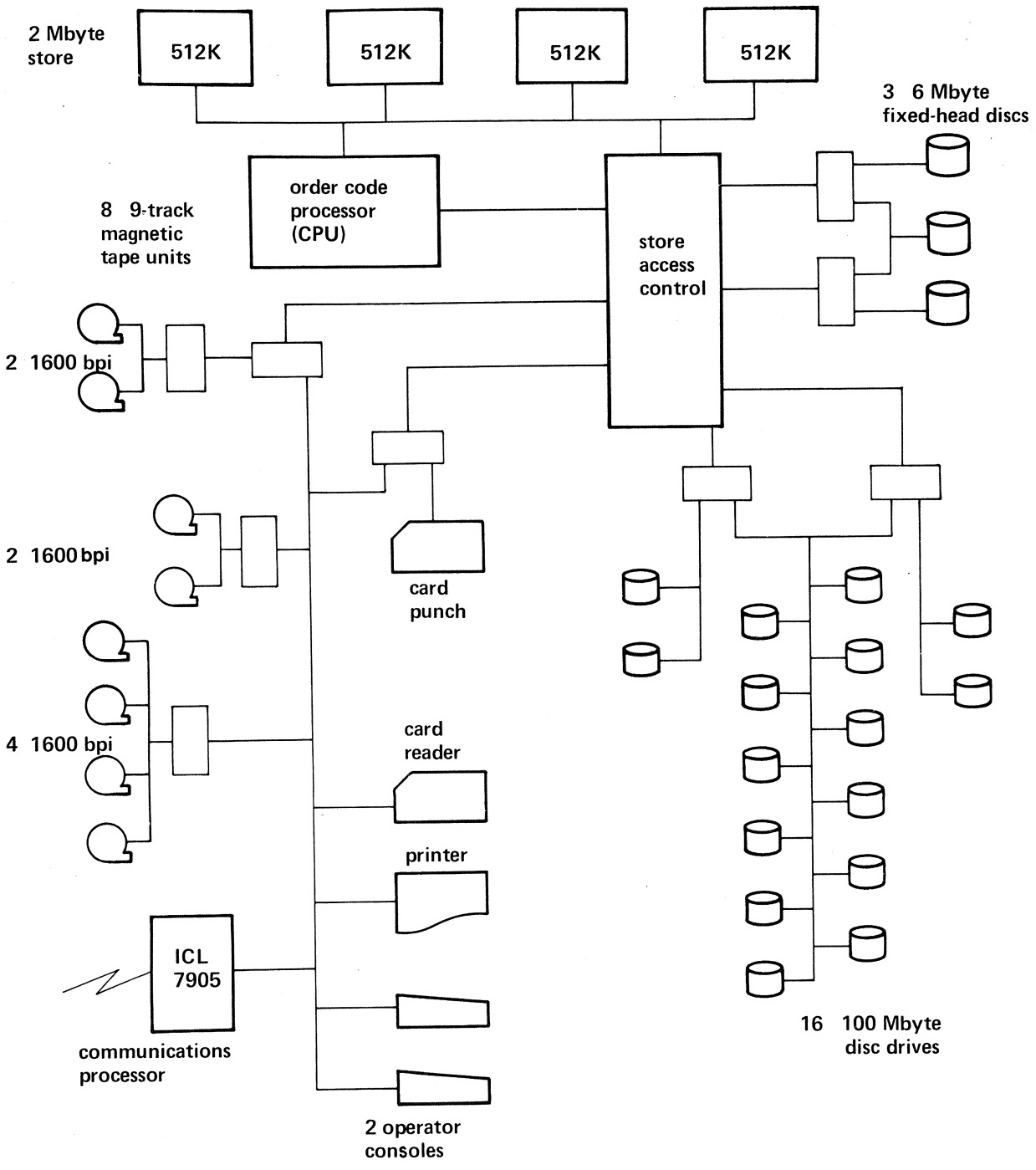


Figure B1 Configuration of the ICL 2980

PRINTER

The line printer has an extended character set including lower case characters. In addition to digits 0-9 and alphabetic characters (upper and lower case), the character set includes those graphics shown in Figure B3. There is also a set of other special characters for mapping and other purposes, details of which can be obtained from your local Advisory Service. The line has 132 print positions and spacing is normally at 6 line/inch though 8 line/inch and 10 line/inch (the latter providing a square grid) are available for special purposes.

OPERATING SYSTEM

ICL's operating system VME/B is used to provide a service in batch mode only. Not all parts of VME/B have been mounted and users should not assume the existence of facilities not described in this Guide. In any case only those facilities described here are supported by Advisory Services. Compilers are available for FORTRAN (ICL New Range and IBM-compatible), ALGOL (ALGOLE), IMP and COBOL. All except IMP are ICL-supported.

Scientific Jobber

IBM-compatible FORTRAN - FORTRAN (G), ALGOL (E) and IMP programs can be compiled and run in a batch mode (like WATFIV on the NUMAC machines) using a sub-system called the Scientific Jobber. This avoids some of the overheads of the main operating system and is intended to provide a more efficient vehicle for the development of programs. Work running under the Scientific Jobber uses a simplified job control language, different from that used by VME/B, which is described later. The Scientific Jobber is ICL-supported.

CHARACTER CODES

As already mentioned, the basic internal character code used by the operating system, which drives all input/output devices, is 2900 EBCDIC. This code is also used by COBOL and ICL New Range FORTRAN programs. Other compilers, IBM-compatible FORTRAN - FORTRAN (G), ALGOL (E) and IMP, operating either independently or within the Scientific Jobber, use ISO code for the internal representation of characters, though there is an EBCDIC option for FORTRAN (G). This is of no consequence to users except in a few fairly uncommon instances. These include

* Situations where data is written out in binary (i.e. unformatted) and read back as character data (e.g. formatted data in FORTRAN). Data written out as character data is translated to EBCDIC, the external form of character data always being EBCDIC. Data written out as binary information has a significance known only to the program which produced it, and is not translated in any way. Thus if information which in fact represented character data were written out as binary information by, say, an ALGOL (E) program, the characters would be in ISO code. If that data were read back as binary information by an ALGOL (E) program no translation would be performed on input and the program could treat the data as character data. If however the program read the data in character mode a translation from EBCDIC to ISO would take place, rendering the data meaningless to the program.

* Instances where a program makes use of a knowledge of the internal code used to represent characters. This is in any case an undesirable programming practice, but will usually only cause trouble if such a program is compiled again using a different compiler; e.g. if a program making use of the internal code representation was developed using FORTRAN (G) (say in the Scientific Jobber) and then recompiled for production running using the ICL New Range FORTRAN compiler.

* Programs which mix modules compiled using FORTE and ICL New Range FORTRAN.

This situation is exactly the same as existed on the IBM 370/158, and still exists at NUMAC, with the IMP and FORTE compilers. With IMP it never causes trouble; with FORTE it usually only (and then rarely) causes trouble when a program also uses IBM's FORTRAN compilers. So far as use of the FORTRAN (G) compiler is concerned, any problems can be avoided by using the appropriate version of the compiler, which can run in either EBCDIC or ISO mode. This is covered in Sections F and G. The choice of ISO is a conscious one: ISO is used by nearly all computer manufacturers except IBM; it is used by nearly all common terminals and minicomputers; and it has a more convenient and efficient layout of characters than EBCDIC. The 2900 range is ISO oriented in that the representation of characters on cards follows an international

standard (that is why there are some differences from the IBM standard), but unfortunately it copies the IBM standard for the translation of card hole patterns to internal code. The net result is a 2900 EBCDIC code which is different from IBM's and that used by ICL on System 4.

The ISO/EBCDIC translation used is shown on Figure B4. It will be noted that the ISO code has been extended to 256 characters by local decision; the upper 128 characters are not part of a wider standard.

Users should note that the collation sequence in ISO is different from that in EBCDIC. In ISO digits precede letters; in EBCDIC digits follow letters.

ROUTING OF OUTPUT

Normally all printed output from a job is sent to the printer associated with the device through which the job was input. For jobs input through the 2980 card reader, or transferred to the 2980 on tape, this is the 2980 printer; for jobs input from an RJE terminal it is the terminal printer; for jobs transmitted from EMAS it is the EMAS printer. It is possible to change this routing by specifying a DEVICE parameter in the JOB macro (Section D) or, for individual print files, in the relevant LIST or LIST_FILE macro (Section E).

Possible output devices are as follows. Only line printer output can be routed at present.

LP10	2980 Line Printer (Bush)
LP14	Buccleuch Place (Satellite One)
LP15	Science Faculty (JCMB)
LP16	S.I.A.E. (Bush)
LP17	ABRO (KB)
LP18	College of Agriculture (KB)
LP19	ARC Unit of Statistics
LP23	EMAS slow devices machine (KB)
LP24	ARC Rothamsted
LP25	Medical Computing Group (Edinburgh)
LP26	Social Science Faculty (WR Building)
LP27	Animal Diseases Research Association
LP28	MRC Cytogenetics Unit (Western General)
LP29	Chemistry (Edinburgh)
LP30	Physics (Edinburgh)
LP33	Scottish Horticultural Research Institute (Invergowrie)
LP34	EMAS
LP35	West of Scotland College of Agriculture
LP38	Buccleuch Place (PDP11/40)
LP40	Alison House TCP
LP41	Appleton Tower TCP
LP64	Heriot-Watt University
LP68	Dundee University
LP72	Stirling University
LP80	Glasgow University (Chemistry Building)
LP89	Strathclyde University

card holes	12	11	0	no punch	12-0	12-11	11-0	12-11-0
9-1	01	11	21	31	41	51	E1	71
9-2	02	12	22	32	42	52	62	72
9-3	03	13	23	33	43	53	63	73
9-4	04	14	24	34	44	54	64	74
9-5	05	15	25	35	45	55	65	75
9-6	06	16	26	36	46	56	66	76
9-7	07	17	27	37	47	57	67	77
9-8	08	18	28	38	48	58	68	78
9-8-1	09	19	29	39	00	10	20	30
9-8-2	0A	1A	2A	3A	CA	DA	EA	FA
9-8-3	0B	1B	2B	3B	CB	DB	EB	FB
9-8-4	0C	1C	2C	3C	CC	DC	EC	FC
9-8-5	0D	1D	2D	3D	CD	DD	ED	FD
9-8-6	0E	1E	2E	3E	CE	DE	EE	FE
9-8-7	0F	1F	2F	3F	CF	DF	EF	FF
8-1	49	59	69	79	80	90	A0	B0
8-2	4A	5A	E0	7A	8A	9A	AA	BA
8-3	4B	5B	6B	7B	8B	9B	AB	BB
8-4	4C	5C	6C	7C	8C	9C	AC	BC
8-5	4D	5D	6D	7D	8D	9D	AD	BD
8-6	4E	5E	6E	7E	8E	9E	AE	BE
8-7	4F	5F	6F	7F	8F	9F	AF	BF
no punch	50	60	F0	40	C0	6A	D0	70
1	C1	D1	61	F1	81	91	A1	B1
2	C2	D2	E2	F2	82	92	A2	B2
3	C3	D3	E3	F3	83	93	A3	B3
4	C4	D4	E4	F4	84	94	A4	B4
5	C5	D5	E5	F5	85	95	A5	B5
6	C6	D6	E6	F6	86	96	A6	B6
7	C7	D7	E7	F7	87	97	A7	B7
8	C8	D8	E8	F8	88	98	A8	B8
9	C9	D9	E9	F9	89	99	A9	B9

Figure B2 2980 Card Code - card holes against internal EBCDIC hexadecimal code.

ISO dec.	ISO hex.	2900 EBCDIC dec.	2900 EBCDIC hex.	2900 printer graphic	card code	IBM 029 stnd graphic (if different)
32	20	64	40	space	no punches	
33	21	79	4F	!	12-7-8	¦
34	22	127	7F	"	7-8	
35	23	123	7B	£	3-8	#
36	24	95	5B	$	11-3-8	£
37	25	108	6C	%	0-4-8	
38	26	80	50	&	12	
39	27	125	7D	'	5-8	
40	28	77	4D	(	12-5-8	
41	29	93	5D	)	11-5-8	
42	2A	92	5C	*	11-4-8	
43	2B	78	4E	+	12-6-8	
44	2C	107	6B	,	0-3-8	
45	2D	96	60	–	11	
46	2E	75	4B	.	12-3-8	
47	2F	97	61	/	0-1	
58	3A	122	7A	:	2-8	
59	3B	94	5E	;	11-6-8	
60	3C	76	4C	<	12-4-8	
61	3D	126	7E	=	6-8	
62	3E	110	6E	>	0-6-8	
63	3F	111	6F	?	0-7-8	
64	40	124	7C	@	4-8	
91	5B	74	4A	[	12-8-2	$
92	5C	224	E0	\	0-8-2	NG
93	5D	90	5A	]	11-8-2	!
94	5E	95	5F	^	11-8-7	¬
95	5F	109	6D	_	0-8-5	
96	60	121	79	`	8-1	NG
123	7B	192	C0	{	12-0	NG
124	7C	106	6A	¦	12-11	NG
125	7D	208	D0	}	11-0	NG
126	7E	161	A1	~	11-0-1	NG

NG = no graphic required

Figure B3 2980 Printer Special Character Set.

FIRST ISO HEXADECIMAL DIGIT

SECOND ISO HEXADECIMAL DIGIT	0	1	2	3	4	5	6	7
0	00	10	40 sp	F0 0	7C @	D7 P	79 `	97 p
1	01	11	4F !	F1 1	C1 A	D8 Q	81 a	98 q
2	02	12	7F "	F2 2	C2 B	D9 R	82 b	99 r
3	03	13	7B £	F3 3	C3 C	E2 S	83 c	A2 s
4	37	3C	5B $	F4 4	C4 D	E3 T	84 d	A3 t
5	2D	3D	6C %	F5 5	C5 E	E4 U	85 e	A4 u
6	2E	32	50 &	F6 6	C6 F	E5 V	86 f	A5 v
7	2F	26	7D '	F7 7	C7 G	E6 W	87 g	A6 w
8	16	18	4D (	F8 8	C8 H	E7 X	88 h	A7 x
9	05	19	5D)	F9 9	C9 I	E8 Y	89 i	A8 y
A	25	3F	5C *	7A :	D1 J	E9 Z	91 j	A9 z
B	0B	27	4E +	5E ;	D2 K	4A [	92 k	C0 {
C	0C	1C	6B ,	4C <	D3 L	E0 \	93 l	6A |
D	0D	1D	60 —	7E =	D4 M	5A]	94 m	D0 }
E	0E	1E	4B .	6E >	D5 N	5F ^	95 n	A1 ~
F	0F	1F	61 /	6F ?	D6 O	6D _	96 o	07

Examples: Graphic J has an EBCDIC hexadecimal representation of D1; the equivalent ISO hexadecimal representation is 4A. ISO X '17' is translated to EBCDIC X '26' which has no graphic associated.

Figure B4 ISO/2900 EBCDIC TRANSLATION TABLE

C. PRINCIPLES OF JOB CONTROL

INTRODUCTION

Work is executed by VME/B in units called jobs. A job can consist of any set of work that needs to be run as a unit. For example, a compilation and test run of a FORTRAN program could be a job. Similarly, the running of a suite of applications programs could also be a job.

Jobs normally involve running programs, e.g. user written programs, packages or system provided programs such as utilities. Each job also needs services from the operating system. For example, the job to run a suite of applications programs will need operating system services to handle the various data files needed by the programs. Both programs and operating system services are items of code.

For each job therefore the system must be provided with information defining which items of code are needed and what actions these items are to carry out. In addition, the actions carried out by the various items must be controlled and co-ordinated. For example, in the application suite job the result of running one program in the suite might be examined before a decision was made whether the remaining programs should be run, and if so in what order.

All these functions are achieved by statements written in the System Control Language, SCL.

SYSTEM CONTROL LANGUAGE

SCL is a powerful block structured high level language like IMP or ALGOL, a full description of which is outside the scope of this Guide. Most users will find that the macros (macro instructions - a set of pre-defined SCL statements) provided by the system are sufficient for their needs, but a few SCL commands may be used for altering the flow of control. Some of these will be described later.

The effect of the block structure of SCL is important and needs to be clearly understood. Jobs are run in a job environment defined by SCL statements provided by the system, the various environments being specified as 'profiles'. These SCL statements constitute an outer block of SCL ('outer SCL'). Resources have scopes defined by the block in which they are declared. This applies to files as well as to more obvious objects like variables and names. The entire job defined by the user - the Job Control Program (JCP) - constitutes a block; inner blocks are delimited by BEGIN and END statements. A file declared within a block is not available outside the block unless it is a permanent file, or is made permanent in the block; a file made temporary within a block is deleted on leaving the block. Jumps may be made from an inner to an outer block, any inner block variables, resources, etc being released. Jumps to an inner block can only be made to the BEGIN statement since any inner block statement labels are local to that block. The full rules may be summarised as follows:

* resources have scopes defined by blocks, which are delimited by BEGIN and END.
* the whole job control program is viewed as a block.
* blocks may be nested but may not overlap.
* names used within a block must be unique.
* if more than one object exists at any time with the same name, only the most recently declared is available.
* entry to a program is treated as entry to an inner block.

Any SCL statement may be given a label. The label follows SCL naming rules (see below) and must be followed by a colon, e.g.

```
LAB1: statement
```

JOB CONTROL PROGRAMS

The work to be run as a job is defined in a JCP, a set of SCL statements of the form

```
JOB (....)
.
.
SCL statements
.
.
ENDJOB
****
```

The four asterisks indicate the end of the JCP and protect the following job. There must be only four asterisks and they must start at the beginning of the line.

The flow of control in the JCP may be varied by statements testing the result of an operation in the job. These will be described later.

Data embedded in a JCP, e.g. a card file input with the job, is called 'alien data' and is dealt with in the section on file handling.

THE CATALOGUE AND NAMING

The system includes a data catalogue which is the one central place where all information about objects in the system is held. Objects of interest here are files and the entries for registered users of the system. The catalogue permits all such objects to be referred to by name, so users do not need to be concerned with any of the underlying physical aspects of the system.

Users

Each user must be registered with the system before he can run a job. User names are the six-character user numbers issued to users by their Computing Services and used to identify them on all the machines to which they have access, e.g. CABL04. When the user name is written in a JCP it must be preceded by a colon, e.g. :CABL04.

Files

Files are identified by file names, which can be up to 31 characters in length, consisting of any alphanumeric characters beginning with a letter. Underline ('break') characters may be included in the name when it is written in a JCP to improve legibility and will be ignored by the system.

The full file name - the hierarchic name - includes the user name of the owner of the file. Thus the full name for file FRED belonging to user CABL04 would be :CABL04.FRED. This is necessary because another user might have a file called FRED. If however FRED is to be used in a job run by user CABL04 it is not necessary for the full name to be specified; FRED is sufficient. The starting colon is needed only when the full name is specified.

The file name may also include a "generation" number. Unless the Scientific Jobber is used to create a file, the attempt to create a file with the same name as an existing file (belonging to the same user) results in a new generation being created. Reference to a file without specifying a generation number implies the most recent generation. By using a negative generation number one can refer to a file relative to the most recent generation. The generation number is enclosed in brackets following the file name, e.g. FRED(3) for the third generation of file FRED. More details are given in Section E.

A file in a library has a full name which includes the name of the library, viz. username.libname.filename. A generation number can be specified for both library and the file, e.g. :CABL04.MYLIB(2).FRED(3).

FORMAT OF SCL STATEMENTS

SCL is essentially a free format language in which spaces between the elements of a statement are not significant. There are however certain rules and conventions which are summarised below.

Statements must start on a new line but may be indented. There are however three delimiter lines which serve to indicate the start and end of data entered with the Job Control Program (---- and ++++) and end of the JCP (****) which must start at the beginning of a line and may not be followed by other characters. Lines should not exceed 80 characters in length.

Spaces may appear between the elements of a statement.

Macro names may if desired be split with break characters (underlines) to improve legibility. The break characters are ignored by the system. For example, ASSIGNFILE could be written ASSIGN_FILE.

Names of variables and labels, which are alphanumeric starting with a letter, may be up to 32 characters long. Break characters may be included but will be ignored.

Statements may be continued on further lines either

* with the explicit continuation symbol +_ (plus underline), which must be preceded by a space. The remaining characters on the line will be ignored, or

* if the line is syntactically incomplete, for example if it ends with a comma or a plus, or as in the following statement. In such cases a continuation symbol is not required.

```
IF X THEN Y
ELSE Z FI
```

Comments may be inserted anywhere that spaces are allowed. A comment starts with @ and ends with end-of-line or another @. If the comment extends over more than one line each line must start with @.

SCL words are reserved (see Appendix 3); i.e. variable names must be different, and must be separated by spaces from other elements of the statement.

Upper and lower case characters are treated as equivalent. Compound characters are not permitted (hence underlining of text is not allowed). Only alphanumeric characters should be used in names.

MACROS

As stated earlier, a macro is a predefined set of SCL instructions. A macro call has the general form

```
macroname (parameters)
```

though a few macros, such as ENDJOB, require no parameters to be specified.

The parameters are all declared by keywords, e.g. NAME=, PROFILE=, with the exception of the jobname in the JOB macro. (In fact any parameter could be declared positionally, any missing parameters being indicated by commas, but this requires a knowledge of all the parameters possible for a particular macro and their order of specification. The macro descriptions given in this Guide usually only specify a few of the possible parameters, and should not be used as the basis of positional specification. In the JOB macro one could write JOBNAME=jobname but in that instance it seems simpler to leave the keyword out.) Keywords may come in any order.

Keywords may be contracted to the first three characters if desired, e.g. PROFILE= could be written PRO=. There are also contractions for the names of certain macros. These are indicated in the definition of those macros.

EXAMPLE OF A SIMPLE JOB CONTROL PROGRAM

The following example illustrates the job control required for a compile and run ALGOL job using the ALGOLE compiler. A file SORTDATA is set up and assigned to channel 6 in the ALGOL program. This writes to it, and the completed file is saved (made permanent) at the end of the job. All the SCL statements are in fact calls on standard macros.

```
JOB(:ECHRO6.SORTRUN)
NEW_FILE(NAME=SORTDATA)
ASSIGN_FILE(NAME=SORTDATA,LNAME=ICL9CE6,ACCESS=W)
ALGOLE
----

ALGOL Source program

++++
----

run-time data

++++
SAVE_FILE(NAME=SORTDATA)
ENDJOB
****
```

TESTING OF RESULT CODES

All macros, compiling systems and utility programs generate result codes (sometimes called return codes) indicating unusual conditions which arise in execution, for example the failure to assign a file required in a compilation, an attempt to access someone else's file for which permission has not been given, or the inability to perform the function requested for some reason.

The result code may reflect a catastrophic error, e.g. a failure to compile a source program, or something as minor as a comment that a default value has been assumed. Whatever the reason, whenever a result code is set its value is placed in an integer SCL variable defined for the purpose. In all macros, whether or not it is explicitly stated in the descriptions in this Guide, the user may nominate an SCL variable into which any result code is to be placed. If this is not specified the result code will be placed in an SCL variable RESULT. RESULT is defined in the outer SCL executed before the job is run; any user-nominated variable should be defined in the JCP before any reference to it, e.g. by a statement such as

```
INT FRED
```

where FRED is the name of the variable. Some macros will do this for the user, if a user nominated variable is specified in the macro call, but this should not be relied upon.

The value of the result code may be tested at any point in a JCP by either an IF statement or a WHENEVER statement. These are described below. The result code is set to zero at every macro or procedure call in the JCP, and stays that way unless an unusual condition is recognised.

The usefulness of result code testing is, unfortunately, seriously diminished by the absence of any rational scheme for the assignment of result codes. The present situation is chaotic: in theory a negative result indicates a warning and a positive result indicates a serious error. In practice this is not always complied with, and the problem is made worse by the lack of information concerning the result codes that different macros set. None the less, when the necessary information is available, result code testing can be very valuable. In particular, although a global test for a positive result code is unlikely to be useful such a test after an individual macro call may well be.

SETTING OF RESULT CODES IN USER PROGRAMS

Users of IMP, ALGOL (E) and FORTRAN (G) can set the value of an integer SCL variable ICL9CERESULT, defined in the code of the macro. In IMP and ALGOL the value is set by a procedure call

```
SETRESULT (integer-value)
```

In FORTRAN (G) the value is set by the statement

```
STOP integer-value
```

There is no similar facility in ICL F1 FORTRAN, though the STOP statement may be used to pass a text string into the SCL variable ICL9LFSTOPMESSAGE, which may then be tested.

The facilities available with COBOL are more complicated and are described in ICL Technical Publication 6834 - "Developing COBOL Programs".

THE GOTO STATEMENT

The statement

```
GOTO label
```

causes a jump to the appropriate statement in the JCP. GOTO is one word. Jumps backwards are only possible if they occur within a macro. SCL statements in a JCP are interpreted a statement at a time, and previous statements are no longer accessible.

THE IF STATEMENT

The statements

```
IF condition THEN action FI

IF condition THEN action1 ELSE action2 FI
```

may be used to test whether a particular condition has arisen and to specify the action to be taken. Note the need for the concluding FI.

The condition specified can compare two values of the same type, e.g. two integer values or two string values using the usual comparison operators EQ, NE, GT, LT, GE, LE or the equals sign.

If one of the values is defined in an expression that expression must be enclosed in brackets, such that there is only one operator not enclosed in brackets, e.g.

```
IF X=(Y+Z) THEN ....

IF (A OR B) OR C THEN ....
```

The action specified can be any unlabelled SCL statement, including another IF statement, e.g.

```
IF RESULT GT 0 THEN Z:=400 FI

IF ICL9CERESULT NE 0 THEN ENDJOB FI
```

If the condition specified is not detected execution of the JCP continues at the next statement.

THE WHENEVER STATEMENT

The WHENEVER statement enables a specified condition to be detected whenever it occurs in the execution of the SCL block in which the statement is defined. The condition which may be detected is more restricted than for an IF statement and may only consist of

```
variable-name   comparison-operator   expression
```

The form of the WHENEVER statement is

```
WHENEVER   condition   THEN   action   FI
```

The action specified may be any SCL statement as for IF. Note the need for the concluding FI.

D. JOB INITIATION AND TERMINATION

A job is initiated by calling the JOB macro, which specifies the user and job names, the environment in which the job is to be run, and any discs or tapes which must be mounted specially for that job. It also indicates what action is to be taken about the job if a system crash occurs during execution - is the job to be re-run identically or in a different way when the system has been reinstated, or is it not to be re-run at all?

The job will be terminated tidily when the ENDJOB macro is interpreted. Any temporary files will be deleted, any spooled files still in use will be marked for outputting, and the job journal (the log of actions executed in the course of the job) will be queued for printing. ENDJOB will normally be the last statement in the JCP, immediately before the four asterisks which mark the end of the JCP, but it can be issued at other points also, for example as the action to be taken if a particular test is satisfied.

Most jobs will be submitted to the system via a card reader, or at least by some device which appears like a card reader to the system. Jobs can however be stored in files on magnetic devices. They can then be initiated by calling the RUNJOB macro.

The JOB, ENDJOB and RUNJOB macros are described in detail below.

JOB

The JOB macro defines the start of the JCP and has the following form:

```
JOB (jobname,PROFILE=profile,TIME=ocptime,DISCS=discs,TAPES=tapes,RERUN=YES,
                        DELIVERY="text",DEVICE=device)
```

'jobname' has the form :username.name, where 'username' is the six-character user number and 'name' is up to 12 characters.

'profile' defines the environment in which the job is to be run. The default is RCOBATCH. It is not usually necessary to specify a profile for compilations and execution of compiled programs, but some packages require a profile. The occasions where a profile is needed are detailed under the appropriate facilities in this Guide or in package documentation.

'ocptime' sets a time limit in seconds for the entire job, after which the job will be terminated. Timing starts when execution of the work specified by the user begins; it does not include system time in setting up the job, which is however included in the ocp time charged to the job. The default is 30 seconds.

'discs' defines the discs needing to be mounted specially for the job, in the form name1 & name2 ..., where name1, etc., refer to six-character volume serial numbers.

'tapes' defines the tapes needed by the job, in the form name & name2 ..., where name1, etc., are six-character tape label numbers, followed by /W if a write permit ring is to be fitted; e.g. TAPES=AL6245/W & BZ6317.

'RERUN=YES' provides for the job to be re-run if the system crashes during execution. The default is RERUN=NO. NO results in files being tidied when the system is restarted and the journals being output to the user.

'text' is a string of up to 30 characters enclosed in double quotes (") defining delivery information for the job, and for any subsequent job unless redefined.

'device' is the name of the device to which the printed output of the job is to be sent. By default output will be sent to the printer associated with the device through which the job was input. For jobs input through the 2980 card reader, or transferred to the 2980 on tape, it is the 2980 printer; for jobs input from an RJE terminal it is the terminal printer; for jobs transmitted from EMAS it is the EMAS printer. A list of possible device names is given in Section B. This parameter applies by default to all printed output: it is possible to direct individual print files to specific printers by means of the LIST and LIST_FILE macros described in Section E. The parameter does not apply to printed output using the special mapping facilities available only on the 2980 printer, nor is any check made that the character set used in creating the print file(s) matches that available on the specified device.

ENDJOB

The ENDJOB macro causes a job to be terminated, any files in use to be closed and, if temporary, deleted; and in general completes the job in a tidy manner. It may be called at more than one place in a JCP to close down a job. The macro has no parameters.

RUNJOB

The RUNJOB macro runs a job already defined in a JCP held in a file or as a macro. The parameters are as for JOB with the addition that the name of the file holding the JCP or the name of the macro to be executed must be specified. The macro has the same form as the JOB macro, with the addition of one or other of the parameters FILENAME=name or MACRO=name, where 'name' specifies the name of the file holding the JCP or the name of the macro to be executed.

Note that the JCP in the file must not contain a JOB statement.

SEQUENCING OF JOBS

The macro RUNJOB does not result in the designated job being run as soon as the macro is interpreted; it merely places a request for the job to be run on the system's job queue. It is thus possible to run two jobs in order by placing the second job in a file (using INPUT, described in Section E) and issuing a RUNJOB statement for it as the last statement in the JCP for the first job. If a third job was to be run in sequence, this would also have to be placed in a file and queued by a RUNJOB statement at the end of the JCP for the second job, and so on.

E. FILE HANDLING

The system recognises four types of file:

* data files embedded in the job control program - known as 'alien data'

* permanent files

* named temporary files

* un-named temporary files - known as 'work files'

A permanent or named temporary file may exist independently or as a file in a library.

ALIEN DATA

An alien data file is a file input to the system through the input stream as card images. Common examples are the source program for a compilation, and data for the execution phase of a program.

Within a job an alien data file must be preceded by a line with four minus signs (only) as the first four characters, and concluded by a line with four plus signs (only) as the first four characters.

```
JOB (....)
.
.
.
SCL statements
.
.
.
----

alien data

++++
.
.
.
SCL statements
.
.
.
****
```

Alien data can only be input in this way if the job calls a macro which specifies such alien data input, as do most of the macros for compilation and for utility operations. Such information is given in the macro descriptions. The general way of entering data through the input stream is by means of the INPUT macro, described below. Within a job the alien data specified by an INPUT macro requires delimiters; outside a job it does not.

If necessary, the alien data delimiters may be changed by means of the CHANGE_DELIMITERS macro. This has the parameters START and FINISH, which define the two delimiters. Each delimiter consists of up to 4 characters, which must not conflict with the first characters on any SCL statement or data card following the macro call. The two delimiters must be different. An example of the macro call is

```
CHANGE_DELIMITERS (START=SSSS,FINISH=EFEF)
```

If a parameter is omitted the relevant default is taken (---- or ++++).

Alien data may not contain a line starting with four asterisks (only) or four slashes, nor may these be used as delimiters.

PERMANENT FILES

A permanent file may be set up by direct input to the filestore using the INPUT macro, or by explicitly saving a named temporary file in the creating job. Permanent files survive a system crash and exist separately from any jobs which may use them. If they are independent files they are catalogued; if they are held in a library only the library is catalogued.

NAMED TEMPORARY FILES

A named temporary file is created within a job and is automatically deleted at the end of the SCL block in which it was first mentioned. It does not survive a system crash. The file may exist independently or as a file in a library.

WORK FILES

Work files are created in a job either to hold data destined for printers or card punches, or to be used as un-named temporary ('scratch') files held on disc. In the first function they survive a system crash and are deleted when their contents have been output, which may be long after the job which created them has finished. As scratch files they do not survive a system crash, and are deleted at the end of the SCL block in which they were defined.

FILE ALLOCATIONS - THE ALLOC PARAMETER

Several of the macros described in this Section and elsewhere accept a parameter ALLOC. This specifies the manner in which space for the file is to be allocated on the physical medium. In most cases a suitable default exists and it is not necessary to specify the allocation required. However, the choice made, explicitly or by default, has an important effect on the default value of the amount of space allocated.

In most cases the default allocation type is L, in which case space is allocated in physical records (blocks) of 2048 bytes, usually three blocks to an allocation unit. Logical records span block boundaries if necessary. An L type file may be sequential or direct access, including an object program file, and may expand until there is no space left. Work files set up by the system use B type allocation by default and space is allocated in units of 228K bytes, the default size being 228K. Magnetic tapes use T type allocation and there are no defaults.

FILE DESCRIPTIONS - THE DESC PARAMETER

Several macros accept the parameter DESC, used to describe the type of file concerned. A description may be very complex, covering such matters as the type of device required to hold the file, blocksize, minimum and maximum record sizes, record type, etc. It is held in a file, either an independent file or associated with the file it describes, and the DESC parameter prints to the file containing the description. A file description may apply to a single file or may be of general applicability. The system provides facilities for a user to set up his own file descriptions, though this is beyond the scope of this Guide, and also provides a set of general descriptions. The system-provided descriptions will nearly always be adequate. In addition, for any macro which will accept a DESC parameter there is a default description, and in many instances this will be sufficient.

Standard file descriptions include *STDM, for a data file to be held on disc or tape; *STDOMF, for a file to contain an object program; *STDLP, *STDLIST and *STDFORT, to hold files to be listed to a printer (the appropriate description depending on the way vertical format control is to be effected); and *STDCP, for a file to be punched. Thus a description DESC=*STDFORT is used to specify that a file is to contain data produced by formatted FORTRAN output statements, and that the records in the file include the FORTRAN vertical format characters. In fact the real name of the file description is :STD.STDFORT, which has been defined (as with all the standard file descriptions) in the outer SCL which set up the environment in which the user job is run; the asterisk indicates that, in effect, the DESC parameter is referring to something already defined (i.e. a reference to STDFORT).

CONTROLLING FILES

A controlling file is an area of disc space into which user files can be placed. When a user creates a file on a demountable disc he must specify the controlling file to be used; in the case of permanent space the system selects an appropriate controlling file. Once created, a file can expand until all the space in the controlling file has been used up. Many users may share the same controlling file, and one of the problems in disc management is to try to ensure that space is being actively and efficiently used and that controlling files do not get full.

INPUTTING AND OUTPUTTING FILES

A file can be input to the filestore by using the INPUT macro. The macro can be used outside a job as well as inside and specifies the name of the file and, even if inside a job, the name of the owner (user). Outside a job the macro call is followed by the data, which is terminated with a line consisting of four asterisks (the JCP terminator). Within a job the data follows the macro call but must be bounded by the alien data delimiters (described above). A file so created has permanent status; if it is only required for the job in which it was input it must be explicitly deleted. A NEW_FILE statement (see below) must not be provided.

The macro has the form

```
INPUT (USER=username,NAME=filename,ACTION=action)
```

where ACTION specifies whether a file is to be created (C - the default) or whether the data following the macro call is to be added to an existing file (ACTION=A).

A file can be output to a printer or a punch using the LIST and LIST_FILE macros. LIST may be used outside a job and requires the name of the owner to be specified as well as the file name; LIST_FILE is used within a job and only needs the file name. Unless otherwise specified in the macro call the file will be printed. Both macros enable more than one copy of the listing or card deck to be produced.

The macros have the forms

```
LIST (USER=username,NAME=filename,COPIES=copies,DESC=description,DEVICE=device)

LIST_FILE (NAME=filename,COPIES=copies,DESC=description,DEVICE=device)
```

The default number of copies is 1. The default description for the file is for output to be printed, and that the file contains no format effectors (newline, etc.). This is equivalent to DESC=*STDLP, and the printer will move to a new line for each record printed. This is appropriate for output from IMP and ALGOL(E) streams, except where the program contains a call on the IMP routine SET MARGINS specifying a right hand margin of zero. In that case a description DESC=*STDLIST must be specified for the relevant stream. Otherwise an extra newline will be inserted for every line printed. Formatted output from FORTRAN programs (all compilers) contains special format effectors and the description DESC=*STDFORT must be supplied.

If the file is to be punched the description DESC=*STDCP must be supplied.

CREATING AND ASSIGNING WORK FILES

A file intended solely to be printed or punched may be created and assigned by calling the WORK_FILE macro. The file will be closed and queued for printing or punching at the end of the SCL block in which the WORK_FILE macro was called. The file will be deleted when all the output has been produced.

The macro call requires the local name of the file (see ASSIGN_FILE below) to be supplied; other parameters specify the number of copies required and whether output is to be printed or punched.

A scratch file on disc may also be created and assigned by calling the WORK_FILE macro. It will be deleted at the end of the SCL block in which the WORK_FILE macro was called.

The macro has the form:

```
WORK_FILE (LNAME=localname,DESC=description,COPIES=copies)
```

The default number of copies is 1. The default file description is DESC=*STDLP. This is appropriate for files to be printed which do not contain format effectors. This is appropriate for output from IMP and ALGOL(E) streams, except where the program contains a call on the IMP routine SET MARGINS specifying a right hand margin of zero. In that case a description DESC=*STDLIST must be specified for the relevant stream; formatted files produced by FORTRAN programs (all compilers) to be printed need DESC=*STDFORT. Card punch files need DESC=*STDCP. Scratch files on disc must be specified as DESC=*STDM.

The default maximum size for a work file is 228K bytes if DESC=*STDM. If that is inadequate a parameter MAXSIZE specifying the size in kilobytes may be supplied, e.g. MAXSIZE=450.

CREATING A NEW FILE

Any new file which is needed to take output from a program must be specified to the system. If the file is to be used only to hold output to be printed or punched it is specified and assigned by the WORK_FILE macro. Usually this will only be necessary if a card punch file or more than one print file is needed; the macros for calling compilers and previously compiled programs usually include provision for the first printer file.

It is convenient to use WORK_FILE for a disc file where the file is only to exist for the duration of the job. Otherwise NEW_FILE must be used. This registers a named temporary file but does not write into the file. If the file is to be a permanent one it must be explicitly saved before the end of the SCL block in which the NEW_FILE macro was called. NEW_FILE is used for both an independent file, in which case the name of the file is catalogued, or a file in a library, in which case the name is placed in the library index. Object programs can only be held in library files in a suitable library (see below under "Libraries").

The NEW_FILE macro has the form

```
NEW_FILE (NAME=filename,INITSIZE=size,ALLOC=allocation,CFILE=cfilename,VOLUME=label)
```

INITSIZE may only be specified for independent disc files. It specifies the amount of space in units of 1024 bytes to be allocated to the file. This space is reserved (and will be charged to the user) regardless of whether it contains data. The size should therefore be an accurate estimate, and the parameter should only be specified when a very large file (several megabytes in size) is being created, to ensure at the outset that there is enough space.

ALLOC is only required for files on magnetic tape, when ALLOC=T must be specified. A VOLUME parameter is also required.

The CFILE parameter is required only for files on demountable discs, when cfilename is the name of the controlling file into which the file is to be placed.

The VOLUME parameter is required only for files on tape, when 'label' specifies the tape label.

ASSIGNING A FILE TO A PROGRAM

Since files exist and are named independently of any program which may use them, each named file used by a program must be assigned to that program by using the ASSIGN_FILE macro. This provides the link between the file's catalogued name and the name or number by which it is referred to in a program - its 'local name'. This presents no problems with COBOL programs, but FORTRAN, ALGOL and IMP programs refer to files by using numbers, which are not acceptable SCL names. Thus each compiling system has to convert numerical file names to valid SCL names. In the case of FORTRAN (G), ALGOL (E) and IMP the number is prefixed ICL9CE; with ICL FORTRAN the prefix is ICL9LF. It makes no difference what type of file it is.

The parameters for the ASSIGN_FILE macro include the file name and the relevant local name, e.g.

```
ASSIGN_FILE (NAME=FRED,LNAME=ICL9CE27)
```

links a file called FRED to a program which refers to the file by using the number 27.

The analogy with job control using OS on the NUMAC machines is with the DD statement, though that does more than merely assign a file. The file's name is the DSNAME; the local name corresponds to the ddname; e.g. ICL9CE27 might correspond to FT27F001 or SQFILE27.

The local name is only valid for the SCL block in which the ASSIGN_FILE macro was called. The same file could be used with a different local name elsewhere in the same JCP.

Source Files

Source files for compilers are often input as alien data within the JCP. If they exist on disc or tape they must be assigned to the relevant compiler, this normally being done by the compile macro.

Control of Access

The ASSIGN_FILE macro provides for a parameter indicating whether the program wishes to access the file simply for reading, or for writing and reading. This protects files against accidental overwriting, the default access being for reading only.

Form of Macro

The ASSIGN_FILE macro has the following form:

```
ASSIGN_FILE (NAME=filename,LNAME=local name,ACCESS=access)
```

The local name is as required by the program using the file. ACCESS specifies the type of file access to be permitted. The default is R - read access only. Other possibilities are W (read and write), RE (read and execute), and WE (write and execute). RE and WE are only relevant with object program files.

Assigning Alien Data Files

In general, when a file is entered with the JCP it is necessary to provide a statement assigning that file to the program which is to use it. The exception is where the macro calling the program provides such an assignment as a default, as in the case with some compiler macros. Where it is necessary to assign the file explicitly the filename is given as *STDAD, e.g.

```
ASSIGN_FILE (NAME=*STDAD,LNAME=ICL9CE2)
```

MAKING A NAMED TEMPORARY FILE PERMANENT

A named temporary file may be made permanent, or "saved", by calling the SAVE_FILE macro before the end of the SCL block in which the NEW_FILE macro creating the file was issued. The macro has the form

```
SAVE_FILE (NAME=filename)
```

FILE DELETION

A permanent file may be deleted by calling either the DESTROY or DELETE_FILE macros. DESTROY takes immediate effect; DELETE_FILE does not take effect until the end of the SCL block in which it is called. In the past a modified DELETE_FILE macro has been in service on the 2980, having the same effect as DESTROY. The macros have the form

```
DESTROY (NAME=filename,GENERATIONS_KEPT=n)

DELETE_FILE (NAME=filename,GENERATIONS_KEPT=n)
```

DESTROY is a local macro (in fact it is DELETE_FILE between BEGIN and END statements); DELETE_FILE is a standard ICL one.

The parameter GENERATIONS_KEPT (which may be contracted to GEN) specifies the number, n, of generations of the file that are to be kept. By default, all will be kept except the most recent.

LIBRARIES

A number of files having the same file description may be held in a library. Since only the library itself is catalogued, access to files within the same library is more efficient and only the library itself counts in calculating the number of files a user possesses. Object programs must be held in library files.

An empty but permanent library is created by means of the macro NEW_LIBRARY; files within the library may then be created using NEW_FILE, saved (made permanent) using SAVE_FILE, and deleted using DESTROY or DELETE_FILE, which can also be used to delete the whole library.

If a library file is used by a program in a job the file must be assigned to that program by using the ASSIGN_FILE macro. Where several files from the same library are used in a single job, or wherever the same library is referred to several times in a single job, it improves efficiency if ASSIGN_LIBRARY is also used first to give the library a local name for that job, that local name being used in subsequent macros. ASSIGN_LIBRARY results in the library index being brought into store and kept there, thus avoiding subsequent macros which refer to the library (providing they use the local name) fetching the index from disc each time.

The macro NEW_LIBRARY has the form

```
NEW_LIBRARY(NAME=libname,DESC=description,INDEXSIZE=n,CFILE=cfilename)
```

The library name, libname, is a normal file name but must be followed by the qualifier M if the library is to hold object programs, e.g. MYLIB(M). If a generation number is included it must precede the qualifier, e.g. MYLIB(6,M) for the sixth generation.

The DESC parameter is only required for object program libraries, in which case DESC=*STDOMF must be specified.

The INDEXSIZE parameter specifies the maximum number of entries which may be held in the library index. The default is 144. In addition to an entry in the index for each file, there will be an entry for each separately defined entry point in an object program, e.g. for each external routine specified in an IMP program, or corresponding to each ENTRY statement in a FORTRAN program.

The CFILE parameter is only required for a library to be kept on a demountable disc, in which case cfilename is the name of the controlling file into which the library is to be placed.

The macro ASSIGN_LIBRARY has the form

```
ASSIGN_LIBRARY(NAME=libname,LNAME=local name,ACCESS=access)
```

The local name is an arbitrarily selected name by which the library is subsequently to be known. It is only valid within the SCL block in which the ASSIGN_LIBRARY macro was called. ACCESS is either R (read only - the default) or W (read or write).

OBTAINING A LIST OF A USER'S FILES

By calling the macro LIST_DIRECTORY, a list of all the permanent files belonging to the user who initiated the job in which the macro was called will be output in the journal (log) of that job.

The macro has no parameters. It provides a considerable amount of information concerning each file, much of which is self explanatory. The following notes may however be of assistance.

* "GEN" is the generation number of the file (see below).

* "VERS" is the version number and "LANG" is the language type. Both are beyond the scope of this Guide, except that LANG for an object file will be M.

* The node type (NT) will usually be R, signifying a "real" file, or L, signifying a library; other types of node being of concern to the system only.

* The file type (FT) defines the method of space allocation and is determined by the ALLOC specified, explicitly or by default, when the file was created.

* Block size is in bytes and will nearly always be 2048. Other sizes are in blocks, the relevant ones being "ALLOC SIZE" and "SECT SIZE". ALLOC SIZE is the number of blocks allocated to a file in a library, and may, because of the method of space allocation, be larger than the number of blocks used. The total size of a library is given at the foot of the listing of the contents of the library. SECT SIZE is the space used for an independent file or a file in a library; for the library itself it is the number of blocks used for housekeeping.

* The file placement will normally for user files be either the name of the "controlling file" in which the file is situated, or the tape serial number if the file is on tape. A controlling file is effectively a block of disc space set up by the system managers to hold files of a particular type for a specified group of users.

FILE GENERATIONS

A useful but potentially troublesome feature of VME/B is that it permits multiple "generations" of the same file. Both NEW_FILE and INPUT create files with specified names. If either macro specifies a file which already exists a second generation of that file is created. Thus

```
INPUT(USER=:ELMNO1,NAME=FRED)
    .
    file FRED
    .
INPUT (USER=:ELMNO1,NAME=FRED)
    .
    new file FRED
    .
    .
```

creates two files, FRED(1) and FRED(2). Any subsequent reference to file FRED is taken to refer to the most recent generation. If a particular generation is required one can either specify it explicitly, e.g. FRED(1), or relative to the most recent generation, e.g. FRED(-1). Thus if one wished to list FRED(1) one would need to write

```
LIST_FILE (NAME=FRED(1))
```

In the case of NEW_FILE the same applies, but that macro creates a temporary file which must be explicitly saved if it is desired to make it permanent. The SAVE_FILE macro, in the absence of an explicit definition of the generation, will save the most recent generation of the temporary file. Any other generation created in the job will be deleted at the end of the SCL block in which it was created.

The automatic creation of new generations of a file is a convenient facility, but is liable in a development environment to result in the inadvertent creation of a number of copies of a file, only the most recent copy being required. It is then a somewhat tedious process to delete all unwanted generations. Users are therefore recommended to review regularly the files they own in the filestore (by issuing the LIST_DIRECTORY macro) and to delete all unwanted ones.

ACCESS PERMISSION

A user may access his own files in any way he likes. Other users may only access one of his files if he has given them permission. Permission specifies the type of access permitted and may be given in general, to all users, or to specified users. The two relevant macros are CHANGE_FILE_ACCESS and CHANGE_FILE_RELATIONSHIP. These have the following forms:

```
CHANGE_FILE_ACCESS (NAME=hierarchic filename,ACCESS=access)

CHANGE_FILE_RELATIONSHIP (NAME=hierarchic filename,USER=username,ACCESS=access)
```

These macros may only be called by the controller of the file. The controller is normally the owner of the file, the person who created it, but it is possible to give control to other people (for details consult the Advisory Service). The first macro gives access to all users; the second gives access only to the user specified. The various access types are R (read only), W (read and write), E (execute - for program files only), F (any type of access, including the ability to delete the file), and X (switch off any access permissions previously granted). E may be combined with R or W, e.g. EW.

It is also possible with the CHANGE_FILE_RELATIONSHIP macro to prevent a specified user from accessing a file which has been made available to all users. This is done by specifying ACCESS=Z.

Any user accessing a file which is not his own must refer to the file by its full hierarchic name.

USE OF DEMOUNTABLE DISCS

Files on disc reside either on discs which are permanently mounted on drives, in which case the placement of the files is left to the system, or on demountable discs, which are mounted on request. Demountable discs enable very large files, or files which are regularly but relatively infrequently used, to be kept on disc without affecting the amount of space available for small and frequently used files, which are best kept on permanently mounted discs.

Users requiring space on demountable discs should contact their Advisory Service, who will tell them which discs to use, and the name of the controlling file into which the user's files are to be put. Whenever a demountable disc is used it must be specified in the JOB macro. The controlling file must be specified in the NEW_FILE, NEW_LIBRARY or NEW_DAFILE macro when a new file is created on a demountable disc.

DIRECT ACCESS FILES

A direct access file must be catalogued and formatted before it is used by a program for the first time. This is done by calling the macro NEW_DAFILE, the parameters for which specify the file name, record size and number of records. Thereafter, and in any subsequent run using the file (provided the file was a permanent one), the file is assigned to a program in the usual way with the ASSIGN_FILE macro. A direct access file may not be put in a library.

The macro has the form

```
NEW_DAFILE (NAME=filename,RECSIZE=length,NUMREC=no,CFILE=cfilename)
```

where 'length' is the length of the logical record, and 'no' is the number of records in the file. Since in using the file for the first time one may write anywhere in the file, this macro writes dummy records into the file, and the number of records specified determines the size of the file.

The CFILE parameter is only used for a file on a demountable disc, and specifies the name of the controlling file into which the file is to be put.

Note that, like NEW_FILE, NEW_DAFILE sets up a temporary file. To make it permanent it must be saved by calling SAVE_FILE before the end of the SCL block on which the NEW_DAFILE macro was called.

Record size cannot exceed 13024 bytes.

MAGNETIC TAPE FILES

Files on magnetic tape are handled in almost exactly the same way as disc files. Magnetic tapes must be labelled and catalogued before use, an operation which is carried out by operations staff when the tape is issued. Files on tape are catalogued in the same way as disc files. When they are created, using the NEW_FILE macro, ALLOC=T and a VOLUME parameter must be specified. The file so created is temporary and must be saved, if it is to be made permanent, by calling SAVE_FILE before the end of the SCL block in which the file was created. It is not necessary to specify the volume in subsequent references to the file, since the file has been catalogued and the system knows where to find it.

More than one file can be written to the same tape, and any file on the tape can be deleted without affecting the others. If a new file is written to a tape which already contains some files, the new file is written as a fresh section following the last file. A deleted file will only be overwritten by a new file if no undeleted file follows it.

The block size is normally fixed at 2036 bytes, which thus sets a limit to the maximum record size. Users needing a larger block size should contact their Advisory Services.

PROTECTION AGAINST LOSS OF A FILE

Although arrangements exist for routinely taking copies of the contents of the filestore in case of loss of data through system or hardware failure, such back-up copies of files will usually be about a week out of date. Further, in exceptional circumstances it might prove impossible to recover a file at all. Users are therefore recommended, where appropriate, to keep up to date copies of files on magnetic tape. Files may be written to tape by using FILE_COPY, described in Section I of this Guide.

The file on tape must have a different name from that of the version on disc, and will be catalogued when written to tape. The procedure for reinstatement of the file onto disc depends on the reason for reinstatement, and whether the disc and tape files are still catalogued. A system or hardware failure may result in either the catalogue or part of the filestore being destroyed or corrupted. If this affects users generally notification will be through Advisory Services and a notice on the output of all jobs run subsequently. Otherwise, affected users will be notified individually.

After such a failure each user must determine the extent to which his files have been affected. The macro LIST_DIRECTORY already described will provide a list of the state of a user's files, as known to the system catalogue. In the case of disc files this is a complete list of his files: later versions and other files may exist on disc but are not accessible. Tape files, however, still exist and can be recovered and LIST_DIRECTORY merely indicates those known to the catalogue. If a tape file appears in the listing it follows that the tape volume itself is also catalogued, but if there is any doubt the macro LIST_VOLUMES (which, like LIST_DIRECTORY, has no parameters) will provide a list of the volumes known to the catalogue. The macro CATALOGUE_MULTIFILE_TAPE, described in Section I, can be used to catalogue both a tape volume and any desired file on that tape. Thereafter a file may be copied from tape to disc by using FILE_COPY.

TRANSFER OF IBM FILES

There is a utility program available to enable users to transfer files from an IBM 370 machine. This is described in the section 'Utility Programs', elsewhere in this Guide.

LIMITS ON FILE SPACE AND NUMBER OF FILES

Users are normally limited to a maximum of 16 catalogued files on permanently mounted discs, with a total maximum size of 1.2M bytes. These limits are compatible with actual usage of file space on EMAS and at NUMAC and should be adequate for the majority of users. If a user needs higher limits he should approach his Advisory Service who will arrange for his limits to be increased. A library only counts as a single file, no matter how many library files it may contain. Users will be advised when they exceed the limits and must then arrange to delete files, or copy them to tape or demountable space, to get back within the limits. There are no limits for demountable space. The limits are intended to ensure that permanently mounted space is always available for users who need it, and assume that many users will never come near the limits. It is also necessary to ensure that permanently mounted space is being actively used: a user does not have the right to occupy, say, 1M byte of space with files which he never accesses. It is in any case very easy to end up with several generations of a file, only the last of which is really wanted.

Users will be advised if they have any files which have not been accessed for 28 days. If the files have still not been accessed 14 days later they will be deleted. It is not at present practicable to take archive copies of such files. This procedure only applies to permanently mounted space; it will eventually be necessary to have some limit for demountable space and users will be advised when that is introduced.

EXAMPLES OF FILE HANDLING IN A JCP

```
                                                                    Note
JOB (:RCAB21.FINAL_TEST)
NEW_FILE (NAME=TRIAL_DATA)                                           1
WORK_FILE (LNAME=ICL9CE20,DESC=*STDFORT)                             2
ASSIGN_FILE (NAME=TRIAL_DATA,LNAME=ICL9CE40,ACCESS=W)                3
ASSIGN_FILE (NAME=BASIC_DATA,LNAME=ICL9CE36)                         4
FORTRANG                                                             5
----

source program

++++
----

data                                                                 6

++++
ENDJOB
****                                                                 7
```

Notes: 1. Sets up an empty file called TRIAL_DATA with temporary status.

2. Sets up a file for printer output and assigns it to unit 20 in the program which will write to the file.

3. Assigns the file TRIAL_DATA to unit 40 in the program.

4. Assigns a permanent existing file BASIC_DATA to unit 36 in the program.

5. Calls the FORTRAN (G) compiler. This will compile the source program (which in this example follows immediately as alien data) and run it, taking data for unit 5 from the next alien data file.

6. The alien data file for the run phase of the job.

7. End of JCP. At this point the work file produced on unit 20 will be queued for printing, also any printed output produced on unit 6, the standard printed output unit (defined within the macro FORTRANG). The file TRIAL_DATA will be deleted.

```
                                                                    Note
JOB (:CLMR07.FINALRUN)
NEW_DAFILE (NAME=INDEX,RECSIZE=100,NUMREC=400)                       1
ASSIGN_FILE(NAME=INDEX,LNAME=ICL9CE23,ACCESS=W)                      2
ASSIGN_FILE(NAME=TRIAL6,LNAME=ICL9CE12)                              3
WORK_FILE(LNAME=ICL9CE8,DESC=*STDCP)                                 4
FORTRANG(INPUT=STATPROG)                                             5
SAVE_FILE(NAME=INDEX)                                                6
ENDJOB
****
```

Notes: 1. Sets up a 400 100-byte record direct access file

2. Assigns the file to unit 23 with write access

3. Assigns an existing file TRIAL6 to unit 12 for read access only

4. Sets up a file for card punch output and assigns it to unit 8

5. Compiles and runs a program, held in the filestore as STATPROG, using the FORTRAN compiler (see Section F). No data is included in the job control program.

6. Saves (makes permanent) the direct access file - providing the program run did not fail: if it had the SAVE_FILE call would never have been reached.

7. At this time the card punch file will be queued for punching and subsequent deletion.

```
                                                                    Note
JOB(:ERNL03.UPDATE6,TIME=40,TAPES=DC1642/W)                          1
INPUT(NAME=DAIRYCOW,USER=:ERNL03,ACTION=A)                           2
----

data

++++
LIST_FILE(NAME=DAIRYCOW)                                             3
NEW_FILE(NAME=DCOWSTAT,ALLOC=T,VOLUME=DC1642)                        4
ASSIGN_FILE(NAME=DCOWSTAT,LNAME=ICL9CE10,ACCESS=W)
ASSIGN_FILE(NAME=DAIRYCOW,LNAME=ICL9CE14)                            5
WORK_FILE(LNAME=ICL9CE8,DESC=*STDM)                                  6
FORTRANG                                                             7
----

source program

++++
SAVE_FILE(NAME=DCOWSTAT)                                             8
LIST_DIRECTORY                                                       9
ENDJOB
****
```

Notes: 1. Defines an OCP time limit of 40 seconds and specifies that the job requires a tape DC1642 to be mounted with a write permit ring.

2. Adds the data which follows to an existing file DAIRYCOW. Since this is done within a job, alien data delimiters are needed. The example shows the standard delimiters.

3. Lists the complete file DAIRYCOW to the printer.

4. Defines a new file DCOWSTAT to be held on tape. The next statement assigns the file to unit 10 in the FORTRAN (G) program which is to be run. Since it is a new file one must be intending to write to it so ACCESS=W is specified.

5. Assigns the file DAIRYCOW to unit 14 in the program - evidently the program is only going to read the file since no ACCESS is specified.

6. Defines a scratch file on disc to be used by unit 8 in the program. It is assumed that one can both read and write a scratch file on disc.

7. Calls the FORTRAN (G) compiler to compile and run the source program which follows. The program evidently does not require any alien data to be supplied for execution.

8. Saves (makes permanent) the file DCOWSTAT if the job gets as far as this statement. Although the file is on tape it would remain temporary without this statement and could be overwritten by a later job.

9. Lists the files belonging to user ERNL03. This should show, inter alia, that DCOWSTAT has been saved and that DAIRYCOW is larger than on a previous occasion, though it cannot show that the increase in size has just occurred. DAIRYCOW was already a permanent file.

```
JOB(:ERNL03.UPDATE7,TIME=40)
ASSIGN_LIBRARY (NAME=AGLIB,LNAME=LIB,ACCESS=W)
INPUT (NAME=*LIB.DAIRYCOW,USER=:ERNL03,ACTION=A)
----

data

++++
LIST_FILE (NAME=*LIB.DAIRYCOW)
NEW_FILE (NAME=*LIB.DCOWSTAT)
ASSIGN_FILE (NAME=*LIB.DCOWSTAT,LNAME=ICL9CE10,ACCESS=W)
ASSIGN_FILE (NAME=*LIB.DAIRYCOW,LNAME=ICL9CE14)
WORK_FILE (LNAME=ICL9CE8,DESC=*STDM)
FORTRANG
----

source program

++++
SAVE_FILE (NAME=*LIB.DCOWSTAT)
LIST_DIRECTORY
ENDJOB
****
```

The above example is the same as the preceding one, except that

1. The existing file DAIRYCOW, to which the input data is appended, is held in a library AGLIB, which must also already exist.

2. The file DCOWSTAT created by the job is put into the library AGLIB.

The ASSIGN_LIBRARY call enables the library to be referred to subsequently by the local name LIB and avoids the library index being read in from disc for each macro which referred to the library. If this were not done all references to *LIB. etc. would have to be to AGLIB. etc., which would be much less efficient.

F. SCIENTIFIC JOBBER

INTRODUCTION

The Scientific Jobber is a batch subsystem which supports an easy-to-use low-overhead development environment for scientific programs. It provides for rapid compilation of batches of ALGOL, IMP and FORTRAN programs by minimising the system overheads associated with job initiation, compilation and program loading. This is achieved by defining a set of facilities less than that provided by the full operating system, and supporting this set within a single program (as far as the operating system is concerned). The Scientific Jobber incorporates a job control language processor, the FORTRAN (G), IMP and ALGOL (E) compilers, a batch program loader, standard input/output routines, mathematical procedures and a basic set of file utilities, including a source program amender (editor).

Jobs for the Jobber are submitted with a Job Request Card overprinted with the word JOBBER. A group of such jobs are then batched together by operations staff and enclosed in the SCL statements necessary to call the Jobber. For each job within the batch the Scientific Jobber processes the control statements, performs any utility actions requested, enters the appropriate compiler and, if compilation is successful and the user requests it, enters the compiled program.

The version of FORTRAN is compatible with IBM level G, as available on EMAS and on IBM machines. It is not compatible with ICL New Range FORTRAN. The ALGOL compiler is similar to that on EMAS, and is an ALGOL 60 implementation with 1900-like input/output statements. IMP is a slightly modified version of the languages available on EMAS and at NUMAC. The differences are listed in Appendix 2.

FORTRAN may handle either 2900 EBCDIC or ISO character codes and the word FORTRAN in this section applies equally to both versions of the compiler. ALGOL and IMP can only handle ISO code.

The compilers are also available directly under VME/B, i.e. outwith the Scientific Jobber, as described in Section G.

FACILITY LEVELS

The Scientific Jobber provides on the one hand for very simple student jobs, requiring only cards-in lines-out facilities, and on the other for the development of substantial programs requiring the use of data files, which, when developed, will be transferred to the standard production environment, where the full set of program and data management facilities are available. A user does not have to be registered on the system unless he uses a file.

RESOURCE CONTROL

Each user job processed by the Jobber is subject to a limitation on the ocp time it may use and the number of lines of printed output it may produce. These limits may be set by the user, subject to maximum values defined by the installation. The limits are specified below.

LIBRARY EXTENSION

The basic set of library procedures which a compiled program may call, for example mathematical functions, are pre-loaded with the Scientific Jobber. This basic set may be extended by an installation when the Jobber is being established on the operating system.

CONTROL STATEMENTS

The commands to the Jobber, while in some instances similar in name and effect to commands at the system command level, are of necessity different in detail and method of implementation. To emphasize this distinction Jobber commands are preceded by a double slash. The double slash also provides an implicit end of file marker for source programs and data included with the jobs. This removes the requirement for alien data delimiters and ensures that users' jobs are not accidently processed as alien data by a previous job.

Each control statement will take the form

```
// command name
```

or

```
// command name (parameter list)
```

where 'command name' may be selected from a prescribed list and 'parameter list' may contain positional or keyword parameters. // must appear as the first two characters of the statement, but otherwise spaces are ignored.

JOB DELIMITERS

Each job within the batch should start with a statement of the form

```
// JOB (parameter list)
```

and be terminated by

```
// ENDJOB
```

where 'par meter list' may contain the following:

JOBNAME = user/job identification, in one of the following forms:

```
:username.jobname
:username
```

where 'username' is the user's number (six characters), and 'jobname' is a string of up to 12 alphanumeric characters starting with a letter.

DELIVERY = text giving user's name or delivery information (included in banner at top and tail of job output). This text may not include a comma.

TIME = ocp time limit (in seconds) for the job. The default is 30 seconds; the maximum is 300 seconds.

LINES = printed output limit (in lines) for the job. The default is 1000 lines; the maximum is 5000 lines.

All parameters after the first are optional.

For example

```
// JOB (:CHEM01.TEST1,J.SMITH_ROOM_2001,L=2000)
```

introduces a job with the default ocp time limit and a 2000 line output limit. This illustrates the use of a mixture of positional and keyword parameter specification.

COMPILER AND RUN-TIME OPTIONS

The degree of checking performed while user programs are being executed, the amount of diagnostic information available following a run-time error, and several other options may be specified by a statement of the form:

```
// OPTIONS (option list)
```

'option list' contains one or more of the following, separated by commas.

LIST	generate full source code listing
NOLIST	generate minimal source code listing (generally first and last statements of each routine and erroneous statements)
CHECK	include code in the object program to check for use of variables which have not been assigned a value
NOCHECK	omit checks for unassigned variables
ARRAY	check that array subscripts lie within the defined bounds of the array being processed
NOARRAY	omit array bound checking

LINE	include code to maintain a record of the line number of the statement currently being executed to assist with diagnosis of errors
NOLINE	omit line number updating
DIAG	retain symbol tables for listing values of variables during a routine trace back after an error has occurred
NODIAG	omit the symbol tables
OPT	has the same effect as NOCHECK,NOLINE,NODIAG
INHIBIOF	inhibit the occurrence of an integer overflow interrupt

Applying to FORTRAN only:

LABELS	include code to maintain, during execution of a program, a list of the most recent labels, subroutine calls and returns. Print this if a run-time error occurs.
NOLABELS	no label trace required
MAP	at the end of each program unit print a list of identifiers and their attributes
NOMAP	no identifier list required
EBCDIC	use ICL 2900 EBCDIC code for the representation of internal character data. See below under "Compilation".
ISO	use ISO code (compatible with ALGOL (E) and IMP) for the representation of internal character data. See below under "Compilation".

Applying to ALGOL only:

QUOTES	keywords are enclosed in quotes, e.g. 'BEGIN'
PERCENT	keywords are preceded by the % symbol, e.g. %BEGIN

Options bracketed together are mutually exclusive. If both are included in an option list the effect is undefined. The standard defaults are underlined.

The effect of an OPTIONS statement is to modify the default set of options for subsequent compilations, for the remainder of the current job or until another OPTIONS statement is encountered within the same job. Each new job resets the installation default values.

COMPILATION

An ALGOL program provided on cards may be compiled by including

```
// ALGOLE
<program text>
```

and similarly an IMP program provided on cards may be compiled by including

```
// IMP
<program text>
```

If the program source is held in a file then the statements are

```
// ALGOLE (source file name)
```

and

```
// IMP (source file name)
```

respectively. For FORTRAN the relevant statements are

```
// FORTRANG or FORTE
<program text>
```

and

```
// FORTRANG (source file name)  or  // FORTE (source file name)
```

The alternatives FORTRANG and FORTE give the same compiler but set different default options for the character set representation. FORTRANG sets the EBCDIC option; FORTE the ISO option, compatible with the FORTE compiler on EMAS and at NUMAC. FORTRAN is an acceptable alternative to FORTRANG.

PROGRAM EXECUTION

The statement

```
// RUN
```

will load and enter the last program compiled in the current job, provided that compilation was successful. Unless otherwise defined the default input channel may be used to access data immediately following this statement and the default output channel writes to the printer file being generated for the job. The default input and output channels are 98 and 99 for ALGOL and IMP and 5 and 6 for FORTRAN. ALGOL will also accept 2 as a default input channel and 1 as a default output channel. Note that if no explicit select is performed in an ALGOL or IMP program then input data will be assumed to follow the RUN statement.

A program may be run several times with different sets of data following a single compilation.

The following example of a complete user job illustrates the use of the commands above:

```
// JOB (:PHYSO2,M.JONES)
// OPTIONS (MAP)
// FORTRANG
   <program text>
// RUN
   <test data 1>
// RUN
   <test data 2>
// ENDJOB
```

DATA FILE DEFINITION

The relationship between a logical channel number and an actual data file is defined by a statement of the form

```
// DEFINE FILE (parameter list)
```

or the equivalent short form

```
// DFF (parameter list)
```

'parameter list' may be made up of positional parameters, in the order of description below, separated by commas, e.g.

```
(1,STATS,,OLD)
```

or made up of keyword parameters, e.g.

```
(CHANNEL=1,NAME=*)
```

It may contain the following:

CHANNEL = integer defining a logical channel in the range 1-98. This definition replaces any previous definition for the specified channel. Note that channel 99 may <u>not</u> be redefined.

NAME = name of permanent file associated with the logical channel, <u>or</u> the single character *, defining the file as comprising the in-stream data immediately following this statement, <u>or</u> an integer specifying another channel to which the specified channel is equivalent. If this parameter is not specified a temporary file is assumed. A channel referred to by number must be defined prior to use unless a default (e.g. 99) is referred to. If a file name is specified it must follow the naming rules given in Section C.

ACCESS = R read access only is required.
W read and write access is required. This is the default value unless NAME=* is specified, when R is the only access allowed.

STATUS = OLD - if specified the job will be abandoned unless the file already exists.
NEW - if specified the job will be abandoned if the file already exists.

If this parameter is not specified a new permanent file is created, unless a file with the specified NAME already exists.

TYPE = DA specifies a direct access file.
SQ specifies a sequential access file (the default if the file's status is NEW).

The parameter is only required when a new direct access file has to be created.

RECSIZE = maximum record size in bytes. This is only necessary for new sequential files where the installation default is not suitable, and for new direct access files.

FILESIZE = an integer specifying a maximum filesize in units of 1024 bytes, or nR, where n is the number of records in a direct access file. This is only necessary for new sequential files where the installation default is not suitable, and for new direct access files.

All files required by a program, with the exception of the default files where these are suitable, must be defined prior to a RUN of the program. The definition for a channel remains valid until that channel is re-defined or the end of the job is reached.

The following example shows a program requiring two sets of input data, on channels 1 and 5, and writing to a file on channel 2.

```
// JOB (:STAT03,J.KENNY)
// FORTRANG
   <program text>
// DEFINE FILE (1,*)                          or  // DFF (1,*)
   <data for channel 1>
// DEFINE FILE (2,STATSDATA)                  or  // DFF (2,STATSDATA)
// RUN
   <data for default channel 5>
// ENDJOB
```

The same example, using keyword instead of positional parameters, could be as follows:

```
// JOB (JOBNAME=:STAT03,DELIVERY=J.KENNY)
   FORTRANG
   <program text>
// DEFINE FILE (NAME=*,CHANNEL=1)             or // DFF (NAME=*,CHANNEL=1)
   <data for channel 1>
// DEFINE FILE (CHANNEL=2,NAME=STATSDATA)     or // DFF(CHANNEL=2,NAME=STATSDATA)
// RUN
   <data for default channel 5>
// ENDJOB
```

As implied by the DEFINE FILE statements above, the order of parameters is unimportant when keywords are used. There is also less risk of error.

All files created by the Jobber are standard VME/B files and may be accessed outwith the Jobber.

INPUTTING NEW FILES

The statement

```
// INPUT FILE (filename)
```

or the short form

```
// INF (filename)
```

checks that a file with the nominated filename does not already exist, and, if it does not, creates a permanent file containing the data which immediately follows the statement. If the file already exists the job is abandoned.

DELETING FILES

The statement

```
// DELETE FILE (filename)
```

or the short form

```
// XF (filename)
```

deletes the nominated file with immediate effect. If the file does not exist this is reported and the job continues.

LISTING SOURCE FILES

The statement

```
// LIST FILE (filename)
```

or the short form

```
// LF (filename)
```

lists the contents of the specified file in the printer file for the job. This listing is subject to the printed output limit imposed on the job. If the file does not exist this is reported and the job continues.

OBTAINING A LIST OF A USER'S FILES

The statement

```
// LD
```

produces a list of the files owned by the user issuing the statement.

SOURCE PROGRAM AMENDER

The Scientific Jobber control statement required to invoke the source program amender is

```
// AMEND (input file, output file)
```

'input file' = name of an existing file which is to be amended (optional)

'output file' = name of a file, which may or may not exist, which is to receive the amended file.

Parameters may be positional, or keyword, i.e. INPUT=, OUTPUT=.

The data controlling the amendments is assumed to follow the AMEND statement immediately.

The facilities available include: creating a card image file and adding sequence numbers to it in any specified set of card columns; resequencing a complete file; deleting cards; replacing and inserting cards, and sequencing inserted cards where necessary. In addition, the output file can be listed.

The control file is made up of a number of data cards and control statements. Any card not conforming to the control statement format described below is treated as a data card.

Amender data cards

Data cards normally have sequence numbers, in the appropriate set of columns (either 73 to 80, or as specified via **SEQ - see below). When sequence numbers are given, leading zeros (filling up the sequence field) should also be included.

Data cards must appear in the correct order with respect to the rest of the control file. Each card either replaces a card in the file being amended if their sequence numbers match, or is inserted into the file if there is no matching card in the input file.

Amender control statements

Each control statement is on a separate card and is of the following form:

```
**control  parm1,parm2
```

The asterisks appear in columns 1 and 2 and they are immediately followed by one of a set of control words, detailed below. Up to two parameters, depending on the control word, can also be specified on a control card and if present they should be separated from the control word by at least one space and at most twelve spaces. If two parameters are specified, the separating comma must immediately follow the first parameter. The amendment process is sequential, and the order of control statements and data cards in the control data set must reflect this. Cards are referenced by their sequence numbers in the input file, not by the (possibly different) numbers they will have in the output file.

The following conditions are initially assumed to hold:

1) the input file is to be copied card by card to the output file
2) the input file is sequenced in card columns 73-80
3) neither the input file nor the output file is to be listed

The control statements are:

**SEQ n1,n2 — The input file is sequenced in columns n1 to n2 inclusive: n1 and n2 are integers in the range 1 to 80. The output file is also to be sequenced in columns n1 to n2. If a sequence field of less than six characters is specified then the increment between card numbers is 10; otherwise it is 100.

**FILE L,N — There is no input file; instead the output file is to be created from the data cards immediately following. Both parameters are optional and may appear in either order.

L - list the new file as it is created
N - do not add sequence numbers while creating the file

**ALTER R,L — If R is coded then the whole file is to be resequenced when copied to the output file. If L is coded then the output file is listed. However, cards deleted from the file are not listed (they are if L is not coded). The parameters may appear in either order.

**DELETE n1,n2 — If n1 alone is coded then the card in the input file with that number is not transferred to the output file, i.e. it is deleted. If n2 is also coded then the set of cards from n1 to n2 in the input file is deleted.

Leading zeros do not have to be coded. Cards deleted as a result of this statement are listed on the line printer followed by the letter D, unless **ALTER L has been previously coded.

**INSERT n1,n2 — This card should be followed immediately by a number of unsequenced data cards. They are inserted in the output file starting at n1, with increment n2. n2 is optional with a default value of 100.

**LISTON n1 — The input file is copied unchanged to the output file until a card with sequence number greater than or equal to n1 is encountered. Cards written to the output file thereafter are listed. If n1 is omitted, the listing of the output file cards starts immediately.

**LISTOFF n1 — The input file is copied unchanged to the output file until a card with sequence number greater than or equal to n1 is encountered. Cards written to the output file thereafter are not listed. If n1 is omitted, the listing of the output file cards stops immediately.

**END — Exits from the Amender. The Amender is also terminated tidily by the next Jobber control statement.

Amender error messages

The following messages may appear:

INVALID CONTROL CARD	A card with a valid control word is faulty. The card is ignored.
INVALID SEQ NUMBER	A sequenced data card in the control data set has a non-numeric character in the sequence field. The data card is ignored.
CARD OUT OF SEQUENCE	A card in the control data set has referenced a point in the input file which has already been passed. The card is ignored.
ATTEMPT TO INSERT TOO MANY CARDS	A sequence number assigned to an unsequenced data card by a **INSERT statement is greater than or equal to the sequence number of the next card in the input file. The rest of the batch of unsequenced data cards is ignored.
SEQUENCING OUT OF FIELD	A sequence number being assigned to a card in the output file is too large to fit into the sequence field given. The Amender is terminated, leaving the output file incomplete.
OUTPUT FILE CAPACITY EXCEEDED	An attempt has been made to write more than the defined maximum capacity to the output file. The Amender is terminated.
FILE DOES NOT EXIST	No input file has been specified. The Amender is terminated.

FORTRAN RUN-TIME ERRORS

If a run-time fault occurs in a FORTRAN program, the diagnostic package is entered. The function of the diagnostic package is as follows:

* To report the nature of the fault; for example

```
*** PROGRAM ERROR 11    UNASSIGNED VARIABLE ***
```

* To identify and report the point in the main program or subprogram at which the failure occurred; for example

```
MONITOR ENTERED FROM SUBROUTINE SUB1 LINE 36
```

* To print out the values at the time of failure of all the local and COMMON scalar variables in the main program or subprogram. If no value has been assigned to a variable, this is indicated. For example

```
LY = NOT ASSIGNED
CZ = (1.0000,1.0000)
F = 1.234
K = 1
P = 6

LABELLED COMMON L2
U = 1.000
V = 2.000
W = 16.234
```

* Unless the failure occurred in the main program, to identify the main program or subprogram from which the current subprogram was called and repeat steps 2, 3 and 4, until diagnostics for the main program have been given.

The user may invoke the diagnostic package, without terminating his program, by calling the subroutine ICL9CEDIAG.

If the option LABELS was selected at compile time, FORTRAN provides further diagnostic facilities, as follows:

* A list of the last 32 labels, subprogram entries and exits may be produced by a call on the subroutine IC 9CELABELS; for example

```
CALL ICL9CELABELS
```

in a FORTRAN program might result in the f llowing output:

```
*****  LABEL TRACE  *****
ENTER MAIN PROGRAM
ENTER FN./SUBR.    TEST85
LABEL      10      TEST85
LABEL      20      TEST85  (50)
LABEL       2      TEST85
ENTER FN/SUBR.     TEST86
LABEL       5      TEST86
```

In the example above the number in brackets, (50), indicates that label 20 in subprogram TEST85 was passed 50 times in succession, e.g. in a DO loop.

* Subroutine ICL9CEFTRACE (n) may be called to switch on and off the listing of the labels and subroutine entries and exits during execution of a program:

```
n = 0  turns the tracing off
n = 1  traces subprogram entry and exit only
n = 2  traces subprogram entry and exit and all labels
```

ALGOL RUN-TIME ERRORS

Various faults can occur during the execution of an ALGOL program and cause execution to cease. If the program was compiled in diagnostic mode, the diagnostic package is entered after the fault has been reported. The function of the diagnostic package is as follows:

* To identify the logical block or procedure in which the failure occurred; for example:

```
PROCEDURE FRED STARTING AT STMNT 31
```

or

```
BLOCK STARTING AT STMNT 6
```

* To print out the values at the time of failure of all the local scalar variables of the logical block or procedure. If no value has been assigned to a variable, this is indicated. Arrays and parameters passed by name are not printed, but parameters passed by value are treated as local scalars. For example:

```
LOCAL SCALAR VARIABLES
I = 1
J = NOT ASSIGNED
X = 4.77777 @-5
```

Note that the names of scalar variables are truncated to eight characters (if necessary) for diagnostic purposes only.

* To indicate the statement from which the logical block or procedure was entered.

* To repeat the above three steps for the logical block from which the current logical block was entered, and to repeat this process until the first begin of the program has been reached. Thus procedures which have been used recursively will have the current values of the variables declared in each activation printed out.

The user can invoke the diagnostic package without terminating execution of his program by means of the pseudo procedure statement MONITOR. For example:

```
if PERCENT>100 then MONITOR
```

Various faults can occur during the execution of an IMP program and cause execution to cease. If the program was compiled in diagnostic mode, the diagnostic package is entered after the fault has been reported. The function of the diagnostic package is as follows:

* To identify the logical block or procedure in which the failure occurred; for example:

```
      ROUTINE/FN/MAP FRED STARTING AT LINE 31
```

  or

```
      BLOCK STARTING AT LINE 6
```

* To print out the values at the time of failure of all the local scalar variables of the logical block or procedure. If no value has been assigned to a variable, this is indicated. Arrays and parameters passed by name are not printed, but parameters passed by value are treated as local scalars. For example:

```
      LOCAL SCALAR VARIABLES
      I = 1
      J = NOT ASSIGNED
      X = 4.77777 @-5
```

  Note that the names of scalar variables are truncated to eight characters (if necessary) for diagnostic purposes only.

* To indicate the statement from which the logical block or procedure was entered.

* To repeat the above three steps for the logical block from which the current logical block was entered, and to repeat this process until the first begin of the program has been reached. Thus procedures which have been used recursively will have the current values of the variables declared in each activation printed out.

The user can invoke the diagnostic package without terminating execution of his program by means of the unconditional instruction monitor. For example:

```
      if PERCENT>100 then monitor
```

RUN-TIME AND CONTROL ERROR MESSAGES

Program Run-Time Errors (All Compilers)

1	REAL OVERFLOW	On evaluation of a real expression a number has been generated which is too large to be held in the accumulator.
2	REAL UNDERFLOW	This should not occur.
3	INTEGER OVERFLOW	On evaluation of an integer expression a number has been generated which is too large to be held as an integer.
4	DECIMAL OVERFLOW	This should not occur.
5	ZERO DIVIDE	A division by zero has been attempted.
6	ARRAY BOUNDS EXCEEDED	An array subscript has been found outside the declared bounds.
7	CAPACITY EXCEEDED	An attempt has been made to assign a value to a location too small to receive it. This is most likely to occur if formal and actual parameters have different lengths.
8	ILLEGAL OPERATION	This should not occur.
9	ADDRESS ERROR	Various types of addressing fault can cause this error. The commonest are trying to write to a protected area or trying to access unallocated virtual storage. Possible reasons for this happening are: a) An access outside the bounds of an array when the bound checking option has been turned off. b) Actual and formal parameters are incompatible, particularly in mixed language working.
10	INTERRUPT OF CLASS n	A hardware detected program error of class n (and not specifically identified above) has occurred. This should not normally occur.

11	UNASSIGNED VARIABLE	An attempt has been made to use a variable to which no value has yet been assigned.
15	ILLEGAL EXPONENT	An exponent greater than 255 (or 63 in an integer context) has been found.
16	SWITCH LABEL NOT SET	An instruction of the form ->SW(<exprn>) has been executed and the required label cannot be found.
18	ILLEGAL CYCLE	A cycle instruction has been executed where it is impossible to reach the final value of the control variable.
19	INT PT TOO LARGE	An attempt to convert a real value to an integer value has failed because the integer part of the real number is greater than the largest allowed integer. This may, in some circumstances, be reported as an overflow fault.
21	NO RESULT	An attempt has been made to exit from a fn or map via an end instruction. Control must be returned via result.
22	PARAM NOT DESTINATION	This fault may occur when trying to assign to a parameter passed by name in an ALGOL program. The actual parameter supplied is not a variable of the correct type and thus the assignment cannot be completed.
24	STREAM NOT DEFINED	An ALGOL or IMP program has attempted to call SELECT INPUT or SELECT OUTPUT on a stream which has not been defined.
25	INPUT FILE ENDED	An ALGOL or IMP program has attempted to read past the end of an input file.
26	SYMBOL IN DATA	When executing an ALGOL or IMP read instruction a non-numeric symbol has been found when not expected.
28	SUB CHARACTER IN DATA	This fault occurs in an ALGOL or IMP program on attempting to read the first character of a line containing a substitute character (indicating an invalid code or corrupted character).
29	STREAM IN USE	An ALGOL or IMP program has attempted to call SELECT INPUT or SELECT OUTPUT on a stream which has previously been selected for the other use, i.e. output or input respectively.
32	RESOLUTION FAULT	An unconditional resolution cannot be completed as the string to be resolved does not contain the symbols being searched for.
33	INVALID MARGINS	The margin settings specified via SET MARGINS are not valid.
34	SYMBOL INSTEAD OF STRING	On executing a READSTRING instruction a symbol other than a string delimiter was read first.
35	STRING INSIDE OUT	In a call of FROMSTRING the first character position specified comes after the last character position specified.
36	WRONG PARAMS PROVIDED	The parameters to an externalroutine were not consistent with its definition. Usually an externalroutinespec has been copied incorrectly.
37	UNSATISFIED REFERENCE	An attempt has been made to call an external procedure which was not available when the program was loaded.

Mathematical Library Errors

50	SQRT ARG NEGATIVE	The argument passed to the SQRT function is negative.
51	LOG ARG NEGATIVE	The argument passed to the LOG or ALOG function is negative.
52	LOG ARG ZERO	The argument passed to the LOG or ALOG function is zero.
53	EXP ARG TOO LARGE	The argument passed to the EXP routine is so large (greater than 172.694) that the results are no longer guaranteed.
54	SIN ARG TOO LARGE	The argument passed to the SIN function is so large that the results are no longer guaranteed.
55	COS ARG TOO LARGE	The argument passed to the COS function is so large that the results are no longer guaranteed.

56	TAN ARG TOO LARGE	The argument passed to the TAN function is so near an odd multiple of $\pi/2$ that n overflow condition would exist.
57	ARSIN ARG OUT OF RANGE	The absolute value of the argument passed to ARSIN is greater than 1.
58	ARCOS ARG OUT OF RANGE	The absolute value of the argument passed to ARCOS is greater than 1.
59	ATAN2 ARGS ZERO	The two arguments passed to ATAN2 are both zero.
60	SINH ARG OUT OF RANGE	The absolute value of the argument passed to SINH is greater than 172.694.
61	COSH ARG OUT OF RANGE	The absolute value of the argument passed to COSH is greater than 172.694.
62	RADIUS ARGS TOO LARGE	The parameters passed to RADIUS are too large: the value of X**2 or of X**2 + Y**2 is greater than the largest real number allowed.
63	TRIG FN INACCURATE	The argument of a SIN, COS or TAN function is so large (>1@7) that the results are no longer guaranteed.
65	GAMMA FN ARGS OUT OF RANGE	
66	LOG GAMMA ARGS OUT OF RANGE	

FORTRAN Format Errors

101	MISSING LEFT BRACKET	The initial left bracket is missing.
102	MISSING RIGHT BRACKET	The terminating right bracket is missing.
103	NEGATIVE SIGN INCORRECT	Negative sign is only allowed with scaling i.e. P-format.
104	INVALID FORMAT	The format construction is not in a valid form.
105	DECIMAL FIELD > WIDTH	The total width field in a FORMAT statement must be greater than the decimal field.
106	FORMAT WIDTH 0 INVALID	The total width field in a FORMAT statement must not be 0.
10	REPETITION FACTOR INVALID	The repetition count, which should be greater than zero, is not a valid integer.
108	NULL LITERAL INVALID	Literal data within apostrophes must contain at least one character.

Input/Output Errors

140	INVALID INTEGER	A real number has been found in the input file when an integer was expected.
141	INVALID REAL	A syntactically incorrect real number has been found in the input file.
142	INVALID SUBSCRIPT(S)	(NAMELIST only) An assignment to an array element failed because either too many or too few subscripts were specified or the subscripts were not unsigned positive integers.
143	INVALID COMPLEX CONSTANT	(NAMELIST and list-directed I/O only) A syntactically incorrect complex constant has been found in the input file.
144	VARIABLE IS NOT AN ARRAY	(NAMELIST only) A variable name in the input file is followed by a '(' but the variable is not an array.
145	NAME IS NOT FOLLOWED BY AN '='	(NAMELIST only) A variable name or array element in the input file is not followed by an '='.
146	VARIABLE NOT IN NAMELIST LIST	(NAMELIST only) An assignment to a variable in the input file is to a name which is not in the NAMELIST list.
147	INVALID ITEM	(NAMELIST and list directed I/O only) An entity in the input file has been detected as being misused or badly constructed.
148	INVALID CHARACTER	A character has been found in the input file which cannot syntactically be part of the entity currently being assembled.
149	INVALID VARIABLE NAME	(NAMELIST only) A variable name specified in the input file is an invalid FORTRAN name.

150	LITERAL NOT TERMINATED	(NAMELIST and list directed I/O only) A literal constant in the input file was not terminated by a closing quote before the end of the current record.
151	CHANNEL NOT DEFINED	An I/O request was made on a channel for which no definition has been supplied.
152	FILE DOES NOT EXIST	An attempt has been made to access a file which does not exist.
153	INPUT FILE ENDED	Data on the specified channel has been exhausted.
154	RECORD TOO LARGE	The record length as defined by a FORMAT statement, or implied by an unformatted READ or WRITE, exceeds the defined maximum for the current input or output file.
156	READ AFTER WRITE	An attempt has been made to read a record after a WRITE or ENDFILE statement.
157	WRITE AFTER ENDFILE	An attempt has been made to write a record after an ENDFILE statement.
158	RECORD NUMBER OUT OF RANGE	An I/O request on a direct-access data file has been made specifying a record which is outside the defined limits of the file.
159	NO FORMAT DESCRIPTOR FOR DATA ITEM	No corresponding format code in a FORMAT statement exists for a variable specified in the I/O list of a READ or WRITE statement.
161	INVALID RECORD SIZE	An invalid record size has been specified in a file definition.
162	NO WRITE PERMISSION FOR FILE	An attempt has been made to write to a file for which the user does not have write permission.
164	INVALID CHANNEL NUMBER	The channel number specified in an input or output statement is outside the range 1-99.
167	INVALID FILE NAME	A filename has an invalid construction.
169	OUTPUT FILE CAPACITY EXCEEDED	An attempt has been made to write more than the defined maximum capacity to a file.
171	INVALID OPERATION ON FILE	An input or output request is not consistent with the file definition, e.g. attempting to read from the line-printer.
172	WRONG LENGTH RECORD	The length of the record being read or written is inconsistent with the definition of the file being accessed.
179	FILE DESCRIPTION INCORRECT FOR D.A.	An attempt has been made to perform direct access operations on a file which has an inappropriate description, e.g. it may have variable length records.
180	FILE RECORD SIZE INCORRECT FOR D.A.	The record length specified in a FORTRAN DEFINE FILE statement is different from that specified in the file description.
181	FACILITY NOT AVAILABLE	An attempt has been made to use a facility which is not available to the current user.

Scientific Jobber Control Errors

201	INVALID USERNAME	Username missing or not satisfying required construction.
202	INVALID PARAMETER parameter	Invalid parameter specified in a control statement.
203	CPU LIMIT > PERMITTED MAXIMUM	Cpu limit requested in JOB statement exceeds the maximum permitted within the batch.
204	OUTPUT LIMIT > PERMITTED MAXIMUM	Output limit requested in JOB statement exceeds the maximum permitted within the batch.
206	INVALID CONTROL STATEMENT	Control statement syntactically incorrect or contains a command which is not in the prescribed set.
207	NO FILE ACCESS ALLOWED	A user who is not permitted to access files in the filestore has attempted to do so.
211	JOB CPU TIME EXCEEDED	The job has exceeded the cpu time limit requested in the JOB statement or set by default. The job is terminated.
212	JOB OUTPUT LIMIT EXCEEDED	The job has exceeded the printed output limit requested in the JOB statement or set by default. The job is terminated.
213	JOB TERMINATED BY OPERATOR	The job has been terminated by the machine operator for some reason.
214	INVALID KEYWORD	Invalid keyword used in a control statement.
215	TOO MANY PARAMETERS	More parameters have been specified to a control statement than were expected.

216	USER NOT REGISTERED FOR FILE USE	A user who is not formally registered in the system has attempted to use a file in the filestore.
218	FILE DOES NOT EXIST	A file does not exist in a situation where it is expected to do so. e.g. DELETE FILE LIST FILE DEFINE FILE with STATUS=OLD
219	FILE ALREADY EXISTS	A file exists in a situation where it is expected not to do so, e.g. DEFINE FILE with STATUS=NEW.
220	INVALID FILE NAME	A filename has an invalid construction.
221	SOURCE LIBRARY NOT DEFINED	Reference has been made to a library which has not been assigned prior to the call on the Jobber.
222	NO 'CURRENT' OBJECT FILE	An attempt has been made to RUN without a previous compilation or after a compilation has failed.
223	INVALID CHANNEL NUMBER	An attempt has been made to define a data file where the channel number is not in the prescribed range 1-98.
224	FILE NAME TOO LONG	A file name has greater than the permitted length.
227	TOO MANY EXTERNAL NAMES	An attempt has been made to RUN a program which has too many external (i.e. subprogram and COMMON) names.
228	PROGRAM TOO LARGE	An attempt has been made to compile and RUN a program which is too large for the Jobber.

FORTRAN (G) COMPILE-TIME ERROR MESSAGES

1	SYNTAX ERROR
2	NO EXECUTABLE STATEMENT ALLOWED IN BLOCK DATA
3	COLUMNS 1-5 OF CONTINUATION STATEMENT SHOULD BE BLANK
4	>19 CONTINUATION STATEMENTS
5	FIRST STATEMENT IS CONTINUATION
6	STATEMENT TOO COMPLEX TO COMPILE
7	STATEMENT INCOMPLETE
8	INCOMPLETE HOLLERITH CONSTANT
9	NON NUMERIC LABEL
10	name IS A PARAMETER
11	name IS ALREADY IN COMMON
12	COMMON MUST BE FULLY DEFINED BEFORE INITIALISATION
13	ILLEGAL COMMON BLOCK NAME
14	INVALID COMPLEX CONSTANT
15	INVALID CONSTANT
16	INVALID REAL NUMBER
17	INVALID STATEMENT NUMBER number
18	VARIABLE NAME EXPECTED
19	INVALID CHARACTER
20	CONSTANT NOT IN PERMITTED RANGE
21	BRACKETS NOT MATCHED
22	name IS A SUBPROGRAM NAME
23	name IS ALREADY USED OR INITIALISED
24	CONSTANT SHOULD BE INTEGER
25	CONSTANT NOT CONSISTENT WITH VARIABLE name
26	NOT ENOUGH CONSTANTS
27	VARIABLE IN BLOCK DATA NOT IN COMMON
28	TOO MANY CONSTANTS
29	COMMON MAY ONLY BE INITIALISED IN BLOCK DATA
30	ATTEMPT TO CHANGE TYPE OF VARIABLE name
31	ILLEGAL USE OF IDENTIFIER name
32	BLANK COMMON MAY NOT BE INITIALISED
33	FN NOT ASSIGNED
34	INVALID LENGTH SPECIFICATION
35	CONFLICTING DEFINITION OF name
36	<2 VARIABLES IN EQUIVALENCE LIST
37	>1 COMMON ITEM IN EQUIVALENCE LIST
38	ALIGNMENT ERROR
39	EQUIVALENCE ATTEMPTS TO EXTEND COMMON BACKWARDS
40	name EQUIVALENCED AFTER INITIALISATION OR USE
41	name IS NOT AN ARRAY NAME
42	ATTEMPT TO INITIALISE A VARIABLE AFTER USE
43	name INVALID IN DATA OR EQUIVALENCE LIST
44	INVALID SUBPROGRAM NAME name

```
 40     name EQUIVALENCED AFTER INITIALISATION OR USE
 41     name IS NOT AN ARRAY NAME
 42     ATTEMPT TO INITIALISE A VARIABLE AFTER USE
 43     name INVALID IN DATA OR EQUIVALENCE LIST
 44     INVALID SUBPROGRAM NAME name
 45     ARRAY name ALREADY DIMENSIONED
 46     CONTRADICTION IN EQUIVALENCE LIST
 48     INVALID ALPHABETIC SEQUENCE
 50     LABEL NOT SET
 51     LABEL SET TWICE
 53     ENTRY NOT ALLOWED IN DO LOOP
 54     STATEMENT NOT ALLOWED TO END DO LOOP
 55     CHECK DO LOOPS
 56     CHECK DO LABEL
 57     >32 NESTED DO STATEMENTS
 58     name IS NOT A SIMPLE INTEGER VARIABLE
 59     >7 IMPLIED DO LOOPS
 60     END= NOT ALLOWED IN D.A. READ
 61     INVALID ITEM ON LEFT OF '='
 62     LABEL PARAMETER NOT ALLOWED IN FUNCTION
 63     INVALID ARGUMENT name
 64     NO ARRAY ELEMENT ALLOWED IN STATEMENT FUNCTION
 65     INVALID ENTRY NAME
 66     INVALID ARGUMENT IN STATEMENT FUNCTION
 67     INVALID ARGUMENT name
 68     INVALID EXPONENT
 69     EXPRESSION SHOULD BE INTEGER
 70     INVALID AFTER IF
 71     INVALID EXPRESSION IN LOGICAL IF
 72     INVALID EXPRESSION
 73     INVALID COMPLEX OPERAND
 74     INVALID COMBINATION OF LOGICAL AND ARITHMETIC OPERANDS
 75     ARRAY NOT DEFINED OR MISPLACED STATEMENT FUNCTION DEFN
 76     LABEL number ALREADY USED AS FORMAT LABEL
 77     LABEL number ALREADY USED AS A STATEMENT LABEL
 78     ARRAY name DIMENSIONED AS A PARAMETER
 79     PARAMETER LENGTH OR TYPE IS INCORRECT
 80     WRONG NUMBER OF PARAMETERS
 81     WRONG NUMBER OF SUBSCRIPTS
 82     >7 DIMENSIONS NOT ALLOWED
 83     SUBSCRIPT TO ARRAY name EXCEEDS BOUNDS
 84     INVALID ARRAY DIMENSION number
 85     CHECK DIMENSION
 86     name IS A NAMELIST NAME
 87     name IS NOT AN ARRAY OR NAMELIST NAME
 88     FORMAT LABEL MISSING
 89     LABEL SET TWICE
 90     INVALID IMPLIED DO INDEX
 91     WRONG CLASS OF VARIABLE IN I/O LIST
 92     INVALID NAMELIST NAME
 93     NAMELIST LIST ITEM NOT SCALAR OR ARRAY NAME
 94     RECURSIVE STATEMENT FUNCTION DEFINITION
 95     INVALID SUBSCRIPT IN IMPLIED DO
 96     NESTED STATEMENT FUNCTION REFERENCE
 97     PROGRAM TOO LARGE
 98     TOO MANY NAMES IN SUBPROGRAM
 99     END STATEMENT MISSING
101     MISSING LEFT BRACKET
102     MISSING RIGHT BRACKET
103     NEGATIVE SIGN INCORRECT
104     INVALID FORMAT
105     DECIMAL FIELD > WIDTH
106     FORMAT WIDTH 0 INVALID
107     REPETITION FACTOR INVALID
108     NULL LITERAL INVALID
```

ALGOL (E) COMPILE-TIME ERROR MESSAGES

Compile time error messages take the form:

```
* sss FAULT nn (message)
```

where
- sss is the statement number of the offending statement
- nn is the fault number
- message is an abbreviated description of the fault

The fault number is provided for reference to the following list:

2	LABEL SET TWICE label	The current statement bears a label which has already been used to identify a statement in the current block.
4	SWITCH NAME NOT SET name	A statement of the form goto name[i] has been encountered where name is not currently declared as of type switch.
5	LABEL NAME IN EXPRSSN name	A name which is currently declared as a label has been found in an arithmetic or Boolean expression.
7	NAME SET TWICE name	The offending statement declares a name which is already declared or used as a label in the current block.
8	INVALID NAME IN VALUE LIST name	The offending statement is a procedure heading. Within this heading a name has appeared in the value list which does not appear in the formal parameter list for the procedure.
9	INVALID PARAMETER SPECIFICATION name	The offending statement is a procedure heading. A name appears in the specification part which has not appeared in the formal parameter list. This error can sometimes occur if the first begin of the procedure body is omitted or incorrectly positioned, so that the declarations of the procedure body are contiguous with the parameter specification.
10	PARAMETER INCORRECTLY SPECIFIED name	The offending statement is a procedure heading. One of the parameters in the parameter list has not had its type specified, or has been specified as a non-existent type (e.g. string by value).
11	LABEL NOT SET label	This fault refers to a statement containing a goto label, where label is not currently declared as of type label. This fault can also appear on a switch statement since labels appearing in a switch list must either be labels in the current block or currently declared as labels from outer blocks.
12	LABEL NOT ACCESSIBLE label	This fault may appear either on a goto statement or on the statement labelled by label. The label is on a statement controlled by a for statement and the offending goto statement is such that it could lead from outside a for statement to within the statement, the effect of which is undefined under Section 4.6.6 of the ALGOL Report.
14	TOO MANY ENDS	The statement referred to is an end statement which matches the hypothetical begin of the conceptually enclosing block which contains the declarations of the standard functions. The compilation will stop at this point.
15	MISSING ENDS	The program text has terminated without the end corresponding to the first begin of the program.
16	NAME NOT SET name	A name has been encountered which has not been declared and which is not the name of a standard function or standard Input/Output function. The name is forthwith declared in the innermost current block to be of the appropriate type, so that further occurrences of the name do not produce further diagnostics.
17	NOT PROCEDURE NAME name	A statement having the syntax of a procedure statement has been encountered but the name is not currently declared as a procedure name.
18	WRONG NO OF SUBSCRIPTS arr	A name, arr, currently declared as an array, has been used but the number of subscripts provided does not correspond to the number of bound pairs given in the array declaration.

19	WRONG NO OF PARAMETERS name	A procedure statement has been encountered where the number of parameters in the actual parameter list is not the same as the number of formal parameters in the procedure declaration.
20	PARAMETRIC ARRAY WRONG DIMENSION arr	The array presented as an actual parameter in place of the formal array arr has not the expected number of dimensions. It is not clear from the ALGOL Report whether or not it is valid ALGOL to present arrays of different dimensions as actual parameters to the same array formal parameter. ALGOL (E) insists that all arrays presented as actual parameters to the same formal parameter should have the same number of dimensions.
21	PARAMETRIC PROCEDURE NOT VALID proc	The procedure presented as actual parameter in place of the formal procedure proc does not have its own formal parameter list identical to that specified for proc by means of the special comment statement.
22	ACTUAL PARAMETER NOT PERMITTED parm	The actual parameter presented in place of the formal parameter parm is not permitted by the rules of formal-actual parameter correspondence, as defined in Chapter 5 and more formally in Sections 4.7.4 to 4.7.6 of the ALGOL Report.
23	PROCEDURE NAME IN EXPRSSN	A name currently declared as a typeless procedure has been discovered in an arithmetic or Boolean expression.
24	VARIABLE IN BOOLEAN EXPRSSN name	A name currently declared of type real or type integer has been found in a Boolean expression.
25	FOR VARIABLE INCORRECT	The controlled variable is not of type integer or type real or is a procedure name or a subscripted variable. It is not clear from the ALGOL Report whether or not a subscripted variable is permissible as a controlled variable. ALGOL (E) follows the recommendations of the IFIP working party and does not allow controlled variables to be subscripted, since this would be likely to result in the exact action of the loop being critically dependent on where in the loop the subscript is evaluated, thus producing programs which give quite different answers on different compilers.
26	DIV OPERANDS NOT INTEGER	The integer division operator has been encountered with operands of type real.
27	LOCAL IN ARRAY BOUND lname	This fault refers to an array declaration. One of the variables in the bound pair list for the array declaration is declared in the current block. This is clearly contrary to Section 5.2.4.2. of the ALGOL Report.
29	INVALID NAME IN LEFT PART LIST	The left part list contains a procedure name but the assignment is not within the body of the procedure or the procedure is not a type procedure or the name is of a different type from others in the left part list. Assignments to a procedure name are only valid with typed procedures and within the procedure body.
34	TOO MANY LEVELS	The current depth of nested blocks exceeds the compiler limit of 31.
35	TOO MANY PROCEDURE LEVELS	The current depth of nested procedures exceeds the compiler limit of 12 for ALGOL (E).
37	ARRAY TOO MANY DIMENSIONS	ALGOL (E) limits arrays to 12 dimensions.
40	DECLARATION MISPLACED	Declarations must be at the head of a block prior to any statements. N.B. A dummy statement is a valid statement. Consequently, begin ; integer I; would be faulted since the begin is followed by a dummy statement and the statement is followed by a declaration.
42	BOOLEAN VARIABLE IN EXPRSSN name	A name currently declared as of type Boolean has been found in an arithmetic expression.
43	ARRAY INSIDE OUT	An array with constant bounds has been discovered where the upper bound is less than the lower bound. This is treated in ALGOL (E) as an error.
47	ILLEGAL ELSE	An else follows an end but the begin corresponding to the end does not follow a then.
48	SUB CHAR IN STMNT	The substitute character, denoting an invalid character, has been found in the program text. The statement is ignored.

57	BEGIN MISSING	Declarations have been encountered within the hypothetical block which surrounds the user's program. The fist begin has probably been omitted.
99	ADDRESSABILITY	The compiler is unable to address a portion of the program. This occurs if the program exceeds 256K bytes of compiled code (K=1024) or if the program has very large own arrays.
102	WORK FILE TOO BIG	The output from the second pass of the compiler has overflowed the work file. The program must be segmented.
103	NAMES TOO LONG	The compiler's dictionary has overflowed. The program must be segmented. The dictionary holds 1023 names of average length six characters, including standard function names.
104	TOO MANY NAMES	See fault 103.
107	ASL EMPTY	The compiler tables are full. The program is too large and must be segmented. N.B. The compiler tables have been designed to enable programs of 10,000 average statements to be compiled.

IMP COMPILE-TIME ERROR MESSAGES

As the compiler processes a source program and produces a map or listing, it may encounter errors in the syntax or the semantics of certain statements. If this occurs, the compiler issues a message to describe the situation of the error and, as far as possible, its nature. Two main types of error are defined: syntactic errors and semantic errors.

Syntactic Errors

These are errors which cause the form of the statement to be unrecognisable, since the strict rules of syntax have not been obeyed. Mistakes such as omission of statement separators or misspelling of key words are examples. This type of error is indicated by the message

```
* sss   SYNTAX
```

where sss is the line number of the incorrect statement. A copy of the offending source statement is output on the next line of the program map or listing.

Where possible, the compiler indicates the exact position of the syntax error by outputting a marker (!) under the character where the analysis failed. Sometimes, however, this marker can only be approximate, as for example in the following case:

If the statement

```
%ROUTINE SPECIFY ORBITALS
```

were mistyped as

```
%ROUTINESPECIFY ORBITALS
```

the failure message would be

```
* sss SYNTAX ROUTINESPECIFY ORBITALS
                            !
```

The marker is misplaced to the right as the erroneous line more nearly corresponds to a routinespec statement than the intended routine statement.

The compiler will continue to process the remainder of a program after detecting such an error, but will not allow the program to be executed.

Semantic Errors

These occur when a statement is syntactically correct, but causes ambiguity in meaning or is totally meaningless. Examples are the declaration of a name for two uses in the same context, or the use of a name which has not been declared at all.

The following list describes the semantic errors detected by the compiler and diagnosed by messages of the type

```
* sss FAULT nn
```

where sss is the number of the faulty line and nn the number of the fault. The fault number will be followed by an abbreviated description of the fault which will usually be sufficient for the programmer to identify his mistake. Fuller descriptions of current compile-time faults are given below. If a program produces any fault not listed below then the user should contact the Advisory Service for further information. All will cause rejection of the offending program at the end of compilation.

1 TOO MANY repeats	A repeat instruction is encountered for which no cycle statement earlier in the same block can be matched uniquely.
2 LABEL SET TWICE name	The current instruction bears a label which has already been used to identify a statement in the same block. This will clearly cause ambiguity.
3 spec FAULTY	The offending statement is a spec in the short form, within a routine, function or map, which specified a name not appearing in the formal parameter list of the current block, or which appears in a begin block context.
4 switch VECTOR NOT DECLARED	A name used in the context of a switch label has not been declared as a switch name in the current block or routine.
5 switch LABEL ERROR	The subscript appended to the name used as a switch label is not a single integer constant or is outside the range defined by the declaration of the switch (see also fault 18).
6 switch LABEL SET TWICE	The current instruction is identified by a switch label which has already been used to label a statement in the same block.
7 NAME SET TWICE name	A declaration statement declares 'name', which is already set in the current block, except when the name is that of a routine, fn or map description which has been previously specified but not described within the block. If the statement declared a number of names, any of these not already set is set in the normal fashion. The diagnostic will appear once for each name which is already set.
8 TOO MANY PARAMETERS IN ROUTINE TYPE DESCRIPTION	A routine type description is encountered for which a declaration already exists, and the number of parameters declared in the description exceeds that in the declaration.
9 PARAMETER FAULT IN ROUTINE TYPE DESCRIPTION name	The type of a formal parameter name appearing in a routine type description differs from the corresponding parameter appearing in an earlier declaration.
10 TOO FEW PARAMETERS IN ROUTINE TYPE DESCRIPTION	A routine type description is found to declare fewer parameters than the corresponding specification.
11 LABEL NOT SET label	On encountering an end statement, it is found that the label has been referenced in a jump instruction in the current block, but has not been identified with any statement in that block. This message appears, therefore, immediately before the 'END OF BLOCK' message. Note that routine, fn or map descriptions are treated as separate blocks.
12 TYPE GENERAL PARAMETER MISUSED	An attempt has been made to store or fetch into a type general parameter (i.e. name). These are only used by the input/output routines and this fault should not normally be encountered.
13 repeat MISSING	This diagnostic appears at the end of a block, and indicates that a cycle has been opened in that block but has not been matched uniquely with a repeat instruction in the same block.
14 TOO MANY ends	An end statement is encountered which matches logically with the opening begin of the program. It should rightfully be an endofprogram. Compilation ceases and the job terminates. Any subsequent text is not scanned.

15 TOO FEW ends	An endofprogram statement is found to correspond to the opening of a block internal to the main program level, showing a lack of block ends.
16 NAME NOT SET name	A name employed in the offending instruction has not been declared. The name quoted is artificially declared as an integer so that this diagnostic is suppressed on later appearances of the name. However, other faults may occur later if 'name' is used subsequently in any other context, e.g. as a real name, which will cause fault 24.
17 NOT A routine NAME	A statement having the form of a routine call is encountered. The name quoted in this statement is that of an item declared as something other than a routine (fault 16 will indicate the case where the name has not been declared at all).
19 WRONG NUMBER OF PARAMETERS OR SUBSCRIPTS name	A reference to the name of an array is made, but the number of subscripts appended to it does not agree with the dimensionality declared for that array. This diagnostic also occurs when the number of actual parameters attributed to a routine call is not equal to the declared complement of formal parameters.
20 switch VECTOR OR recordformat NAME IN EXPRESSION	A name appearing in an arithmetic expression has been found to identify a switch, in which circumstances it is patently illegal. This also occurs if a recordformat name is found in an expression.
21 ROUTINE TYPE OR record NAME NOT YET SPECIFIED	A routine, fn or map named as a formal parameter of another routine type block is referenced in that block before the parameter has been specified (by a spec statement). Hence its own parameter list is unknown. This also occurs if a recordname variable is used before being specified.
22 ACTUAL PARAMETER FAULT	In a reference to a routine type name, an actual parameter is of incorrect type as defined in the declaration of that routine.
23 routine NAME IN EXPRESSION name	A name appearing in an arithmetic expression has been found to identify a routine, in which circumstances it is patently illegal.
24 REAL QUANTITY IN INTEGER EXPRESSION	In any expression assigned to an integer variable or otherwise expected to have an integer value, a real constant, variable or function name has been employed illegally. If the expression occurs in an array declaration, as a dimension bound, the array name remains set. This also occurs if a logical operator is found in a real expression.
25 cycle VARIABLE NOT integer TYPE	The control variable named on the left-hand side of a cycle assignment is not an integer variable. N.B. byteinteger and shortinteger variables are not permitted as control variables.
28 routine BODY NOT DESCRIBED name	This occurs at the end of a block when a specification appears in that block for a routine type name which is not described, whether referred to or not. The name given is that quoted in the offending routinespec.
29 LHS NOT A DESTINATION OR NAME IS NOT AN ADDRESS	In an assignment statement, the name appearing on the left-hand side of the assignment is not a variable name.
30 return OUT OF CONTEXT	A return statement occurs in a block other than a routine, in which circumstances it is meaningless.
31 result OUT OF CONTEXT	A result statement occurs in a block other than a fn or map body, in which circumstances it is meaningless.
34 TEXTUAL LEVEL > 8	This occurs immediately after the opening statement of a new begin block or routine type description which causes nesting of blocks to eight levels (the main program being at level 1, allowing for the compiler level). The compiler is unable to monitor this depth of nesting during execution, and hence subsequent object code produced is lost. However, the compiler contrives to scan subsequent text in the normal way for further syntactic or semantic errors.
37 ARRAY HAS TOO MANY DIMENSIONS	In this implementation, arrays are restricted to a maximum of twelve dimensions.
38 OVERFLOW	Overflow has occurred while compiling the statement. This is caused by using a constant that is too large for the type of variable involved.

39 REAL QUANTITY AS EXPONENT	A real constant, variable or parenthesised expression appears immediately to the right of an exponentiation symbol, in which position it is illegal. This diagnostic replaces the diagnostic 'fault 24' in these circumstances.
40 DECLARATIONS MISPLACED	Declarations must be placed at the head of the block in which they occur. In particular they must come before any labels, fault statements, jumps, conditional statements or cycles in the same block.
42 string VARIABLE IN ARITHMETIC EXPRESSION	In an expression deemed by context to be arithmetic, a string variable or the concatenation operator '.' has been found.
43 BOUND PAIR INSIDE OUT	In a declaration, a bound pair consisting of two constants has the lower bound greater than the upper bound. Both bounds are set to the lower bound and the declaration is accepted, in order to reduce the number of subsequent error messages.
44 CONST ERROR	A constant of incorrect type has been used to initialise an own variable. May be a real constant for an integer variable or too large a constant for the variable.
45 own ARRAY ERROR	An own array has been declared where the number of constants does not correspond with the bounds.
46 extrinsics	An attempt has been made to initialise an extrinsic variable. Extrinsic variables exist in a separately compiled module and thus cannot be initialised.
47 DANGLING else	An else has been found after a finish which is not associated with a condition.
48 SUBSTITUTE CHARACTER IN PROGRAM	The substitute character has been found in a line of program, which is listed. No attempt is made to analyse the offending line as this would merely result in a SYNTAX fault.
51 SPURIOUS finish	A finish has been found for which no start exists.
52 MISSING repeat INSIDE start / finish	This occurs at a finish statement and indicates that a cycle has been started within a start / finish block but that the corresponding repeat has not been found.
53 finish MISSING	Within a block or routine there exist more start statements than finish statements.
54 exit OUT OF CONTEXT	An exit statement has been found which is not within a cycle / repeat.
55 externalroutine IN PROGRAM	An externalroutine has been found in a program. externalroutines are only allowed in library files (see Section 6 of the Edinburgh IMP Language Manual, 1974).
56 endoffile OUT OF CONTEXT	An endoffile statement has been found in a program. endoffile is only allowed when compiling library files. This fault also occurs if endofprogram occurs when compiling a library file (see Section 6 of the Edinburgh IMP Language Manual, 1974).
57 LEVEL 0 USED	Statements have been found at level 0. The first begin has probably been omitted. When compiling library routines, own, const, external and extrinsic variables may be declared at level 0 and used to communicate between routines.
62 WRONG FORMAT	An attempt has been made to declare an entity of type record by reference to a name which is not currently declared as a recordformat.
63 recordspec IN ERROR	An attempt has been made to assign a format to an entity not of type record or to a record whose format is already known.
64 SUBNAME OMITTED	A reference to a record element does not specify a subname.
65 WRONG SUBNAME name	The indicated subname cannot be found in the recordformat statement referenced by the record declaration.
66 RECORD ASSIGNMENT	An attempt to assign one record to another cannot be compiled as the records are of different sizes.
69 SUBNAME OUT OF CONTEXT	A subname has been attached to an entity which is not of type record.
70 INVALID LENGTH IN STRING DECLARATION	The maximum length of the string being declared has either been omitted or lies outside the range 1 to 255.
71 STRING EXPRESSION CONTAINS A VARIABLE	A variable of type other than string has been found in a string expression.

72 STRING EXPRESSION CONTAINS INVALID OPERATOR	An operator other than '.' has been found in a string expression.
73 RESOLUTION COMPARATOR OUT OF CONTEXT	The resolution comparator '->' has been used with a variable which is not a string, or else in a double-sided condition.
74 RESOLUTION FORMAT INCORRECT	The bracketed expression is not correct. This is usually caused by the omission of the brackets themselves.
75 STRING EXPRESSION CONTAINS SUBEXPRESSION	Bracketed subexpressions are neither required nor permitted in string expressions. Brackets may only occur in string expressions as described under resolution (see Section 8 of the Edinburgh IMP Language Manual, 1974).
81 ITEM == exprn	The address assignment operation '==' has been used to equivalence a variable to an expression, which is patently absurd.
82 NOT AN ADDRESS	The address assignment operator has been used to assign an address to a variable that is not of name type.
83 NON EQUIVALENCE	The '==' operator has been used to equivalence operands that are not of identical type.
84 RECORD MISUSED	The special mapping function RECORD has been misused.
85 DIMENSIONALITY	An assignment to an array name, either explicitly via '==' or implicitly by parameter passing, cannot be completed as the two arrays have different dimensions. The dimension of the array name is defined by the first assignment to it encountered by the compiler.
98 ADDRESSABILITY	The program or one of its data areas exceeds the limit of addressability (at present 1/4 megabyte).

Catastrophic Compile-Time Faults

Some errors which exceed the physical limits imposed on the compiler by its particular operating environment are catastrophic and result in compilation ceasing at the point at which the error occurred. These faults have numbers greater than 100.

101 SOURCE LINE TOO LONG	The line of source text to be analysed is larger than the input buffer (currently a minimum of 300 characters). The statement should be broken down into several simpler statements.
102 LONG ANALYSIS RECORD	The current statement requires too many compiler recursions in its analysis. The statement should be broken down into several simpler statements.
103 DICTIONARY OVERFLOW	Too many or too long names are currently declared. The remedy is either * to use the block structure so that names are undeclared when not required, or * to use shorter names.
104 TOO MANY NAMES	See fault 103.
105 TOO MANY LEVELS	Textual level 10 has been reached.
106 STRING CONSTANT > 255 SYMBOLS	This fault will occur if a string constant contains more than 255 symbols.
107 ASL EMPTY	The compiler tables are full - this is unlikely to occur and the user should contact the Advisory Service.
108 END MESSAGE CHARACTER IN PROGRAM	The end message character was found in the IMP program. This could be caused by the omission of the endofprogram statement.

Faults greater than 200 indicate compiler errors, which should be reported to the Advisory Service.

G. COMPILATION AND EXECUTION UNDER VME/B

COMPILERS AVAILABLE

Compilers are available for ALGOL 60, IMP, IBM-compatible FORTRAN - FORTRAN (G), ICL New Range FORTRAN and COBOL. All compilers except IMP are ICL-supported. The ALGOL (E), IBM-compatible FORTRAN and IMP compilers are the same as those used in the Scientific Jobber.

The compiler-calling macros described in this Section provide for compilation followed (if desired) by execution. An object program may be kept as part of the process. The execution of previously compiled programs which used any of the ERCC written compilers, FORTRAN (G), ALGOL (E) and IMP, may be carried out using the macro EXECUTE, which is also described. There is also an outline description of the RUN macro, which may be used to execute previously compiled ICL New Range FORTRAN programs.

ALGOL

The ALGOL (E) compiler is a full ALGOL 60 implementation and is a variant of the compiler available on EMAS. It provides 1900-like input/output statements. Although released and supported by ICL the compiler was produced by ERCC. There is an ICL-written ALGOL compiler, which uses Knuth input/output, but the precise status and future of that compiler is uncertain. The language implemented by ALGOL (E) is described in the ERCC manual "Edinburgh ALGOL Language Manual" and the following description applies to that compiler.

The compiler is invoked by calling the macro ALGOLE in a job running under the default profile (RCOBATCH). The general form of the macro is

```
ALGOLE (parameter list)
```

The parameters which may be specified are as follows.

INPUT=filename	where filename is the name of a file containing the source program. This file must have been set up with the description *STDM or *STDTEXT. If no file is specified the source program is assumed to immediately follow the macro call as alien data.
OMF=filename	where filename is the name of a file in an already existing library, set up with the description DESC=*STDOMF. The object code resulting from the compilation will be put into the file, which will be created and saved by the macro, and may be executed subsequently by reference to the file in an EXECUTE macro. If no object file is specified a temporary one will be set up for use by the compiler. If a program contains multiple procedures the names of these procedures will appear in the library index as synonyms for the file and so could be accessed if referred to by another program.
OPTIONS=option list	where option list defines those options selected for the run. Items in the list must be separated by ampersands (&), e.g. OPTIONS = NOCHECK & NOARRAY. Possible options are given under the heading 'ERCC Compiler and Run-time Options' elsewhere in this Section.
RUN=NO	specifies that the compiled program is not to be run. Otherwise the program would be run if compilation were successful.

The macro initiates compilation of the source program and, unless RUN=NO is specified, runs the compiled program if the compilation is successful. Data for the default input channel (2 or 98) may be input as alien data immediately following any source program. In such a case the two sets of alien data must be separately delimited. The default channels for printed output are 1 and 99. Other files referred to by the program must be assigned to the appropriate channel, the local name being ICL9CEn, where n is the channel, e.g. ICL9CE7.

The following example illustrates the use of the macro ALGOLE to compile a program entered on cards and run that program with data for the default input channel also entered on cards. Printed output will be produced on the default output channel; the program also produces a permanent file if the program executes without an error apparent to the system. (If the SAVE_FILE call had been placed earlier in the JCP the file would have been made permanent regardless of successful execution.) The permanent file was written by the program on channel 24.

The object program is not kept.

```
JOB (:GLASO6.ALGOLTEST,TIME=15)
NEW_FILE(NAME=TESTOUTPUT)
ASSIGN_FILE(NAME=TESTOUTPUT,LNAME=ICL9CE24,ACCESS=W)
ALGOLE
----

source program

++++
----

data for channel 2 or 98

++++
SAVE_FILE(NAME=TESTOUTPUT)
ENDJOB
****
```

A more complicated example is given under IMP, but is equally applicable to the ALGOLE macro. An even more complicated example, involving two calls on the compiler, is given under IBM-compatible FORTRAN, but is equally relevant to ALGOLE.

The ALGOL (E) compiler uses ISO as the internal character code. The effect of this (probably none) is described in Section B under the heading 'Character Codes'.

Compile and run-time diagnostic messages are listed in Section F.

IMP

The IMP compiler is a development of that available on EMAS and at NUMAC. The language is described in the ERCC manual "Edinburgh IMP Language Manual" but the 2980 version of the language differs in some respects from the EMAS and NUMAC versions. The differences are described in Appendix 2 of this Guide. The compiler is supported on the 2980, but not by ICL, and may not be available on other 2900s.

The compiler is invoked by calling the macro IMP in a job running under the default profile (RCOBATCH). The general form of the macro is

```
IMP (parameter list)
```

The parameters which may be specified are as follows.

INPUT=filename — where filename is the name of a file containing the source program. This file must have been set up with the description *STDM or *STDTEXT. If no file is specified the source program is assumed to immediately follow the macro call as alien data.

OMF=filename — where filename is the name of a file in an already existing library, set up with the description DESC=*STDOMF. The object code resulting from the compilation will be put into the file, which will be created and saved by the macro, and may be executed subsequently by reference to the file in an EXECUTE macro. If no object file is specified a temporary one will be set up for use by the compiler. If a program contains an

external routine the name of this routine will be placed in the library index as a synonym for the file name so the routine can be accessed if it is referred to by another program.

OPTIONS=option list — where option list defines those options selected for the run. Items in the list must be separated by ampersands (&), e.g. OPTIONS = NOCHECK & NOARRAY. Possible options are given under the heading 'ERCC Compiler and Run-time Options' elsewhere in this Section.

RUN=NO — specifies that the compiled program is not to be run. Otherwise the program would be run if compilation were successful.

The macro initiates compilation of the source program and, unless RUN=NO is specified, runs the compiled program if the compilation is successful. Data for the default input channel (2 or 98) may be input as alien data immediately following any source program. In such a case the two sets of alien data must be separately delimited. The default channels for printed output are 1 and 99. Other files referred to by the program must be assigned to the appropriate channel, the local name being ICL9CEn, where n is the channel, e.g. ICL9CE7.

A simple example of the use of the macro ALGOLE is given under ALGOL above. That example is equally applicable to the macro IMP. A more complicated example for IMP, but applicable to ALGOL, is given below. There is a still more complicated example, which includes two calls on the compiler under IBM-compatible FORTRAN, but is equally relevant to IMP.

The example below involves compiling a source program in a file SPROG, linking that after compilation with two object files FRED and JOE, referring to a library REFLIB to find these and resolve other external references, and running the linked program with data on cards and also in an existing file GRUMP, read on channel 5. The object program kept in file UGH is the result of the compilation process; the linking in of FRED and JOE is done later for running only. UGH is to be held in an existing library OBJLIB, to which (for efficiency) the local name L is given. The SEARCH macro is described in Section H. UGH does not need to be assigned since it is referred to by a macro, not by a program. All files belong to user ERCC24. The program also produces printed output on channel 14.

```
JOB (:ERCC24.TRIALJOB7,TIME=50)
ASSIGN_LIBRARY(NAME=OBJLIB,LNAME=L,ACCESS=W)
SEARCH(NAME=REFLIB)
ASSIGN_FILE(NAME=GRUMP,LNAME=ICL9CE5)
WORK_FILE(LNAME=ICL9CE14)
IMP(INPUT=SPROG,OMF=*L.UGH)
----

data for channel 2 or 98

++++
SAVE_FILE(NAME=*L.UGH)        @ ONLY IF THE JOB GETS THIS FAR
ENDJOB
****
```

The IMP compiler uses ISO as the internal character code. The effect of this (probably none) is described in Section B under the heading 'Character Codes'.

Compile and run-time diagnostic messages are listed in Section F.

IBM-COMPATIBLE FORTRAN - FORTRAN (G)

The FORTRAN (G) compiler is intended to be as compatible as possible with IBM FORTRAN (level G) and is a development of the FORTE compiler on EMAS and at NUMAC. The language is fully described in the ERCC manual "Edinburgh FORTRAN Language Manual"; the differences between the FORTRAN (G) compiler, IBM FORTRAN and FORTE are listed in Appendix 1 to this Guide. The FORTRAN (G) compiler is fully supported by ICL, and in most cases the use of this compiler is to be preferred to one of the other ICL FORTRAN compilers, particularly for the development phase of a program, on the grounds of compatibility with compilers on other machines, good diagnostic facilities, and rapid correction of any bug which might be discovered. There should be no problem in transferring a program developed using FORTRAN (G) to, for example, the ICL Optimising compiler when that becomes available.

Internal Character Code

The compiler can, at the user's option, use either ISO or 2900 EBCDIC as its internal character code. In many cases it will make no difference to a program which code is used, but the ISO version is compatible with the FORTE compiler at NUMAC and on EMAS, and the EBCDIC version is compatible with other FORTRAN compilers on the 2980 and (apart from the slight differences between 2900 EBCDIC and IBM EBCDIC) with IBM compilers. In the version of the compiler distributed by ICL the default code is EBCDIC, and this is probably the better option if a program is to be run exclusively on the 2980. If a program is likely to be switched between the 2980 and EMAS the ISO option is generally more suitable. For simplicity on the 2980 the compiler is invoked by either of two macros, one setting EBCDIC as the default code, the other setting ISO.

Compiling and Running Programs

As noted above, the compiler is invoked by calling either of two macros in a job running under the default profile (RCOBATCH). The macro FORTRANG sets the default internal character code to 2900 EBCDIC; the macro FORTE sets the default code to ISO. In either case (though there seems little point) the code can be changed by use of the options parameter. The general form of each macro is the same:

```
FORTRANG (parameter list)

FORTE (parameter list)
```

where the parameters which may be specified are as follows.

INPUT=filename — where filename is the name of a file containing the source program. This file must have been set up with the description *STDM or *STDTEXT. If no file is specified the source program is assumed to immediately follow the macro call as alien data.

OMF=filename — where filename is the name of a file in an already existing library, or the name of a library.

If the name of a library file is specified all the object code resulting from the compilation will be put into the file, which will be created and saved by the macro, and may be executed subsequently by reference to the file in an EXECUTE macro. The name of the file overrides any program name which may have been specified.

If the name of an existing library is specified each object program or subprogram resulting from the compilation will be placed in a separate file in the library. The name of the file will be the program or subprogram name. If a name for a main program has not been defined in a PROGRAM statement in the source, the name ICL9CEMAIN is given. Care must be taken to avoid name clashes in the same library. Any entry points defined in a program will appear in the library index as synonyms for the file name. A program may subsequently be executed by reference to its name (or an entry point) in an EXECUTE macro. The library name should be followed by a full stop.

If no object file is specified a temporary one will be set up for use by the compiler.

OPTIONS=option list — where option list defines those options selected for the run. Items in the list must be separated by ampersands (&), e.g. OPTIONS = NOCHECK &

and Run-time Options' elsewhere in this Section.

RUN=NO — specifies that the compiled program is not to be run. Otherwise the program would be run if compilation were successful.

The macro initiates compilation of the source program and, unless RUN=NO is specified, runs the compiled program if the compilation is successful. Data for the default input unit number (5) may be input as alien data immediately following any source program. In such a case the two sets of alien data must be separately delimited. The default output unit number for printed output is 6. Other files referred to by the program must be assigned to the appropriate unit number, the local name being ICL9CEn, where n is the unit number, e.g. ICL9CE7.

The following two examples illustrate the use of the FORTRAN (G) compiler. In the first example a simple compile and run job is shown, where the default input and output units are used and both source and data are input with the JCP on cards. In this example the version of the compiler using ISO as the default internal character code is used.

```
JOB (:ERCC24.FORTRUN6,TIME=10)
FORTE
----

source program

++++
----

data for unit 5

++++
ENDJOB
****
```

The second example shows a source program held in a file SPROGM being compiled, linked with a module held in a library GENLIB, and run without any check for unassigned variables. The run creates a file BERT, written on unit 20, and a scratch (work) file on unit 44. It reads an existing file FRED on unit 33. It produces punched output on unit 8, and printed output on unit 9 (instead of, or as well as, on the default unit 6). It also reads data on cards included with the JCP on unit 1 (instead of on the default unit 5 - if the program also read from unit 5 it would take cards from the same data deck in the order of execution of the relevant READ statements).

```
JOB (:CLRN04.TESTRUN,TIME=120)
SEARCH(NAME=GENLIB)
NEW_FILE (NAME=BERT)
ASSIGN_FILE (NAME=BERT,LNAME=ICL9CE20,ACCESS=W)
WORK_FILE (LNAME=ICL9CE44,DESC=*STDM)
WORK_FILE (LNAME=ICL9CE8,DESC=*STDCP)
WORK_FILE (LNAME=ICL9CE9,DESC=*STDFORT)
ASSIGN_FILE (NAME=FRED,LNAME=ICL9CE33)
ASSIGN_FILE (NAME=*STDAD,LNAME=ICL9CE1)
FORTRANG (INPUT=SPROGM,OPTIONS=NOCHECK)
----

data for unit 1

++++
SAVE_FILE (NAME=BERT)
ASSIGN_FILE (NAME=IGLOO,LNAME=ICL9CE15,ACCESS=W)
FORTRANG
----

source program

++++
ENDJOB
****
```

The file BERT is made permanent if the first compilation and run was successful. Another program entered on cards is then compiled and run. This program writes on unit 15 to an existing file IGLOO. Note that if this second program refers to units 20, 44, 8, 9 or 33 it will use the files defined for the first program. These files will however have been closed at the end of execution of the first program and will be reopened at start of file if referred to in the second program. Reading file 44 would therefore be successful but any attempt to write to one of the files would overwrite the file.

RUNNING FORTRAN (G), ALGOL (E) AND IMP PROGRAMS

The macros FORTRANG, FORTE, ALGOLE and IMP provide for the compilation of a program and, if desired, its execution. They will not permit execution only, for which a macro EXECUTE is available. This macro can only be used with object programs produced by the ERCC-written compilers. The macro call is

```
EXECUTE (PROC=name)
```

where the name is that of the library file containing the main program, as determined by the OMF parameter when the program was compiled. The call on EXECUTE must be preceded by a SEARCH or EXLBL macro pointing to the appropriate library (or libraries, if the main program refers to modules in other libraries).

The following is a simple example of the use of EXECUTE to run a main program held in a file PROGFILE in the library OBJPROGS, specifying two libraries LIB1 and LIB2 to be searched to resolve external references. All files belong to the user running the job.

```
JOB (:GLASO2.RUNJOB)
SEARCH(NAME=OBJPROGS & LIB1 & LIB2)
EXECUTE(PROC=PROGFILE)
----

data for default input unit or channel

++++
ENDJOB
****
```

The examples given under the various compilers are all relevant to the use of EXECUTE if references to source programs are deleted. In particular, the second example under FORTRAN (G) is applicable to jobs where EXECUTE is called more than once. Complicated jobs involving the linking of many modules will probably require an understanding of the principles of loading and collection, described in Section H.

ERCC COMPILER AND RUN-TIME OPTIONS

The following options may be specified in the FORTRANG, FORTE, ALGOLE, IMP and EXECUTE macros. Items must be separated by ampersands, i.e. option1 & option2 etc. The default options are underlined.

LIST NOLIST	generate full source listings. generate minimal source listings (generally the first and last statements of each routine and erroneous statements).
CHECK NOCHECK	check at run-time for the use of variables which have not been assigned a value. do not check.
ARRAY NOARRAY	check at run-time that array subsripts lie within their defined bounds. omit array bound checking.
LINE NOLINE	include code to maintain a record of the line number of the statement currently being executed to assist in error diagnosis. omit line number updating.

DIAG	retain symbol tables for listing the values of variables during a routine trace back after an error has occurred.
NODIAG	omit symbol tables.
OPT	equivalent to the combined options NOCHECK, NOLINE and NODIAG.
INHIBIOF	inhibit the occurrence of an integer overflow interrupt.

The following options apply to FORTRAN (G) programs only.

LABELS	include code to maintain at run-time a list of the most recent labels, subroutine calls and returns, and to print this if an error occurs.
NOLABELS	omit label tracing.
MAP	at the end of each program print a list of identifiers and their attributes.
NOMAP	Do not print on identifier list.

The following option applies to ALGOL (E) programs only.

QUOTES	keywords are enclosed in quotes, e.g. 'BEGIN'
PERCENT	keywords are preceded by the % symbol, e.g. %BEGIN.

There is also the option EBCDIC or ISO in the case of FORTRAN (G) programs. This option selects the internal character code used by the compiler and has a permanent effect on the compiled program - it cannot be used with the EXECUTE macro. The default setting depends on the macro used to call the compiler: it is EBCDIC for the macro FORTRANG and ISO for the macro FORTE. With either of these macros the default can be overriden but the practice is not recommended.

ICL NEW RANGE FORTRAN

There are two ICL written FORTRAN compilers available, both of which implement the proposed ANSI standard as far as possible. The language is described in ICL Technical Publication 6800: "ICL 2900 FORTRAN Language". The basic compiler, F1, has been available for some time and is intended for applications, such as program development, where the speed of compilation is of more importance than very high execution efficiency. There is also an optimising compiler, OFC, which is based on F1 and should be used for production running of programs, particularly if execution time is over a few minutes. Users should not experience any significant language problems in transferring FORTRAN (G) programs to either compiler, though they should note that the three compilers may evaluate expressions in different orders.

Jobs using F1 must call the macro FORT. Jobs using OFC must call the macro FOPT. Both macros permit a source program to be entered as alien data immediately following the macro call. No default input or output unit numbers are provided for in either macro: any unit referred to in a program requires a matching WORK_FILE or ASSIGN_FILE statement. The local name for such a unit is ICL9LFn, where n is the unit number, i.e. not the same as for FORTRAN (G) or other ERCC compilers.

The following example shows the compilation and execution of a program using the F1 compiler. Apart from a different macro call it is the same as for the OFC compiler. The program reads data on cards on unit 5 and produces printed output on unit 6.

```
JOB (:LRCM04.FORTJOB,TIME=30)
ASSIGN_FILE (NAME=*STDAD,LNAME=ICL9LF5)
WORK_FILE (LNAME=ICL9LF6,DESC=*STDFORT)
FORT
----

source program

++++
----

data for unit 5

++++
ENDJOB
****
```

Parameters for the FORT and FOPT Macros

The following parameters may be specified in the FORT macro call.

RUN=NO	The compiled program will not be run. Normally the program will be run unless errors were detected during compilation.
OMF=libname.	The compiled program(s) will be placed in the named library, the name of which must be followed by a full stop. Each program or subprogram compiled as a result of the macro call will be placed in a separate file in the library, the file names being the same as the program names. Multiple entry points in the same program will be indicated by additional pointers in the library index. If the name of a main program has not been specified in a PROGRAM statement in the source program, the name ICL9HFMAIN will be given to it. The library cannot contain two programs with the same name, or identical entry names. The library must already exist, i.e. it must at least have been created by a NEW_LIBRARY macro.
INPUT=filename	This specifies the name of a file containing the source program. Any alien data supplied will be regarded as run-time data. If no INPUT parameter is specified the source program is expected as alien data following the macro call.

There are many other possible parameters, a description of which is beyond the scope of this Guide. For details see ICL Technical Publication 6858: "Developing FORTRAN Programs".

Running a Previously Compiled F1 or OFC Program

A complete program in OMF form can be executed by calling the macro RUN, specifying the program name (either that specified by a PROGRAM statement in the source program or ICL9HFMAIN).

Before the RUN macro is issued the library containing the module must have been made accessible by specifying it in an EXLBL macro, and any files used by the program must have been assigned. The form of the RUN macro is as follows.

```
RUN(PROCEDURE=programname)
```

or simply `RUN(programname)`

The macro can take other parameters (for tracing, etc.) but these are beyond the scope of this Guide. A simple example of the use of the macro to run a program producing printer output and reading in data on cards is given on the next page. When the program was compiled no name was specified, so the system gave it the default name ICL9HFMAIN. It is held in the library PROGLIB, which belongs to the user running the program. There can, of course, only be one module called ICL9HFMAIN in PROGLIB.

```
JOB (:SLDR37.FORTJOB,TIME=10)
EXLBL (NAME=PROGLIB)
ASSIGN_FILE (NAME=*STDAD,LNAME=ICL9LF5)
WORK_FILE (LNAME=ICL9LF6,DESC=*STDFORT)
RUN (ICL9LFMAIN)
----

data for unit 5

++++
ENDJOB
****
```

Resolution of External References

Any references to standard routines will automatically be satisfied when an object program is loaded. References to other routines will only be satisfied if the appropriate libraries have been specified in EXLBL macros.

COBOL

There are two ICL COBOL compilers, CO and C2. Both implement the same language specification and the future of the two compilers is not clear. It seems likely that CO will eventually disappear and users should preferably use C2. The language is described in ICL Technical Publication 6809: "COBOL Language". Detailed information on how to run COBOL jobs is given in ICL Technical Publication 6834: "Developing COBOL Programs", for CO, and Technical Publication 6866: "Developing COBOL Programs using the C2 Compiler".

Compilations using the CO compiler must call the macro COBOL; those using C2 must call COB. The macros will initiate execution of the compiled program, providing compilation was successful, unless the parameter RUN=NO is specified. An object module may be produced and placed in a file by specifying the parameter OMF=libname. Such a module may be executed later, as shown in the following example.

```
JOB(:ERDBO2.COBRUN)
EXLBL(NAME=OBJFILES)
WORK_FILE(LNAME=PRINT)
ASSIGN_FILE(NAME=*STDAD,LNAME=INPUT)
COBPROG
----

data read as "INPUT"

++++
ENDJOB
****
```

The program, COBPROG, is held in a library OBJFILES, belonging to use ERDBO2. This example is unaffected by the compiler used originally. Printed output is written to file PRINT in the program.

The following example shows the use of the macro COB to compile and run a program entered on cards, using data also entered on cards. If the names SYSIN and SYSOUT were used in the program for input and print output the WORK_FILE and ASSIGN_FILE statements would not be needed.

```
JOB(:ERDBO2.COMPRUN)
WORK_FILE(LNAME=PRINTOUT)
ASSIGN_FILE(NAME=*STDAD,LNAME=DATA)
COB
----

source program

++++
----

data for file "DATA"

++++
ENDJOB
****
```

H. LOADING AND COLLECTION OF OBJECT PROGRAMS

To run a previously compiled program it is necessary to load the relevant object module created at compilation time by specifying an OMF parameter, together with any other object modules needed to satisfy external references, and enter the object module at the appropriate entry point. In the case of object modules produced by any of the ERCC compilers FORTRAN (G), ALGOL (E) or IMP the macro EXECUTE may be used; in the case of ICL FORTRAN the macro RUN may be used. Both of these macros are described in Section G.

This method will normally be adequate during the development phase of a job, and for the production use of simple programs consisting of only a few modules. The efficient running of larger and more complicated program structures, however, requires some understanding of the loading process, and will probably benefit from the use of the Collector, an OMF module utility, at some stage in the creation of the production run job. This Section provides an introduction to loading and collection which will probably be sufficient for most users; the writers of large program packages however will need to read ICL Technical Publication 6850: OMF Utilities.

LOADING

An OMF module is generally directly loadable and will run if its references to any other code are satisfied, or can be satisfied by the Loader, a system utility program. It is possible to run a program merely by reference to its procedure name in the job control program, but this will only be satisfactory if the correct environment for the run has been set up first. Thus macros like EXECUTE and RUN set up the appropriate environment before loading and entering the specified program.

Program loading is performed by the Loader, which is invoked either by calling the LOAD macro or by calling another macro (such as EXECUTE) which explicitly or in effect calls the LOAD macro. The SCL statement

```
LOAD (FRED)
```

will load a module FRED, then load any other modules necessary to satisfy external references in FRED, any further modules necessary to satisfy external references in those modules, and so on. This method of loading, where only the first module is explicitly loaded, is known as cascade loading and is the method used by EXECUTE and RUN. There may however be several modules called FRED and there may be several modules with the same name as an external reference in FRED. In those circumstances the order of libraries in the user's library list is critical in determining which modules are selected.

Library Lists

A user's library list consists of any libraries he has explicitly declared in any EXLBL or SEARCH macros (described later), the most recently specified library being at the top of the list, followed by a number of standard libraries. The standard libraries consist of a basic set of system libraries and any declared in an already-called macro, or in the job profile. The order of the standard libraries is the order in which the libraries were added to the list, the most recent coming first and the basic system libraries coming last. A user will rarely be concerned with the standard libraries, but must take care in what order he adds his own libraries to the library list

Loading Sequence

The Loader is entered afresh for each procedure call and commences the search at the top of the library list, unless a library is specified with the procedure (or module) name, in which case only that library is searched. The Loader knows about procedures which are already in store, either directly or indirectly as a result of a load, and which libraries they came from. In searching for a procedure, therefore, the Loader starts with the first library in the list and checks whether a procedure with the right name from that library is already loaded. If not, the library is searched. Thus if procedures with the same name exist in the first and a later library in the list, even if that from the later library has already been loaded as a result of a load of a module containing that procedure, a reference to that procedure name will result in a load of the procedure in the first library in the list.

For example, if LIBRARYA contains procedures called JOE and BILL, and LIBRARYB contains FRED, JOE and BILL, where FRED calls JOE and BILL, then the sequence of commands

```
EXLBL (LIBRARYB)        @ CONTAINS FRED, JOE, BILL @
EXLBL (LIBRARYA)        @ CONTAINS JOE, BILL @
LOAD (FRED)
```

will result in the procedures JOE and BILL from LIBRARYA being loaded to satisfy the references in FRED. If, however, JOE and BILL were procedures in a module ROY, a (different) copy of which existed in both libraries, then the sequence

```
EXLBL (LIBRARYB)        @ CONTAINS JOE AND BILL WITHIN ROY @
EXLBL (LIBRARYA)        @ CONTAINS FRED, AND JOE AND BILL WITHIN ROY @
LOAD (LIBRARYB.ROY)
LOAD (FRED)
```

would result in the copy of ROY in LIBRARYB being loaded. The subsequent load of FRED would lead to the copy of ROY in LIBRARYA being loaded to satisfy the references in FRED to JOE and BILL.

Loading is block structured: the LOADER forgets about procedures it has loaded after the end of the block in which loading took place.

Efficiency Considerations

Although cascade loading is convenient during program development, for production running (at least of programs consisting of several modules) it is inefficient and it is worth considering collecting the various modules into a single one which can be loaded in a single operation. This can be done using a utility program called the Collector. Pre-loading of modules which would otherwise be loaded several times in the one run may also be appropriate, but is beyond the scope of this Section.

COLLECTING

The Collector utility merges a number of OMF modules to produce a single output module. In the process it satisfies references between the input modules and can suppress external names which are not required.

In the simplest case, which will often be adequate, the Collector is invoked by the command COLLECT, followed by a series of what are known as minor commands as alien data, finishing with the minor command PERFORM (no parameters). Relevant minor commands are INPUT, NEWMODULE (to specify the name of the output module) and SCAN (to specify libraries to be searched to resolve any external references which are still unsatisfied).

Format of Minor Commands

Minor commands look like SCL macro calls, but only take a single parameter in the form of a literal or superliteral (a series of literals separated by ampersands). There are no keywords. INPUT and SCAN can be specified several times and can be continued from one line to the next

* after the & in a superliteral parameter, and,

* in INPUT, after the '-' for omitted files.

INPUT

The INPUT minor command specifies the modules to be collected in the form of a list, each element of which is a module (file), a complete library, or a complete library except for certain named files. If a complete library is specified the name must end with a full stop. If a file is specified without the name of the library which contains it, the file name must be preceded by a full stop; the file is assumed to be in the most recently specified library. Files with a complete library may be omitted by specifying them with minus signs as in the example below. In that case the names need not be preceded by full stops. If a generation number is not specified the most recent generation is assumed. The following example

illustrates the various possibilities.

```
INPUT (LIBA. & LIBB.FRED & .JOE)
INPUT (LIBC. - MIKE - ALAN)              @ ALL LIBC EXCEPT MIKE & ALAN @
INPUT (LIBD.JANE)
INPUT (.MARY & .SALLY(3) & .ALICE(-1))   @ FROM LIBD @
```

NEWMODULE

This specifies the name of the output module (file) to be produced. If the named file already exists a new generation is created, unless an existing generation is specified, in which case the existing file is overwritten. If the named file does not exist it is created. The following are examples of NEWMODULE.

```
NEWMODULE (LIB.BILL)
NEWMODULE (LIB.TONY(6))
```

SCAN

The SCAN minor command specifies, in order, the libraries to be searched after all modules have been INPUT. In the following examples the libraries would be scanned in the order LIBA, then LIBB, then LIBC, etc.

```
SCAN (LIBA.)
SCAN (LIBB. & LIBC. & LIBD)
SCAN (LIBE.)
```

PERFORM

The PERFORM minor command, which has no parameters, is used to terminate a set of minor commands. It is optional, providing the set of minor commands is enclosed in alien data delimiters (---- and ++++).

Example of Collection

Suppose a, say, FORTRAN program CONFIT calls subroutines SQT, ELMER, MAXFN and LSTFN, and that ELMER calls ELMIN and ELMAX. Each module (main program or subroutine) occupies a separate file having the same name as the module, the files being distributed among three libraries, LIBA, LIBB and LIBC as follows:

```
LIBA:   CONFIT, SQT, ELMER
LIBB:   MAXFN, MINFN
LIBC:   ELMIN, ELMAX
```

Assuming that all three libraries belong to the person running the job, the commands

```
EXLBL (LIBC)
EXLBL (LIBB)
EXLBL (LIBA)
LOAD (CONFIT)       (or EXECUTE or RUN)
```

would load all the required modules. Seven separate loads however would be involved, together with 13 library searches. One could reduce the number of searches, but not the number of loads.

If CONFIT is to be run regularly it would pay to collect all the modules into a single module called, say, KONFIT. (Or one might collect the modules into a single module called, and replacing, CONFIT). This could be done with the following commands, putting KONFIT in LIBA.

```
EXLBL (LIBC)
EXLBL (LIBB)
EXLBL (LIBA)
COLLECT
----
INPUT (LIBA.CONFIT & .SQT & .ELMER)
INPUT (LIBB.MAXFN & .MINFN & LIBC.ELMIN & .ELMAX)
NEWMODULE (LIBA.KONFIT)
PERFORM
++++
```

The output module KONFIT could then be loaded with a single load and a single library search:

```
EXLBL (LIBA)
LOAD (KONFIT)
```

THE EXLBL AND SEARCH MACROS

These two macros are identical, SEARCH being retained for compatibility with past releases of the operating system. EXLBL (or EXLB) is the ICL-provided macro and adds the specified library to the top of the user's library list. Many examples of the use of this macro have already been given: in all cases EXLB or SEARCH could be specified instead of EXLBL. More than one library can be specified in each call of the macro, as in the following example which would place LIBC at the top of the user's library list.

```
EXLBL (LIBA & LIBB & LIBC)
```

I. UTILITY PROGRAMS

A number of commonly required utility functions are provided by programs or macros which have been described earlier in this Guide. These include the following file manipulation functions.

card to disc	INPUT // INPUT_FILE in Scientific Jobber
disc to printer	LIST LIST_FILE // LIST_FILE in Scientific Jobber
list a user's files	LIST_DIRECTORY // LD in Scientific Jobber
delete a file	DESTROY DELETE_FILE
program file operations	CREATE_LIBRARY UPDATE_LIBRARY MERGE_MODULES
source file editing	Amender (in Scientific Jobber)

In addition, this Section describes facilities performing the following operations on files.

disc to disc	FILE_COPY
disc to tape	FILE_COPY
tape to disc	FILE_COPY
tape to tape	FILE_COPY TAPE_COPY (for copying a complete tape, including copying or changing the tape label)
sorting	SORT
editing	various editors
disc to printer (binary)	RECORD_LIST
list a user's tapes	LIST_VOLUMES
catalogue a tape and its files	CATALOGUE_MULTIFILE_TAPE

There are also facilities for transferring IBM 370 files on tape to the 2980, and for producing tapes for transfer to IBM installations.

FILE_COPY

The macro FILE_COPY can be used to copy a file from one tape to another, from tape to disc, from disc to tape, or to produce another copy of a file on disc. The file description and organisation of the copy may be different from those of the original file. A new file is created to hold the copy unless the concatenation option is specified, in which case the copy is appended to an existing file. The macro has the general form

```
FILE_COPY (parameter list)
```

The following parameters may be specified.

INFILE=filename where 'filename' specifies the name of the input file.

OUTFILE=filename — where 'filename' specifies the name of the output file. If this parameter is not supplied the input file is copied to a fresh generation of the input file. The output file is a permanent file.

DESC=description — where 'description' is the name of a file description to be used for the output file. The default is the same description as for the input file.

ORG=type — where 'type' defines the type of output file. The possibilities are

S	serial
D	direct serial (for direct access files)
I	indexed sequential
P	object program file

The type can be different from that of the input file except in the case of program files. This parameter does not describe the organisation of the output file, which must be done where necessary by using the DESCRIBE_FILE macro, the description so set up being specified in the DESC parameter. See also the parameter COMPATIBLE.

SIZE=size — where 'size' is the maximum size, in units of 1024 bytes, of the output file. The default depends on the allocation used for the output file, specified in the next parameter. The defaults are

L	456K
F	228K
T	none

ALLOCATION=alloc — where 'alloc' defines the method of space allocation to be used for the output file. The default is L for serial files and object library files; F for other types. The possibilities are

L	sub-track allocation
F	full track allocation
T	tape

Wherever possible the default should be taken.

CFILE=filename — where 'filename' specifies the name of a controlling file to hold the output file. Controlling files are described in Section E and must be specified for demountable discs. For permanently mounted discs the system will make the necessary decision.

VOLUME=label — where 'label' specifies the label (serial number) of the disc or tape to which the output file is to be written. This parameter is only required for demountable discs or tapes. The volume concerned must have been catalogued before it is used for the first time, as will be the case if a tape or disc was labelled on the 2980.

APPEND=TRUE/FALSE — where TRUE results in the file being appended to the end of an existing output file. The default is FALSE.

COMPATIBLE=FALSE — This parameter must be specified if the input and output files differ in any of the following respects:

medium (disc or tape)
file description
file organisation

RESPONSE=variable — where 'variable' specifies the name of an SCL variable to hold a result code. The default variable name is RESULT.

TAPE_COPY

The macro TAPE_COPY provides a means of copying the entire contents of one tape, including if desired the tape label, to another. Copying ends when a double tape mark is encountered. The macro has the general form

```
TAPE_COPY (parameter list)
```

The following parameters may be specified.

INTAPE=label	where 'label' is the label (serial number) of the tape to be copied. This tape need not be catalogued. The label must be six alphanumeric characters starting with a letter.
OUTTAPE=label	where 'label' is the label of the tape on which the copy is to be made.
LABEL=YES	specifies that the output tape is to be labelled. But see below.
BLOCKSIZE=size	where 'size' specifies the maximum size of any block, in units of 1024 bytes, on the input tape. The default is 4.
RESPONSE=variable	where 'variable' is the name of an SCL variable to hold a result code. The default variable name is RESULT.

Where it is desired to produce an exact copy of a tape, including the label, the same label must be specified for INTAPE and OUTTAPE, as well as LABEL=YES being specified. If LABEL=YES the output tape will be labelled with the label specified in OUTTAPE, any existing label being overwritten. The practice of specifying the same label for both tapes should be avoided, since it leads to operational problems in identifying the original and the copy.

SORT

Users with complicated sorting requirements are recommended to read the ICL Technical Publications

6859	VME/B	File Manipulation Language
6863	VME/B	Data Management Utilities

A straightforward sort can however be done as shown in the following example.

```
JOB (:SRCP08.SORTEXAMPLE,TIME=40)
SORT (INPUT=FRED,OUTPUT=JOE,DETAILS=SCL)
----
MODULE SORT PARAMETERS
MAINSTORE IS 100000
DATASIZE IS 50000
WORKFILE GROUP FOR DA
FILENAME IS WORKONE; FILESIZE IS 100000
FILENAME IS WORKTWO; FILESIZE IS 100000
END
MODULE KEY SPECIFICATION
KEY IS BYTES FROM 1 TO 18, BYTES FROM 25 TO 26, BYTES FROM 22 TO 23, /+
BYTES FROM 19 TO 20 AS CHARACTER
ASCENDING
END
++++
ENDJOB
****
```

This job runs under the default profile RCOBATCH. The sort takes a file FRED and sorts it as specified in the following statements, the sorted result being the file JOE. The parameter DETAILS=SCL indicates that the sorting details follow as alien data. The input and output files may be the same, in which case the sorted output replaces the original contents of the file.

The sorting details are specified in the example in two modules. The module SORT PARAMETERS must always be supplied and defines the environment in which the sort is to be carried out.

MAINSTORE specifies the maximum store in bytes available to the sort utility. The amount shown in the example is about the minimum permissible size and the parameters should probably be increased for large sorts.

DATASIZE is an estimate of the size in bytes of the file to be sorted. A high or low estimate merely makes the sort less efficient.

WORKFILE GROUP parameters define the names and sizes of the workfiles to be used. The example is for direct access sort (the normal one). Two work files are needed. The names are arbitrary; the minimum size of each in bytes is given by the formula

$$.01d + nV + 13030$$

where d is the setting of the DATASIZE parameter, n is the number of records, and V is zero for fixed length records or 3 for variable length records. The validity of this formula is however in some doubt and a sensible choice is probably 2*d.

The module KEY SPECIFICATION defines the portions of each record to be used in sorting, and the order in which the records are to be put, either ASCENDING or DESCENDING. The example shows four fields being used, the fields being defined in hierarchic order and treated as characters (the alternative is BINARY). The /+ is a continuation symbol. In the example shown the input file contained the date in positions 19 to 26 of each record, in the form 25/02/77. The sort needed therefore to treat these bytes as three separate fields so as to order the file in ascending dates. It also sorted on positions 1 to 18, and in the output file records were ordered according to that field, and, if more than one record had the same field in positions 1 to 18, in date order. In the example all the fields are more than one byte long: one can however specify a single byte:

```
KEY IS BYTES FROM 3 TO 7, BYTE 9, BYTES FROM .....
```

Coding Rules for Specification Statements

The statements in the alien data file detailing the sorting operation must be punched in columns 1 to 72. If more than one statement appears on a line the first statement must be terminated with a semi-colon (;), otherwise the end of a line is regarded as a statement terminator. If a statement must be continued onto another line it should be broken where a space would appear, and the continuation symbol /+ inserted before column 73. It is probably advisable to leave a space in front of the continuation symbol.

Other Facilities available with SORT

The example shows a sort taking place on all records in the input file. It is possible also to specify that certain records are to be skipped, and what should be done if two records with identical keys are found (if it occurred in the example given the order of the two records in the output file would be unpredictable). These facilities are described in the publications given above. These publications also describe a MERGE utility.

EDITORS

Apart from the Amender, which runs in the Scientific Jobber, three editors are available, the ICL standard editor, LINEEDIT, an editor familiar to users of the OS service at NUMAC, and ECCE, familiar to users of EMAS. The ICL editor is very comprehensive but is basically a context editor, used more easily on-line, though capable of being used in batch mode. It is fully described in the ICL Technical Publication 6854 "Editing".

LINEEDIT is a straightforward line editor and is described in Appendix 4 of this Guide.

RECORD_LIST

The macro RECORD_LIST may be used to produce a listing of a file in both text and hexadecimal form. In its full form the macro will accept a number of parameters, but in nearly every instance the following form, where only a single parameter is specified, will be adequate.

```
RECORD_LIST (INPUT=filename)
```

The INPUT parameter specifies the name of the file to be listed. RECORD_LIST uses a utility held in the library ICL9TEDMUSUT and the macro must be preceded by a SEARCH macro call, e.g.

```
SEARCH (NAME=ICL9TEDMUSUT)
RECORD_LIST (INPUT=MYFILEFRED(6))
```

In that example the sixth generation of the file MYFILEFRED would be listed.

The format of the output listing is fairly obvious, except that each line begins with five characters which are not part of the record being listed.

LIST_VOLUMES

This macro produces a list of the volumes (in practice, tapes) owned by the user calling the macro. The macro has no parameters.

CATALOGUE_MULTIFILE_TAPE

If one produces a copy of a tape containing a number of files for back-up purposes and the original tape is damaged, the macro CATALOGUE_MULTIFILE_TAPE can be used to catalogue the files on the back-up tape and, if required, the tape volume itself. The macro has the form

```
CATALOGUE_MULTIFILE_TAPE (parameter list)
```

where the possible parameters are

FNAME=list	'list' specifies the names of the files in order of occurrence on the tape, in the form name1 & name2 & name3 etc.
VNAME=label	'label' is the label of the relevant tape.
SECTNO=number	where 'number' is the number of the file at which cataloging is to start. If this is 1 (the default) the tape itself is catalogued also.
RESPONSE=variable	where 'variable' is the name of an SCL variable to hold a return code. The default variable name is RESULT.

If the files to be catalogued are already catalogued, but on a different tape, as would be the case if files were being recatalogued from copies on a back-up tape, the catalogue entries must first be deleted by using DESTROY or DELETE_FILE. The entry in the catalogue for the original tape volume can then be deleted by calling the macro DELETE_VOLUME, specifying the label of the tape concerned, i.e.

```
DELETE_VOLUME (NAME=label)
```

TRANSFER OF IBM 370 FILES

The utility program available to transfer files from IBM 370 machines to the 2980 is fully described in the ICL Technical Publication 6849 'Transferring IBM 360/370 Data Files,' which also describes the process for transfer in the opposite direction. The following abbreviated description of the utility will probably, however, prove sufficient for most users. The operation will be carried out on behalf of a user by the local Service Support Unit but the request must be accompanied by sufficient information, either in written form or as punched utility control statements, to enable the transfer to be accomplished.

J. BIBLIOGRAPHY

The following is a short list of documentation relevant to the 2980 service. The use of ICL Technical Publications should be approached with caution, not because they are bad (frequently they are very good) but because they describe the full range of VME/B software, not all of which is necessarily available on the 2980. Users are recommended to seek such advice as may be available before using a facility not described in this Guide.

Edinburgh FORTRAN Language Manual (ERCC)

Edinburgh ALGOL Language Manual (ERCC)

Edinburgh IMP Language Manual (ERCC)

TP6800 ICL 2900: FORTRAN Language

TP6858 Developing FORTRAN Programs

TP6809 COBOL Language

TP6834 Developing COBOL Programs

TP6866 Developing COBOL Programs using the C2 Compiler

TP6859 VME/B File Manipulation Language

TP6866 VME/B Data Management Utilities

TP6854 Editing

TP6850 OMF Utilities (covers loading and collecting)

Each university has a complete set of ICL Technical Publications. Further copies must be purchased from ICL if required. The local Advisory Services can advise on the availability of ICL documentation and obtain any manuals published by ERCC.

K. MACROS DESCRIBED IN THIS GUIDE

The following macros are described in this Guide. Those which have changed with the introduction of the SV21 release of the operating system are marked with an asterisk. Scientific Jobber and Collector commands are not included.

Macro	Section	Macro	Section
* ALGOLE	G	* FORTRANG	G
ASSIGN_FILE	E	* IMP	G
ASSIGN_LIBRARY	E	INPUT	E
CATALOGUE_MULTIFILE_TAPE	I	JOB	D
CHANGE_DELIMITERS	E	LIST	E
CHANGE_FILE_ACCESS	E	LIST_DIRECTORY	E
CHANGE_FILE_RELATIONSHIP	E	LIST_FILE	E
* COB	G	LIST_VOLUMES	I
* COBOL	G	LOAD	H
COLLECT	H	* NEW_DAFILE	E
* DELETE_FILE	E	* NEW_FILE	E
DELETE_VOLUME	I	NEW_LIBRARY	E
* DESTROY	E	RECORD_LIST	I
ENDJOB	D	* RUN	G
* EXECUTE	G	RUNJOB	D
EXLB	H	SAVE_FILE	E
EXLBL	H	SEARCH	H
* FILE_COPY	I	SORT	I
FOPT	G	TAPE_COPY	I
* FORT	G	WORK_FILE	E
* FORTE	G		

APPENDIX 1: CHARACTERISTICS OF FORTRAN (G)

DIFFERENCES BETWEEN FORTRAN (G) ON 2980 AND FORTE ON EMAS AND NUMAC

The following is a summary of the differences between FORTE implementations on EMAS and NUMAC and FORTRAN (G) of which users should be aware:

1. The free-format facilities are different. The 2900 facilities are described in the following pages.

2. Array bound checking. If array bound checking is switched on, arrays will be checked in accordance with their definition in the current program unit. If, however, a dummy array has been dimensioned as, e.g. A(1), that is with only one element, then the array bound checking is carried out with respect to the definition of the actual array passed. This means, among other things, that programmers who have used this form when dimensioning array parameters may now compile their programs with checks on. Note that the checking is done by hardware and that there is thus no overhead in asking for it. Thus users should be encouraged to use it. Note that these remarks do not in general apply to dummy arrays of more than one dimension.

3. Alignment of variables in EQUIVALENCE and COMMON statements. Hitherto it has been a requirement that variables of type REAL*8 (or DOUBLE PRECISION) have had to be placed at a double-word boundary. The user only has control of such placements when giving COMMON or EQUIVALENCE statements, and there he has been told that such as the following are invalid:

```
REAL*8 A,B
REAL*4 C(3),D
EQUIVALENCE (C(2),A)
COMMON D,B
```

   Both the COMMON and the EQUIVALENCE statements above would fail in FORTE, but would be acceptable to FORTRAN (G).

FREE FORMAT INPUT/OUTPUT STATEMENTS

Warning

These statements are not part of 1966 ANS FORTRAN but do follow the proposed ANS FORTRAN, IBM FORTRAN (G1) and ICL New Range FORTRAN.

Free-format input differs from standard input in that the layout of the data on the cards need not be fixed in advance, nor need it be constrained to predetermined fields on each card. However each number must be bounded by a recognisable separator; see 'Free-format input and output data' for details.

This facility is also known as 'list-directed' I/O.

READ statement

The statement has the form

```
READ(i,*, END = m1, ERR = m2) list
```

i, ERR, END and list have the same meanings as in the description of the standard READ statement (5.2).

* replaces the specification, in the standard READ statement (5.2), of a FORMAT specification, or a NAMELIST name. Its presence indicates that the statement is a free-format READ statement.

Execution of a free-format READ statement causes data to be read from the data set corresponding to i. The data values are assigned to the elements specified by the list, and as many records of the data set as necessary to satisfy the list are used. Reading starts at the beginning of a new record, so that data values will be skipped if the previous READ statement did not read all of the previous record. See also below, 'Free-format input and output data', which defines the layout of data in a data set to be read by free-format READ statements.

The BACKSPACE statement may not be used with free-format data.

Another form: READ *,list

is also permitted. The data set reference number is assumed to be 5 (the card reader), and the END and ERR parameters cannot be specified.

WRITE statement

This statement has the form

WRITE(i,*) list

i and list have the same meanings as in the description of the standard WRITE statement (5.2).

* replaces the specification, in the standard WRITE statement (5.2), of a FORMAT statement label, or the name of an array contained a FORMAT specification, or a NAMELIST name. Its presence indicates that the statement is a free-format WRITE statement.

Execution of a free-format WRITE statement causes the next record to be created on the data set corresponding to i. For further details, see below, 'Free-format input and output data'.

The following two forms:

PRINT *, list
PUNCH *, list

are also permitted. The data set reference number for the PRINT statement is taken to be 6 (the line printer) and, for the PUNCH statement, 7 (the card punch).

FREE-FORMAT INPUT AND OUTPUT DATA

Free-format input data

A record containing free-format ('list-directed') input data consists of an alternation of constants and separators.

Separators

A separator consists of:

* a comma
* a blank
* an end-of-record condition (end-of-card with card input)
* a comma preceded or followed by one or more blanks or end-of-record conditions

At the start of execution of a free-format READ, a preceding separator is assumed, and initial blanks or end-of-record conditions, if present, are considered part of that separator.

Constants

An input constant may be any valid FORTRAN data type, except that a literal constant must be enclosed in quotes. Blanks may not be embedded in any free-format constant except a literal constant, since they would be interpreted as separators. Blanks may however be inserted around the two numbers making up a complex constant. Numeric constants may optionally be signed, but there must be no embedded blanks between the sign and the constant.

Null items

A null item is represented by two consecutive commas with no intervening constant. Any number of blanks or end-of-record conditions may be embedded between the commas. If a null item is specified, the corresponding list item is skipped; its current value remains unaltered.

A repetition factor may be specified for a constant or null item. For a constant, the form is

```
i*constant
```

and for a null item, the form is

```
i*
```

In each instance, i is a non-zero integer constant which indicates that the following constant or null item is to occur i times. When a repetition factor is applied to a constant, no blanks should follow the asterisk, since the repetion factor would then be taken to apply to a null item.

Special-purpose separator (/)

A slash (/) serves as a special-purpose separator, indicating that no more data is to be read during the current execution of a free-format READ statement. The slash may be surrounded by any number of spaces or end-of-record conditions; these do not constitute additional separators. If the list has not been satisfied, the value of the remaining list elements remain unaltered. If the list has been satisfied, the slash has no effect.

Example

In the following example, a free-format READ statement is used to read a record containing constants of various types into main storage. Correct type specifications are assumed throughout.

```
READ(5,*) (ARRAY(I),I=1,50),HD1,HD2,A,B,C,D,E,F,G,H,J,P

      ⎧50*0.   'HEADING'  'FOOTING'
Data  ⎨
      ⎩(2.17E+15, 3.14E0), 1.,2.0,0.125D-3,2* ,.TRUE.,,8/
```

When this READ statement is executed, the value 0. is read into each of the first 50 elements of ARRAY; text is read into the locations HDI and HD2; a complex is read into location A; three values are read into locations B, C and D; locations E and F are skipped because of the repeated null specification; the logical value .TRUE. is placed in location G; location H is skipped; location J receives the value 8; and location P receives no data because of the terminating slash.

Free-format output data

Free-format output may contain any reproducible form of data which is readable as free-format input. However, certain forms which are permissible as free-format input are not produced as free-format output. For example, literal constants are never produced by free-format output, and neither are repetition factors.

A separator is generated after each data item, including the last item of a record.

REFERENCES

Paragraph numbers in the foregoing refer to the relevant paragraphs in the "Edinburgh FORTRAN Language Manual".

FORTRAN (G) FEATURES NOT IN ANS FORTRAN

Call by location

Direct Access Input/Output Statements

Double Exponentiation

END and ERR parameters in READ

ENTRY

G format for integers and logicals

Generalised subscripts

H code in variable format

Hexadecimal constants

IMPLICIT

Initial data values in type statement

Length of variables as part of type specifications

Literal enclosed in apostrophes

Mixed mode expressions

More than 3 dimensions in an array

NAMELIST

PAUSE message

PRINT *,list

PUNCH *,list

READ (i,*,END=1,ERR=m2) list

READ, list

Rentention of local variables in subprograms

T and Z format codes

RETURN i

WRITE (i,*) list

$ considered alphabetic

IMPLEMENTATION DIFFERENCES BETWEEN FORTRAN (G) AND IBM LEVEL G FORTRAN IV

1. A program will not execute with incorrect alignment of variables due to layout of a COMMON block or forced by EQUIVALENCE statements.
2. There is no limit on the number of comments.
3. The effect of the PAUSE statement is different.
4. Array subscripts may not be of the type REAL.

5. The order of statements allowed by FORTRAN (G) is in some ways slightly more restrictive than that allowed by the ANSI FORTRAN standard or by IBM level G. The following order for executable subprograms is recommended, although deviations from it are not necessarily wrong. In general a variable name should first have its length and dimensions declared and may then be equivalenced and finally initialized.

    a. SUBROUTINE or FUNCTION statement, if appropriate

    b. IMPLICIT statement

    c. COMMON, DIMENSION and TYPE statements

    d. EQUIVALENCE statements

    e. DATA statements

    f. NAMELIST statements

    g. Statement function statements

    h. Executable and FORMAT statements

    i. END statement

    Notes:

    * Variables which later appear in an EQUIVALENCE statement must not be initialized in a type statement.

    * If integer variables are used as array dimensions in a subprogram and are also in COMMON, the COMMON statement must precede the dimensioning statement.

APPENDIX 2: CHANGES IN 2900 IMP

The following is a summary of the differences between 2900 IMP and the versions available on EMAS or at NUMAC.

1 Abolition of the modulus operator '!...!' coupled with the introduction of a standard function IMOD to yield the absolute value of an integer (parallel to MOD for reals).

The change removes a source of ambiguity.

2 Abolition of the use of a special symbol for π in favour of a standard function PI.

The special symbol is not found in standard character sets.

3 Re-specification of the standard procedures LENGTH and CHARNO as maps rather than functions.

The extension permits a number of string-manipulation operations to be effected more neatly.

4 Abolition of numeric labels.

Most current programming languages provide only alphabetic labels so that labels may be meaningful, in the same way as other identifiers. The need to use labels at all in IMP, as compared with Atlas Autocode, is greatly reduced by the availability of such program-structuring facilities as compound conditional statements and loop-control clauses.

5 Restriction of the division-operator '/' to yield a result of type real in all cases.

The integer-division-operator '//' would become obligatory where an integer result is intended. The effect of this and the next three changes is to remove all remaining cases of type ambiguity from the language.

6 Restriction of the exponentiation-operator '**' to yield a result of type real in all cases, coupled with the introduction of an integer-exponentiation operator '****'.

The case of exponentiation becomes exactly parallel to the case of division.

The reverse slant, \ for real and \\ for integer, is a more elegant alternative notation for these operators.

Note that K=I**J, where I, J and K are integers, would fail, since the right hand side is a real expression.

7 Restriction of the form of integer constants to disallow the inclusion of decimal points and exponent symbols.

A constant form like 0.1@1 was technically an integer because it has an integral value. Such forms are not allowed in 2900 IMP.

8 Introduction of a distinctive quote symbol for strings (double-quote in place of single-quote).

The main effect is to make it possible to distinguish a single character string from a character constant. For the time being both single and double-quote is allowed.

9 Extension of the set of ranges of integer to include long integer (64-bits).

The 2900 implementation offers the capability to handle 64-bit integers. Implementation of this feature on existing systems is not possible.

Note the following:

```
integer I,J; longinteger K

K = (I<<32)!J ; !  EFFECT is K = J, SINCE RHS USES 32-BIT WORKING.

K = I; K =(K<<32)!J ; !  THIS ASSIGNS K AS INTENDED
```

Thus, integer arithmetic will be done with a precision of 32 bits if all the operands are of 32 bit length.

A longinteger can be shifted using the << and >> operators, but an attempt to shift either way 64 bits will have no effect (it might be expected to clear a longinteger to 0).

10 Revision of the specification of the precision of reals.

The 2900 implementation offers three precisions for reals: 32-bit (somewhat inefficient), 64-bit and 128-bit. These are specified as real, longreal and longlongreal. Working is either in 64 bits or in 128 bits. Thus type real is only effective in storing values in scalars and arrays. Any variable of type real will have its value lengthened when used in any calculation.

11 Abolition of short integers.

12 The following in-line functions for changing lengths of reals and of integers have been introduced:

LENGTHENI(K)	Parameter of type integer, result of type longinteger
SHORTENI(K)	Parameter of type longinteger, result of type integer
LENGTHENR(S)	Parameter of type longreal, result of type longlongreal
SHORTENR(S)	Parameter of type longlongreal, result of type longreal

13 The grave sign (`) will now stand for itself; i.e. it is not now to be mapped onto @ on input.

14 The command monitorstop is no longer available. It will now be necessary to give the two commands monitor; stop.

15 Implied multiplication is not allowed.

On EMAS, constructions such as 23A(6) are allowed; i.e. constant followed, without an asterisk, by a variable or array element. This is not allowed on the 2900.

16 Abolition of fault trapping in favour of an event mechanism.

The architecture of the 2900 Series has enabled a more structured approach to be adopted and the following notes describe the current implementation.

The term event is introduced to describe the class of conditions which may be detected or signalled, and is broader than the fault concept in that it may include user-defined events as well as the conventional 'faults' - e.g. division by 0.

The following list groups certain faults to provide a set of events, of which 1-10 are predefined; events 11-14 may be user defined.

Event classes

No.	Name	Old Numbers
1	Arithmetic overflow	1,2,17,22,23,24,27
2	Excess store (or other resource)	4
4	Substitute character in data	18
4	Invalid data	14
5	Invalid arguments	3,5,6,7,21,28
6	Out of range	30,32
7	Resolution failure	26
8	Unassigned variable	31
9	Input ended	9
10	Library subroutine error	25
*11-14	GENERAL PURPOSE	

*11 is also used by the graph package (old fault 19)

Within an event class, the individual faults are assigned different sub-event numbers. The existing faults are thus categorised as follows:

Old Number	Description	New event/sub-event no.
1	Integer Overflow	1/1
2	Real Overflow	1/2
4	Invalid cycle	5/1
4	Not Enough Store	2/1
5	SQRT Negative	5/2
6	LOG Negative	5/3
7	SWITCH Variable Not Set	5/4
9	Input Ended	9/1
10	Non-integer Quotient	*
11	result Not Specified	*
14	Symbol in Data	4/1
16	Real Instead of Integer in Data	*
17	Divide Error	1/3
18	Substitute Character in Data	3/1
19	Graph Plotter Fault	11/n
21	Illegal Exponent	5/5
22	Trig Function Inaccurate	1/4
23	TAN Too Large	1/5
24	EXP Too Large	1/6
25	Library Function Fault	10/n
26	Resolution Fails	7/1
27	INTPT Too Large	1/7
28	Array Inside Out	5/6
30	Capacity Exceeded	6/1
31	Unassigned Variable	8/1
32	Array Bound Fault	6/2

The following existing faults are omitted from the event classes:

* Non-integer quotient (10) - redundant in 2900 IMP

* Result not specified (11) - will be made redundant by adding a compile-time check

* Real instead of integer in data (16) - it is proposed that an integer 'read' should terminate on a '.' sign or '@' sign, in addition to its present definition.

The syntactic structure is:

```
on event nlist start        e.g. on event 1,7,12 start
   .                                .
   .                                .
   .                                .
finish                          finish
```

This structure must follow the declarations at the head of a block (routine) and may be regarded as the last declaration of the block. The code within the start ... finish is not executed on entry through the head of the block but is jumped to should an event which is contained within the list occur. The flow of control then depends on the contents of the start ... finish.

An event may be forced by the unconditional instruction:

signal event n, exprn

where n is the event required and 'exprn' is an optional integer expression which may be used to specify sub-event information. n must be given as a constant, and 'exprn' must yield an integer in the range 0-255.

signal event statements are the only way of causing user-defined events to occur, although they can also be used with the predefined events (1-10).

If an event is forced by a signal event statement in an on event start/finish block which includes the occurring event in its event list, a branch is not made to the head of that block, since such a branch would probably cause looping. Instead the event is traced up the stack through each superior block until either a suitable on event statement is found or the user environment is left.
In parallel with these language statements two integer functions are introduced which enable the programmer to determine further information when an event occurs. They may only be meaningfully called in a block which has an on event statement within it:

integer fn event inf

returns (event no <<8)!sub-event no. Fault 16 occurs at compile time if the function is called in a block with no on event statement, and an unassigned variable will result at run time if no event has in fact occurred when the function is called.

integer fn event line

returns the program line number at which the last event occurred during execution of the block in question (provided the program was compiled with line number updating; otherwise 0 will be returned). If no event has occurred, an unassigned variable will result.

If an event is not trapped in the block in which it occurs then it is traced up the stack through each superior block until either a suitable on event statement is encountered or the user environment is left, the diagnostic package being entered in the latter case. When a suitable on event statement is encountered in a superior block, program control is transferred to its start ... finish block.

As a result of these facilities it follows that, for example, 'input ended' may be detected and dealt with within an external routine or a routine within a main program.

APPENDIX 3: SCL RESERVED WORDS

The following are reserved words and should not be used as the names of entities by the user:

AFTER	FALSE	MACBEGIN	STARTSWITH
AND	FI	MACEND	STATUS
	FILL	MACRO	STINT
BEFORE	FIND		STRING
BEGIN	FOR	NE	SUBSTR
BIN	FROM	NEQ	SUPERLITERAL
BOOL		NOT	SUPERSTRING
BOUND	GE	NUMERIC	SYSCALL
BY	GOTO		
	GT	OR	THEN
CHARTOINT			TO
CLOCK	HEX	PROC	TRUE
CODE	HEXTOCHAR		
COUNT		REF	UNLESS
	IF	REPEAT	UNTIL
DIGITS	INCLUDES	RESIDUAL	
DO	INDEX	RETURN	VAL
	INT		
ELSE	IS		WHENEVER
ELSF			WHILE
END	LE		
ENDSWITH	LENGTH		
ENTER	LITERAL		
EQ	LOAD		
EXT	LT		

Note that although not all the above are currently part of the syntax of SCL their use should be avoided to assist with forward compatibility of user-written SCL.

APPENDIX 4: LINEEDIT

LINEEDIT is a text editor written by Dr. P. Moran of the University of Glasgow Computing Service. It acts upon any standard magnetic file whose records are 80 bytes or less in length. Although LINEEDIT is basically a line editor, i.e. complete cards are inserted into or deleted from the text, there are some simple commands which enable changes to be made to part of a card without re-typing the whole card.

A file can be edited either to create a new file or to replace the original. In the latter case the new version is given a generation number one greater than the original file.

The operation of LINEEDIT is guided by control cards: options cards or edit cards. The options cards enable the user to control aspects of the editor's behaviour, such as the type of listing produced by it. The edit cards specify what has to be done.

Before giving the formal definition of the control cards, here first is a simple example of a run of LINEEDIT. This example shows a job step used to update the file MAIN owned by user in whose job LINEEDIT is called.

```
LINEEDIT (NAME=MAIN)
----
#10
          PRINT 6
#14,17
          CALL EXIT
++++
```

LINEEDIT uses the relative position of a card in the input file to specify what changes are to be made. The first card in the input file is card 0, the second card is card 1, etc. The above job step would cause the file MAIN to be updated as follows:-

The edit card #10 says 'copy from the current position in the file (i.e. card 0 in this case) up to and including card 10.'

Then a card containing

```
          PRINT 6
```

is added to the new version.

The edit card #14,17 would cause the cards from the current position in the old version (i.e. from card 11) up to and including card 13 to be copied to the new version of the file, after which cards 14 to 17 of the old version are read but not copied to the new version; i.e. cards 14 to 17 are deleted from the file.

Next a card containing

```
          CALL EXIT
```

is added to the new version.

Having now reached the end of the edit cards, the remaining cards, from the current position in the old version up to the end of the file, are copied into the new version and the old version is deleted.

Although LINEEDIT does not use the contents of the sequence field (i.e. columns 73 to 80 of a card image) when editing, it will by default insert in the updated file the number of each card in the card's sequence field. In this way a compilation listing can be used when updating a file.

THE OPTIONS CARD

The options card allows the user to pass control information to the editor; e.g. whether the output file is to be listed, whether sequence numbers should be inserted in the output records.

More than one options card can be used in a LINEEDIT run, allowing the user to alter the options in force dynamically. Thus he can selectively list parts of the output file, change RECLNTH to suit the information being edited, alter the control character, etc.

Format

The options and edit cards are read from the control file or the job source. The options cards are identified by the appearance of the control character in column one of the card. The same control character is also used on some edit cards.

This control character is defined to be the first character of the first card of the control stream. This implies that the first card is either an options card, or an edit card (other than a line insertion). In this way the user has complete freedom in choosing the control character. The control character can also be changed during a run of LINEEDIT.

The general format of an options card is:

<control char> <option 1>,<option 2>,...

where the control character must be punched in column 1. All blanks on the options card are ignored. There should be no comma after the last <option>, and at least one <option> should be specified.

Examples of the options card

```
# LIST,RECLNTH
#   NOEDITLIST,SEQUENCE
```

Options

The options which can be specified on the options cards are:

LIST, NOLIST — These two options control whether a listing of the edited file is produced. The default is NOLIST.

EDITLIST, NOEDITLIST — These two options control whether a listing is to be produced of the edit cards which specify the changes to be made. The default is EDITLIST.

SEQUENCE, NOSEQUENCE — These two options specify whether columns 73 to 80 of the output file are to contain the card numbers of the cards within the output file. If NOSEQUENCE is specified then the existing sequence numbers of the input cards are copied to the output file with no changes. The default is SEQUENCE.

RECLNTH = <integer>
where <integer> is an unsigned decimal integer of at most three digits. This option is used only when editing character strings within a card. If text is inserted into a card image, then provided that only blank characters are lost from the end of the card no error is reported. However, if non-blank characters are lost from the end of the card, the edit is assumed to be in error. When the input is sequenced, the insertion of any extra characters into a card would result in an error, as at least one non-blank character from the sequence field would be lost. If, however, we set RECLNTH = 72, then only the first 72 characters in the card are used and the remainder of the card image is treated as blank characters. The default value for RECLNTH is 72. RECLNTH can be set to any value between 1 and 80; however, note that it is only used when editing within a card image.

CONTROL = <character>
where <character> is any printable EBCDIC character. This option changes the control character to the character supplied. Note that the control character keeps its new value until the end of the run, unless it is explicitly changed again.

DATA — This option is used to simplify editing when all 80 columns of a card image contain significant data. Its presence is equivalent to the three options

```
RECLNTH=80,NOSEQUENCE,LIST
```

i.e. all 80 columns of a card image contain significant information, sequence numbers are not to be inserted in the output file, and the output file is to be listed.

If any of these options occurs more than once on an options card, then its last occurrence on the card determines the value taken for the option.

THE EDIT CARDS

Editing is carried out by transcribing the text from the specified input file to the output in accordance with the edit cards. The changes to be made are specified in terms of a card's position in the file being edited. The first card of the input text is card 0, the second card is card 1, etc.

LINEEDIT does not use the contents of the sequence field of a card to control the editing - it regards each card as 80 columns of data. The option SEQUENCE inserts the card number in the sequence field of the card solely to enable the file to be updated from, for example, a compilation listing.

The method of editing may best be understood by visualising a pointer which can be moved forward through the text to where an amendment is to be made. The pointer is initially set at the start of the first card (i.e. before card 0) and may be positioned anywhere within the text by the use of line directives, which move the pointer through the file line by line, and character directives, which position the pointer within a line.

After the last edit card has been obeyed, the remainder of the input text (i.e. from the current position of the pointer in the text) is transcribed to the output file.

The position of the pointer in the input text cannot be moved backwards. Hence if the pointer is after the card which is specified in a line directive, the editing is abandoned and the input file is not changed in any way. The same action is taken if any of the edit cards are faulty.

Line Editing

A line directive can take one of four possible formats, three of which require a control character to be in column one of the edit card.

a) <control char> n

where n is an unsigned integer constant. The directive is an instruction to transcribe from the current position in the input text, up to and including the card numbered n. The pointer in the input text is left at the start of the card numbered n+1.

b) <control char> n1, n2

where n1 and n2 are unsigned integer constants. The directive is an instruction to delete cards n1 to n2 from the text. This is achieved by transcribing from the current position in the input text until before the card numbered n1. The cards following in the input text up to and including n2 are then skipped. The pointer in the input text is therefore left at the start of the card numbered n2+1.

c) <control char> = <char>

This instruction changes the control character to the first non-blank character following the =. The control character keeps its new value for the rest of the run of LINEEDIT unless it is explicitly changed again later in the run. This instruction does not change the position of the pointer in any way, unless immediately following character directives (see below).

d) An edit card with no control character in column 1.

If the previous editing was a character directive, then the remainder of the current input card is first transcribed to the output file. The edit card is then written to the output file and thus any edit card with no control character in column 1 is inserted in the output file. The pointer in the input file is left unchanged.

When line editing, an edit card of format (a) is therefore used to specify where cards are to be inserted, and an edit card of format (b) is used to delete cards from the text. Any edit cards with no control character in column 1 which follow an edit card of format (b) will effectively replace the deleted card or cards.

Any blanks which appear in an edit card of format (a), (b) or (c) are ignored.

Character Directives

The line editing directives described above can, of course, be used to carry out any required changes to a file. However, it often happens that only one or two characters in a card require to be changed and to help in these cases certain character editing directives can be used. The character directives described below are used to change the text within a card image without retyping the whole card. To apply the character directives to a particular card, the pointer in the input text is positioned at the card by means of the line editing directives.

An edit card containing character directives has the form:

<control char> <char dir> , <char dir> ,..., <char dir>

where <char dir> takes one of the forms described below. There must be at least one <char dir> on the card and any <char dir> which is followed by another must be separated from it by a comma.

The four character directives are:

a) B/string1/string2/

This directive transcribes from the current position of the pointer until before the next occurrence of the character string string1, on the current card. String2 is then written to the output card. The position of the pointer in the input card is left immediately before string1. Hence string2 is inserted before string1.

b) A/string1/string2/

This directive transcribes from the current position of the pointer until after the next occurence of the character string string1, on the current card. String2 is then written to the output card. The pointer in the input card is positioned immediately after the string string1.

c) R/string1/string2/

This directive transcribes from the current position of the pointer until before the next occurrence of the character string string1, on the current card. String2 is then written to the output card. The position of the pointer in the input card is left immediately after the string string1. Hence string1 is replaced by string2. If string2 is null, i.e. contains no characters, then string1 is effectively removed from the card.

d) N

This directive transcribes from the current position of the pointer on the input card to the end of the current card, and the completed output card is then written to the output file. The pointer is placed at the start of the next record of the input file and a new card is started in the output file. Any following character directives will therefore apply to the new current input card.

Successive edit cards containing character directives will all apply to the current input card unless the directive N is used to move the pointer onto the next input card.

On reaching the end of the edit cards or an edit card which is a line editing command, LINEEDIT transcribes from the current position of the pointer on the input card to the end of the card, and the record is written to the output file.

If the first string (string1) is null, then the position of the pointer is not changed, but the second string (string2) is inserted. In this way, text can be inserted at the start of a card.

The string delimiter (/) used in the character directives above can in fact be any character from the set

: ; < = > ? ! # % & ' + /

Note that the delimiter must not occur in either of string1 or string2. Further, in a given character directive, each of the three occurrences of the delimiter must be the same character: for example, R:AB:DEX/ is invalid since the / would be taken to be part of string2.

On an edit card containing character directives, blanks are only significant between the string delimiters. When looking for a match of string1 on the card, blanks in string1 and in the card are significant.

If the first string is not found on the current card, an error is reported; i.e. only the current card is searched for the string.

If, as a result of the insertion of characters into a card, non-blank characters have to be truncated from the output card, then the edit is assumed to be in error. Note that if the output file is to be sequenced then the end of the output card is taken to be at column 72. However, if the output file is not to be sequenced, then the end of the output card is column 80.

Example:

The contents of a file called MAIN are listed below in the format used by LINEEDIT.

```
 0 *      INTEGER I,J,K,AM
 1 *      REAL A+L,M,N
 2 *      DO I = 1,10,-2
 3 *      READ (5,10) L,M
 4 *  10  FORMAT (215)
 5 *      N = L * M
 6 *      WRITE (6,12) L,M,N
 7 *  12  FORMAT (' ',I5,'+',I5,'=',I6)
 8 *  100 CONTINUE
 9 *      STOP
10 *      END
```

To update this file we have the edits shown below, where # is the control character:

```
#1,2
      REAL A,L,M,N
      DO 100 I = 1,10,2
#5,5
      N = L + M
      T = SQRT(N)
#7
      WRITE (6,9) L,M,T
    9 FORMAT (' SQRT(',I5,'+',I5,')',
     1  ' = ', F9.3)
```

The updated file is SEQUENCED by default, and so the updated file would contain:

```
      INTEGER I,J,K,AM                                          00000000
      REAL A,L,M,N                                              00000001
      DO 100 I = 1,10,2                                         00000002
      READ (5,10) L,M                                           00000003
   10 FORMAT (2I5)                                              00000004
      N = L + M                                                 00000005
      T = SQRT(N)                                               00000006
      WRITE (6,12) L,M,N                                        00000007
   12 FORMAT(' ',I5,'+',I5,'=',I6)                              00000008
      WRITE (6,9) L,M,T                                         00000009
    9 FORMAT (' SQRT(',I5,'+',I5,')',                           00000010
     1  ' = ', F9.3)                                           00000011
  100 CONTINUE                                                  00000012
      STOP                                                      00000013
      END                                                       00000014
```

Example:

In this example the same changes to the text of the file MAIN are effected as in the previous example but using the character directives to achieve some of the changes. Again assuming # as the control character, the necessary edits are:

```
#0
#R/+/,/,N,A/DO /100 /
#R/-//
#4
#R/*/+/
      T = SQRT(N)
#7
      WRITE(6,9) L,M,T
    9 FORMAT(' SQRT(',I5,'+',I5,')',
     1 ' = ', F9.3)
```

Notes:

* An edit card cannot contain both a character directive and a line directive.

* On the second edit card, the directive N causes the rest of the input line to be transferred to the output file and leaves the pointer in the input file at the start of the card which contains the DO statement.

* The space character is significant in the string 'DO ', so that the characters '100 ' will be separated by one space from the 'DO' on the output card.

* Although the directive R/-// appears in a separate edit card, the position of the pointer in the input text is unchanged.

* The directive #4 causes the remainder of card 2 to be output, and then the whole of cards 3 and 4 are written to the output file.

* The directive R/*/+/ consequently operates on card 5 and leaves the pointer before the character M.

* The card T = SQRT(N) is a line directive and as such causes the remainder of input card 5 to be output. Then the edit card itself is written to the output file and the pointer is left before input card 6.

RUNNING LINEEDIT

LINEEDIT is the name of a macro which contains the necessary job control statements to run LINEEDIT. The macro LINEEDIT has the following parameters:

INPUT=filename	where filename is the name of the file to be edited. This parameter has no default, and so must be given.
OUTPUT=filename	where filename is the name of the file to receive the edited text. If the file does not already exist it will be created. If the parameter is omitted, then a file with the same name as the input file will be created, but with a generation number one greater than the highest existing generation of the input file. In the latter case, if the edit is successful, the file with the generation one less than the output file will be deleted.
CONTROL=filename	where filename is the name of the file containing the control cards to be used in the run. If the parameter is omitted the control cards will be assumed to follow the macro call as alien data.
RESPONSE=SCLINTEGER	on return from the macro the SCLINTEGER will hold the result code issued by LINEEDIT; i.e. 0 - edit was successful 8 - errors detected If it is omitted the outer SCL variable RESULT will be used.

Examples:

To edit the file IN into the file OUT, the control cards being alien data:

```
LINEEDIT (INPUT=IN,OUTPUT=OUT)
----
#100
       GO TO 101
++++
```

To edit the file START, taking the control cards from the file EDITS:

```
LINEEDIT (INPUT=START,,CONTROL=EDITS)
```

In this case the updated file will be a file called START but with a generation number one greater than the generation number of the original file.

DIFFERENCES BETWEEN LINEEDIT ON 2900 AND ON IBM 370

* The method of running LINEEDIT is totally different
* Options cards replace operations cards, and you may have more than one in a run
* The first card of the control cards can be an options card or any edit card (except a line insertion)
* The options MEMBER, VERSION and TO do not apply
* The default options are: NOLIST, SEQUENCE, EDITLIST, RECLNTH=72

INDEX

2980 User's Guide - Comments Form

Notification of any errors, omissions, obscurities, and any other comments will be much appreciated. Please write them below and send to

P.E. Williams
ERCC
The King's Buildings
Mayfield Road
EDINBURGH EH9 3JZ