Oxford University Computing Laboratory

Computer Manuals

4159

SCIENTIFIC LANGUAGES

FORTRAN: Magnetic Tape Compiler

1900

ICL International Computers Limited

User notice

RECEIVED 18 NOV 1971

MANUAL (NOTICE NO.) 16/12/70

4149 FORTRAN: 32K DISC COMPILER (16)

4131 FORTRAN: 16K DISC COMPILER (12)

4159 FORTRAN MAGNETIC TAPE COMPILER (6)

O... COMPUTING LABOR...

Copy 1

File one copy of this notice with each of the manuals indicated.

4159

FORTRAN COMPILERS #XFAT, #XFEW, #XFAE AND #XFAM

If the mark 16 version of the SRF7 library is used with the above compilers, the following will apply

Errors corrected

1 Object programs using formatted disc files will no longer overwrite information on rewind (Software Notice 1900 Scientific Languages/33, item 63); this does not apply, of course, to #XFAM.

2 Extended mode programs may now be compiled using SRF7/16 subroutine group on M.L.T. 4/114; that is, Software Notice 1900 Scientific Languages/34, item 64 no longer applies.

© International Computers Limited, Reading, 1970

FORM 1/230/45(3.69)

Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire. RG1 1JP

International Computers Limited

ICL User notice

MANUAL (NOTICE NO.) 16/12/70

4149 FORTRAN: 32K DISC COMPILER (16)
4131 FORTRAN: 16K DISC COMPILER (12)
4159 FORTRAN MAGNETIC TAPE COMPILER (6)

Copy 2 4159

File one copy of this notice with each of the manuals indicated.

FORTRAN COMPILERS #XFAT, #XFEW, #XFAE AND #XFAM

If the mark 16 version of the SRF7 library is used with the above compilers, the following will apply

Errors corrected

1 Object programs using formatted disc files will no longer overwrite information on rewind (Software Notice 1900 Scientific Languages/33, item 63); this does not apply, of course, to #XFAM.

2 Extended mode programs may now be compiled using SRF7/16 subroutine group on M.L.T. 4/114; that is, Software Notice 1900 Scientific Languages/34, item 64 no longer applies.

© International Computers Limited, Reading, 1970

FORM 1/2300/45(3.69)

Technical Publications Service 20 - 21 Friar Street Reading Berkshire RG1 1JP

ICL International Computers Limited

User notice

RECEIVED

MANUAL (NOTICE NO.) 21/4/71

4131 FORTRAN: 16K DISC COMPILER (14)

4149 FORTRAN: 32K DISC COMPILER (18)

4159 FORTRAN MAGNETIC TAPE COMPILER (7)

File one copy of this notice with each of the manuals indicated.

OXFORD UNIVERSITY ... LABORATORY
Library
Copy 1 COMPUTING ... 4159.

FORTRAN COMPILERS #XFAT, #XFEW, #XFAE AND #XFAM

The following information applies when these compilers are used with the mark 17 version of the SRF7 library, or, in the case of #XFEW, the mark 2 version of the SRF8 library.

Errors corrected

1 The mathematical routine INT now gives an accuracy up to 8 significant decimal places when the argument is negative. (Software notice 1900 Scientific Languages/30 item 58(1))

2 #XFAT and #XFEW only: Incorrect results will no longer occur when the G format field descriptor is used with a repeat count. (Software Notice 1900 Scientific Languages/30 item 59)

3 #XFAT and #XFEW only: Execution error 128 is no longer incorrectly flagged by the NAMELIST input routine if the single character 0 appears in the data as any constant after the first in a set of constants assigned to an integer array. (Software Notice 1900 Scientific Languages/36 item 69)

4 #XFEW only: When hardware extended precision object code is compiled (1906A only), exponentiation of a DOUBLE PRECISION item to an INTEGER power will no longer produce incorrect results when the absolute value of the exponent is greater than 3. (Software Notice 1900 Scientific Languages/46 item 90)

Changed execution error

Execution error 106 is no longer fatal.

© International Computers Limited, Reading, 1971

FORM 1/250/45(3.69)

Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire. RG1 1JP

ICL International Computers Limited

RECEIVED NOV 1971

User notice

MANUAL (NOTICE NO.) 23/6/71

4159 FORTRAN MAGNETIC TAPE COMPILER (8)

4131 FORTRAN: 16K DISC COMPILER (15)

Copy 1 4159

File one copy of this notice with each of the manuals indicated.

FORTRAN COMPILER LIBRARY SRF7/18

The mark 18 version of the SRF7 library corrects errors reported in Software Notices:

1900/Scientific Languages/41, item 82
1900/Scientific Languages/42, item 84
1900/Scientific Languages/30, item 58(2)

© International Computers Limited, Reading 1971

FORM 1/230/45(3.69)

Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire. RG1 1JP

ICL International Computers Limited

User notice

L. RECEIVED 20 DEC 1972

MANUAL (NOTICE NO.)

6/12/72

4159 FORTRAN: MAGNETIC TAPE COMPILER (9)

File one copy of this notice with each of the manuals indicated.

FORM 1/230/45(3.69)

FORTRAN COMPILER #XFAM

When a program is compiled and consolidated by #XFAM, there are certain restrictions on the incorporation of semicompiled segments:

1 Segments produced by the FORTRAN compilers #XFAT, #XFEW, #XFIV, #XFEV, #XFIH and #XFEH may not be included.

2 Segments containing top common (elastic) data areas may not be included. Apart from #XALM, Algol compilers produce such a data area for the stack. #XALM also produces such an area unless the program description statement 'SPACE' is used in the Algol compilation. In this case, however, the size of stack specified in the 'SPACE' statement must be sufficient to ensure that the stack does not need to be extended at run time (3000 words should be sufficient in most cases).

ERRORS IN MANUAL

Certain errors have been discovered in the manual, as detailed below.

Page 10

The description of the default block size assumed when no size is specified in a channel description statement is inaccurate. For a FORMATTED file, a default block size of 128 words is used. For an UNFORMATTED file, the size used is the largest block or buffer size so far encountered for an UNFORMATTED magnetic tape or disc file, unless this is less than 128 words in which case 128 words is used.

Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire. RG1 1JP

Page 49

If output from #XMUM is to magnetic tape, the program subfile is given the name PROG*xxxxxxxx* where *xxxxxxxx* is the eight character name used in the job description statement.

Pages 75 and 76

The section *Compiling mixed FORTRAN/PLAN programs* incorrectly states that segments compiled by the PLAN compiler can be input to the FORTRAN compiler by means of a statement:

READ FROM (MT,*filename*)

In fact a subfile must be specified, and the form of the READFROM statement is as described on page 22. The subfile name should be either PROGRAM *name* or PROGRAM XXXX, depending on which of the following was the form of the #PROGRAM line in the PLAN compilation:

name/segment
/segment

For full details, the manual *PLAN Reference Manual* should be consulted.

FORTRAN SEGMENTS IN NON-FORTRAN PROGRAMS

FORTRAN segments to be included in a program with a non-FORTRAN MASTER segment should be compiled at trace level 0. Segments compiled at trace level 1 may also be included, but trace 1 facilities will only be obtained if the segments include a READ or WRITE statement. The effect of including a FORTRAN segment compiled at trace level 2 in a program with a non-FORTRAN MASTER segment is not defined.

TEXT CONSTANTS

The maximum number of characters that can be specified in a text constant is 511.

OBJECT PROGRAM ERROR HALTS

In addition to the object program halt messages given on page 90, programs using disc files may halt with the message

HALTED:- SZ

This occurs if a disc logical bucket number is out of range or zero and can happen if an attempt is made to read from a scratch file before it has been written to.

Extension of Scratch Disc Files in FORTRAN programs

If a FORTRAN program opens a scratch disc file and further space is required, the FORTRAN library routines will attempt to make an appropriate extension. If there is no space available on the same cartridge as the scratch file an extension onto any other suitable online cartridge will be made if the program is not run under UDAS. If UDAS executive is used, or if the scratch file is an online exofile under GEORGE 3, then an extension will not be made to another cartridge.

In either case, if more space cannot be allocated, the program will halt

NEED ED SCRATCH SPACE

or, if the scratch file is on fixed disc not under UDAS

NEED FD SCRATCH SPACE

If this occurs when UDAS is not used, the program may be restarted once a further cartridge with space available has been put online. Under UDAS executive (or GEORGE 3), the run must be repeated with a disc containing more spare space as the first online.

Execution errors

The following execution error numbers can occur. These are additional to those listed in Appendix 2 and all are fatal.

109 Error in transfer on card input channel.
110 Error in transfer on paper tape input channel.
111 Error in transfer on magnetic tape input.
112 Error in transfer on disc (E.D.S. or F.D.S.) input.

ARRAY ELEMENT BOUND CHECK

The following information amplifies that given on page 31 concerning the action at trace level 2 when an array element falls outside the bounds of the array.

For an array A, declared as $A(D_1,D_2,...D_n)$, element $A(d_1,d_2,...d_n)$ is held in the position (relative to the beginning of the array) specified by the formula:

$$d_1+D_1(d_2-1)+D_1D_2(d_3-1)+...+D_1D_2...D_{n-1}(d_n-1)$$

The trace 2 check only verifies that an element is within the total area allotted to the array by checking that the position calculated from the above formula lies between 1 and $D_1D_2...D_n$ inclusive. Thus, in the

example given on page 31 of an array declared as ARRAY(3,2), a reference to ARRAY(4,1) is not faulted. The position calculated by the formula above is 4+3×(1-1) or 4, which is less than 3×2 or 6, the array size.

In some cases, an array element reference where the product of the subscript expressions exceeds the size of the array is not faulted. For example, if an array is declared as:

```
DIMENSION  B(3,4)
```

a reference to B(5,3) will not be faulted: this element is considered to be held in position 11, within the bounds of the array.

© International Computers Limited, Reading 1972

International Computers Limited

User notice

L

RECEIVED 10 AUG 1973

PUBLICATION (NOTICE NO.)

1/8/73

4159 FORTRAN MAGNETIC TAPE COMPILER (10)

File one copy of this notice with each of the publications indicated.

FORTRAN COMPILERS #XFAM, #XFAE AND #XFAT

The following restrictions apply to use of the EQUIVALENCE statement in programs compiled by the above compilers:

1. A variable that is assigned a value in a DATA statement cannot be equivalenced to an array element.

2. An array may not be equivalenced to a shorter, non-common array that is wholly or partially initialised in a DATA statement.

3. An item appearing in a DATA statement may not be equivalenced to another item that appears in a DATA statement. An array or array element is considered to appear in a DATA statement if any element of the array is initialised in a DATA statement.

FORTRAN COMPILERS #XFAM, #XFAE, #XFAT, #XFEW, #XFEV, #XFIV, #XFEH AND #XFIH

Formatted input records in FORTRAN object programs

The following rule should be observed when inputting formatted records:

If more than one sign appears before a number on the input record, the sign of the number is taken as that specified by the final plus or minus sign (that is, the one at the right hand end of the + - string) Hence + - 5 will have the internal representation of - 5, and - + 5 will have the internal representation of + 5. The additional signs will not be flagged as incorrect characters.

© International Computers Limited, Reading 1973

Issued by Software Distribution Department, 30 - 31 Friar Street, Reading, Berkshire RG1 1JP

International Computers Limited

User notice

PUBLICATION (NOTICE NO.) 7/1/76

4300	FORTRAN: 16K DISC COMPILER (3)
4159	FORTRAN: MAGNETIC TAPE COMPILER (11)

File one copy of this notice with each of the publications indicated

FORTRAN COMPILER #XFAE

This notice announces the issue of a new version of the compiler #XFAE, which is intended for use with versions of the library SRF7 from Mark 23 onwards. The compiler has the mark number 5A and requires a minimum core size of 10624 words.

Errors corrected

The new issue corrects the errors reported in Scientific Languages Software Notices:

153/340, 156/343 and 209/2

In addition, a number of minor errors have been corrected.

DIRECT ACCESS FACILITIES

The following paragraphs describe new facilities which are now available in FORTRAN programs compiled by #XFAE/5A for the direct access of files on disc. These facilities are compatible with those already available in other 1900 compilers.

General information

Files may be processed serially or by direct access. In the former case the records are handled in the order in which they occur on the file, while in the latter case operations may be performed on selected records in any order.

Files specified to be processed by direct access (see 'The DEFINE FILE statement' below) may not be processed serially, and vice versa.

DIRECT ACCESS CHANNEL DESCRIPTION STATEMENTS

The format for a channel description statement specifying a direct access disc file is as follows:

hk=*xn*/DIRECT (*f* (*g*))/*b*

where

h is one of CREATE, INPUT, OUTPUT or USE, which are interpreted as follows:

INPUT - use the named disc file which has been opened and written to in a previous program as a direct access disc file for input only. (See section Programming Techniques, item 3,page 5)

OUTPUT - use the named disc file (assumed to contain no useful information) for output only.

CREATE - use the named disc file (assumed to contain no useful information) for output followed by input if required.

Issued by Software Distribution Department,30-31 Friar Street,Reading,Berkshire RG1 1JP

USE - 1 with a permanent file: use the named disc file which has been opened and written to in a previous program as a direct access disc file for input and/or output. (See section Programming Techniques, item 3, page 5)

2 with a scratch file: use a scratch file for input and/or output.

k is the channel number or list of channel numbers by which the file is referenced in the FORTRAN program.

x is either ED or DA

n is the logical unit number for the disc file

f is either the file name or a dummy file name (see 'File Names' below).

g is the generation number (optional)
if (*g*) is omitted, the file of name *f* of highest generation number will be used.

b is the size in words of the buffer to be associated with the file (optional: see 'Buffer Size' below).

File names

f is either the name of a file or a dummy file name which will be replaced at run time by the actual file name (see the ICL 1900 Series manual *FORTRAN: 16K Disc Compiler*, TP 4300 page 14 section 'File names at run time'). All files to be used must already have been allocated using the file allocator program XPJC, or created using the GEORGE 3 and 4 command CREATE.

If the option USE or CREATE was specified in the program description, *f* and its enclosing parentheses may be omitted. In this case a scratch file will be created and will cease to exist when the program is deleted.

Buffer size

For optimum use of both main store and space on the disc, the value of *b* should be the same as the bucket size on the file.

The bucket size is defined when the file is allocated and may be 128, 256, 512 or 1024 words. If main store is at a premium, a buffer size less than the bucket size may be used for output files: this will reduce the object program store size but will be wasteful of space on the disc. The larger the value of *b*, the more main store will be occupied and in general the more efficient will be the object program. *b* should not be greater than the bucket size as only the first *n* words in the buffer (where *n* is the bucket size) would be used. If /*b* or *b* is omitted, or *b* is zero, a buffer size of at least 128 words is assumed.

Direct access records, both formatted and unformatted, are held one or more to a bucket and may be up to *b*-4 words long.

File extension

If the file specified as a direct access file is not large enough to hold all the records an attempt will be made to extend the file.

This extension will be permitted only on the cartridge on which the file lies. Such extensions will be reported on the monitor output channel when the file is released (as for serial files).

It should be remembered that when calculating the size of the file required, one word must be added to the record size, as given in the DEFINE FILE statement, for the word count. Also, an extra bucket must be added for the end-of-file bucket.

Examples

1 CREATE 3 = ED3/DIRECT (FILE ONE)

Channel number 3 in the program corresponds to the direct access file FILE ONE to be allocated as ED3

2 USE 4 = DA1/DIRECT/256

Channel 4 in the program corresponds to a direct access scratch file with logical unit number 1. 256 word buckets will be used

DIRECT ACCESS File usage

Whereas a READ or WRITE statement referencing a serial file will cause input or output of the next record in sequence following that input or output by the last READ or WRITE statement, for a direct access file, records within the file may be accessed in any order. Each record within the file has a unique number associated with it, the records being numbered from one, and a record identified by specifying this number either as a constant or as a variable. For the appropriate forms of the READ and WRITE statements for direct access files see 'Input and Output for Direct Access Files' below.

The DEFINE FILE Statement

The DEFINE FILE statement is a specification statement and must appear before any executable statement. It describes the characteristics of any direct access file to be used. In order to use the direct access I/O operations each file must be described at least once by a DEFINE FILE statement as well as in the program description.

At run time the first DEFINE FILE statement encountered for one particular channel will set up the specification for that direct access file. Any subsequent statements encountered for that same channel will be ignored.

The DEFINE FILE statement has the form:

DEFINE FILE k_1 (m_1, x_1, t_1, v_1), k_2 (m_2,x_2,t_2,v_2),...,k_n (m_n,x_n,t_n,v_n)

where

k is an INTEGER constant identifying a particular channel.

m is an INTEGER constant giving the number of records in the direct access file associated with channel number *k*.

x is an INTEGER constant giving the size of each record in the direct access file associated with channel number *k*. The units in which the size is measured depends on *t*.

t is one of the following characters:

- U indicating that the file will contain only unformatted records. The record size will be measured in elements of two words.
- E indicating that the file will contain only formatted records. The record size will be measured in characters.
- L indicating that the file may contain both formatted and unformatted records. The record size is measured in half words.

v is an INTEGER variable, the associated variable (see below), which may not be a dummy argument of a subroutine or function segment, but may be an item in a common block.

In any master, function or subroutine segment, DEFINE FILE statements must occupy a position between statement function definitions and executable statements. See the manual *FORTRAN* TP 4261 page 40 Section 'Statement ordering'.

Example

The statement:

DEFINE FILE 4(50,100,L,I),6(10,50,E,J), 1(40,20,I,K)

defines three direct access files. The first file, referred to as channel 4, consists of 50 records, of length 100 half words, that are to be transferred either with or without format control, the associated variable being I. The second file, referred to as channel 6, consists of 10 records, of length 50 characters, that are to be transferred with format control, the associated variable being J. The third file referred to as channel 1, consists of 40 records, of length 20 elements each of 2 words, that are to be transferred without format control, the associated

variable being K.

Input and Output for Direct Access Files

The appropriate forms of the READ and WRITE statements for transfer of direct access records are:

READ (*k*'*r*,*l*) list
WRITE (*k*',*r*,*l*) list

where

k is an integer constant or an integer variable representing the channel number. *k* must be followed by an apostrophe.

r is an integer expression which may or may not contain the associated variable, representing the record number.

l is present for formatted, but not for unformatted, records. If present, it is either the label of a FORMAT statement or the name of an array containing the format information.

'list' is an I/O list which is optional for READ or formatted WRITE statements.

Execution of either the READ or WRITE statement transfers the items in the 'list', in order, to the direct access file associated with *k*, starting at record number *r* of the file. (See Section Programming Techniques item 1 on page 5 for actual number of records transferred for each READ or WRITE.)

The FIND Statement

FIND statements are used to find the next record to be used in direct access file while the present record is being processed. Thus, the amount of time taken by a subsequent READ statement to that record is reduced. The FIND statement has the form:

FIND (*k*'*r*)

where *k* is an INTEGER variable identifying a particular channel number; *k* must be followed by an apostrophe. *r* is an INTEGER expression giving the number of the record to be found.

The file associated with channel number *k* must have been defined in a DEFINE FILE statement.

The FIND statement will find the record *r* in the direct access file associated with channel number *k*. The record will not be made available until a READ statement referring to that record is encountered. No advantage is gained in using the FIND statement before a WRITE statement.

An example of the use of this statement follows:

```
  DEFINE FILE 6(100,60,E,I)
  ...
  READ (6'1,9)A,B,C
  FIND (6'2)
9 FORMAT (3F12.8)
  VAL=SQRT (A*A+B*B+C*C)
  ...
  READ (6'2,9)A,B,C
```

If these statements were included in a program, the FIND statement will cause record 2 of the direct access file associated with channel number 6 to be found whilst the data from record 1 is being processed. The second READ statement then makes record 2 available.

The Associated Variable

At the start of execution of the program, the associated variable of each direct access file defined in a DEFINE FILE statement is assigned the value 1, corresponding to the first record of the file.

The execution of any READ, WRITE or FIND statement referencing the file causes the value of the associated variable to be reset. A READ or WRITE statement will cause the associated variable to be set to the number of the record after the last record accessed by the READ or WRITE statement.

A FIND statement sets the associate variable to the record number specified in the FIND statement.

The expression *r* in any of the statements:

READ (*k*'*r*...
WRITE (*k*'*r*...
FIND (*k*'*r*)

may include the associated variable. If for a particular file *k* each READ and WRITE statement specifies the associated variable as the expression *r* the records will be accessed serially. Thus a direct access file can be used as though it were a serial file.

If the current value of an associated variable is to be transferred between segments this must be done by specifying the associated variable as an item in a common block. Associated variables may not be used as dummy or actual arguments.

Auxiliary I/O Operations and I/O Subroutines

1 BACKSPACE, ENDFILE, and REWIND have no effect on a direct access file

2 The subroutines FILE, RENAME, ALLOT, RUNOUT and RELEASE are available as for serial disc files

PROGRAMMING TECHNIQUES

1 Number of records transferred.

If the records concerned are formatted, input and output are controlled by the input/output list and the format specification as described in the section 'Effect of FORMAT statements and arrays' on page 43 of the ICL 1900 Series manual *FORTRAN*, Edition 2, 1971, TP 4261. The record lengths implied by the format specification must not be greater than the record size given in the DEFINE FILE statement. If the record size is greater than the length of a particular record as specified in a format specification for a WRITE statement, the record will be space filled.

If the records are unformatted, the number of records transferred is determined by the input/output list: transfer is complete when all items listed have had their values input or output, and as many records will be transferred as are required. A WRITE statement will cause any spare space in the last record to be zero filled

2 The DEFINE FILE statement for a direct access file does not have to be in the same segment in which the I/O operations occur. For example the DEFINE FILE statement can be given in the master segment with a subroutine performing the I/O operations. The associated variable will still be updated by the I/O operations even though it may not be stated explicitly in the subroutine

3 The following information is included to clarify the use of Execution Error 92 (see below):

When first accessed, any direct access file declared as OUTPUT or CREATE is opened and skeleton records, together with an end of file bucket, are written to the disc to correspond to the maximum number of records required.

When first accessed, any direct access file declared as INPUT or USE is opened and the first bucket is read. This is to ensure that the record word count which the first record should already contain corresponds to that calculated from the DEFINE FILE statement. Hence any direct access file declared in either of the program description statements INPUT or USE must already have been written to by a previous program. Execution Error 92 may be reported if the record word length of this file is not the same as the record word length specified in the DEFINE FILE statement

4 Example of direct access file handling

```
      DEFINE FILE 1 (1000,72,E,J)
      DIMENSION A(100),B(100),C(100),D(100),E(100),F(100)
      ...
 1    FORMAT (6F12.4)
      FIND   (1'5)
      ...
      DO 2 I = 1,100
      READ (1'J,1)A(I),B(I),C(I),D(I),E(I),F(I)
      FIND (1'J+4)
      ...
```

```
2     CONTINUE
      J = J-4
      DO 3 I = 1,100
3     WRITE (1'J+4,1)A(I),B(I),C(I),D(I),E(I),F(I)
```

This example illustrates the ability of direct access statements to read and write data in an order specified by the user. The first loop fills arrays A to F with data from the 5th, 10th, 15th ... 500th record of the file with channel number 1. Array A receives the first value in every record, B the second and so on, as specified by the FORMAT statement 1 and the input list of the READ statement. The first time the READ statement is executed, J is reset to 6; after the second execution, J is set to 11, and so on. At the conclusion of the first DO loop J has the value 505.
J is then reset to 501.
The second DO loop groups the same elements from each array as specified by the output list of the WRITE statement and the FORMAT statement 1. Each group of elements is written to the file with channel number 1, starting at the 505th record and continuing at intervals of five until record 1000 is written.

ERROR MESSAGES

Compilation Errors in Direct Access Input/Output Statements

Compilation error numbers have the same significance as in XFAE Mark 4D (see page 123 of the ICL 1900 Series manual *FORTRAN: 16K Disc Compiler* Edition 2, 1971, TP 4300) with the following additions:

26 DEFINE FILE channel number is not small integer

27 DEFINE FILE Non-integer numbers of records or record size
FIND Apostrophe expected

28 DEFINE FILE Format specification not recognised

29 Direct Access READ / Direct Access WRITE / FIND } Integer record number expected

25 Amended to mean 'incorrect characters found'.

Execution Errors in Direct Access Input/Output Statements

Additions:

85 Channel number in DEFINE FILE statement does not refer to a direct access file.

86 File not large enough (even with extension) to hold all the records defined in the DEFINE FILE statement.

87 Record length in the DEFINE FILE statement larger than the minimum of the buffer and bucket sizes.

88 Relative position of a record is not a positive integer or exceeds the number of records in the data set.

89 Formatted record too long.

90 No previous DEFINE FILE statement.

91 Direct access file opened outside system.

92 DEFINE FILE statement does not correspond with previously created file.

Amendments:

23 An attempt has been made to read or write
(1) serially to a direct access file
(2) unformatted to a file described as E
(3) formatted to a file described as U

Other errors and console halts as for serial I/O operations.

RESTRICTIONS PRESENT IN #XFAE

It should be noted that an item declared in a COMMON statement may not be equivalenced to another item which has also been declared in a COMMON statement.

The following restrictions apply to use of the DATA statement compiled by XFAE:

1 Hollerith strings of less than 5 characters assigned to a real variable, or less than 13 characters assigned to a double precision or complex variable, should be space-filled to at least 5 or 13 significant characters respectively to ensure that any remaining character positions allocated to the variable are space-filled by the compiler

2 The number of characters in a Hollerith string must not exceed the size specified by the variable, array or array element name to which it has been assigned

LIBRARY SUBGROUPSRF7

The following changes, which apply to the execution error messages, take effect with release of FORTRAN subroutine library routines RENAME/9, FILE/8 and %FINEDS/15.

1 Execution Error 28 has been redefined and the interpretation is now as follows:

   An attempt has been made to use FILE or RENAME but a dummy file name has not been given in the Program Description, or

   The new file name does not commence with an alpha character.

2 The message HALTED:NR may now occur once Execution Error 41, 42, 43 or 46 has been reported. This will signify that the program may not have released all its peripherals and hence any output from previous WRITE statements need not necessarily have been transferred to the appropriate media

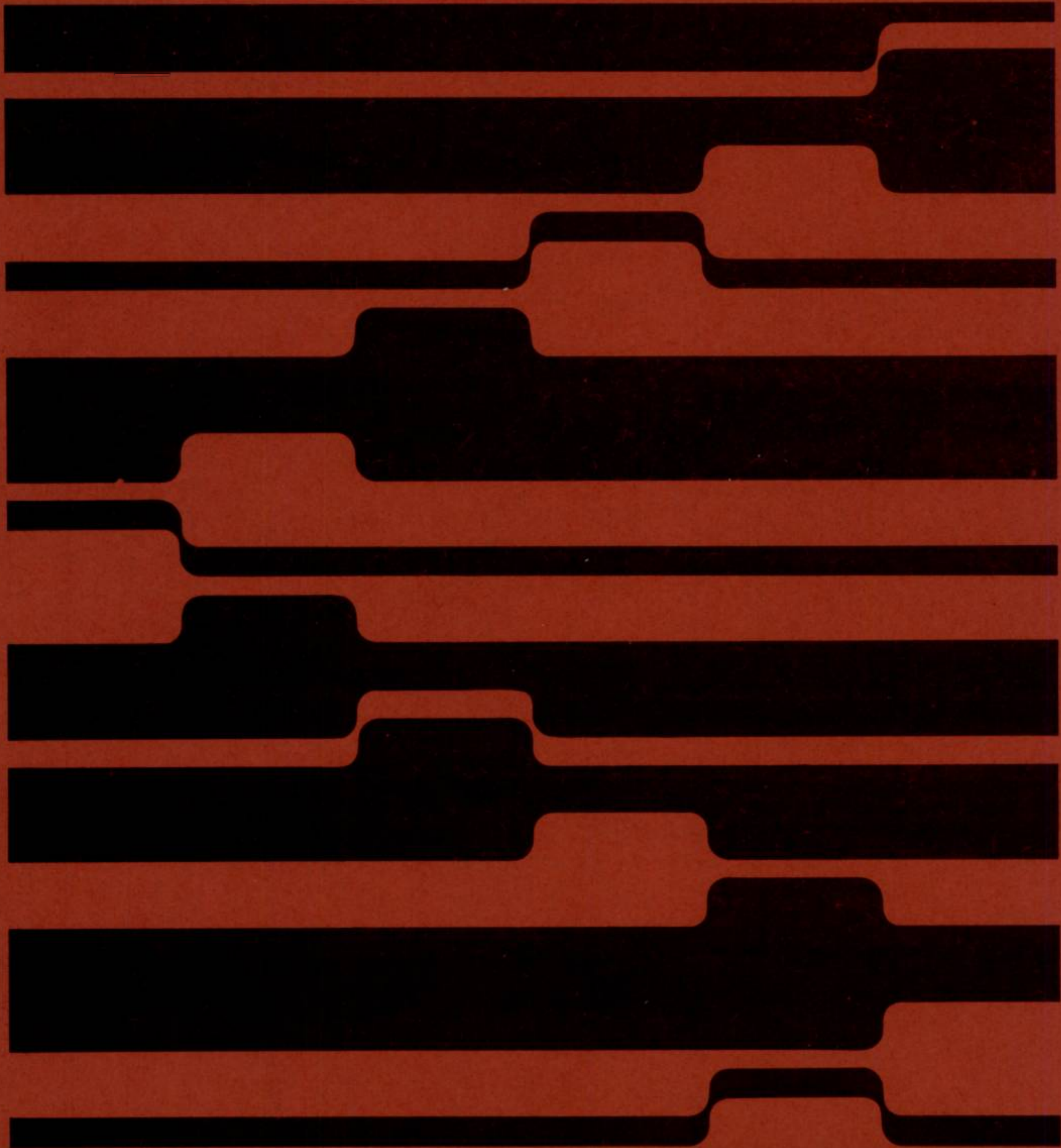
© International Computers Limited, 1975

ICL

FORTRAN Magnetic Tape Compiler

1900 Series

OXFORD UNIVERSITY COMPUTING LABORATORY
Copy 1
COMPUTING SERVICE 4159



ICL

FORTRAN: Magnetic Tape Compiler

1900 Series

OXFORD UNIVERSITY COMPUTING LABORATORY
Copy 1 COMPUTING SERVICE 4159

A

The policy of International Computers Limited is one of continuous development and improvement of its products and services, and the right is therefore reserved to alter the information contained in this document without notice. ICL makes every endeavour to ensure the accuracy of the contents of this document but does not accept liability for any error or omission. Any equipment on software performance figures and times stated herein are those which ICL expects to be achieved in normal circumstances. Wherever practicable, ICL is willing to verify upon request the accuracy of any specific matter contained in this document.

Technical Publication 4159

© International Computers Limited 1969

First Edition July 1969

Issued by Technical Publications Service
International Computers Limited
Head Office: ICL House, Putney, London SW15
and printed in Great Britain by
ICL Printing Services, Letchworth, Hertfordshire

Preface

OXFORD UNIVERSITY COMPUTING LABORATORY
Copy 1. COMPUTING SERVICE 4159

This manual describes the preparation of programs for compilation by the magnetic tape version of the ICL 1900 Series FORTRAN compiler #XFAM and explains the operation of this compiling system. Part 1 of the manual is intended for programmers who already have a knowledge of 1900 FORTRAN as described in the 1900 Series manual *FORTRAN* (TP 4088). Part 2 is concerned with operating considerations.

Chapter 1 explains the use of compiling system statements which enable the programmer to control the course of a compilation and make the best use of the facilities provided by the compiler. The general use of magnetic tape files is described in Chapter 2. The individual compiling statements are described in detail in Chapters 3 to 9. The use of the editing programs #XMUM and #XKYA to edit magnetic tape files is described in Chapter 10. Chapter 11 describes how segments written in PLAN may be incorporated into FORTRAN programs and vice versa. Chapter 12 describes the use of the compiler under the GEORGE 1 and 2 operating systems. Chapter 13 is designed primarily for those responsible for the running of an installation and consists of a system description with particular reference to magnetic tape file organization. Chapter 14 describes how several programs may be batched together and compiled in one run of the FORTRAN compiler. Chapter 15 gives the operating instructions and exception conditions.

The minimum 1900 Series configuration recommended for running the FORTRAN magnetic tape compiling system is as follows:

> A 1900 Series central processor with 16K words of core store and floating point facilities (hardware or extracode).
>
> 4 magnetic tape decks (but see below).
>
> 1 paper tape reader or card reader.
>
> 1 line printer.

One tape deck may be omitted if the source program is not to be held on magnetic tape. One tape deck may be omitted if the object program will never be overlaid and will never exceed 32768 words in size.

Further information relevant to the writing and compilation of FORTRAN programs is given in the 1900 Series manual *FORTRAN: Compiler Libraries* (TP 4170), which describes the various standard functions and subroutines available to FORTRAN programs.

Contents

Preface iii

Introduction xv

1900 SERIES COMPUTERS xv

1900 SERIES COMPILERS xv

THE FORTRAN MAGNETIC TAPE SYSTEM xv

PART 1 PROGRAMMING CONSIDERATIONS

Chapter 1 Program structure 1

COMPILING SYSTEM STATEMENTS 1

Initial statements 1

Program description statements 1

Intersegment statements 2

Terminator 2

Input sequence 2

Format of compiling system statements 2

Batch compilations 2

Chapter 2 Magnetic tape files 5

SIMPLE FILES, COMPOSITE FILES AND SUBFILES 5

Named files and scratch files 5

File and subfile names 5

Generation number 5

Retention period 6

Write permit ring 6

FORTRAN magnetic tape formats 6

Chapter 3 Object program input/output 7

CHANNEL DESCRIPTION STATEMENTS 7

BASIC PERIPHERALS 8

Double buffering 8

MAGNETIC TAPE FILES 8

Effect of source program statements for magnetic tape files 8

Opening of magnetic tape files 9

Magnetic tape channel description statements 9

File names at run time 10

File renaming 11

Printing of formatted magnetic tape files 11
END OF FILE MARKERS 11
CHANNEL 0 12
CONSOLE TYPEWRITER 12
CHARACTER TRANSFERS IN CORE STORE 12
SUPPRESSION OF TRANSFERS 13

Chapter 4 Compiler listing 15
LISTING DESCRIPTION 15
Full listing 15
Short listing 15
LISTING STATEMENTS 15

Chapter 5 Compiler output 17
TYPES OF OUTPUT 17
Load and go 17
Binary program 17
Semi-compiled segments 17
Consolidated semi-compiled program 17
INITIAL STATEMENTS 17
The DUMP ON statement 17
The RUN statement 18
The SEND TO statement for semi-compiled segments 18
The SEND TO statement for consolidated semi-compiled program 19
COMPILATIONS WITH ERRORS 19
SUPPRESSION OF COMPILER OUTPUT 20

Chapter 6 Compiler input 21
COMPILER INPUT MEDIA 21
Initial input medium 21
COMPILING FROM BASIC PERIPHERALS 21
COMPILING FROM MAGNETIC TAPE 22
Input from subfiles 22
Compiling system statements on magnetic tape 23
Input from simple files 23
Editing of source programs on magnetic tape 23
INCORPORATION OF LIBRARY SUBROUTINES INTO PROGRAM 24
FORTRAN compiler library 24
Other system libraries 24
Libraries held on other tapes 25
Paper tape and card libraries 25
Recommended order of searching libraries 25
ACCEPTANCE RULES 26

Chapter 7 Program testing	27
COMPILATION ERROR DIAGNOSTICS	27
Error messages	27
LABEL ERRORS	27
MISSING SEGMENTS	27
OBJECT PROGRAM ERROR DIAGNOSTICS	27
Trace output channel	28
Trace level 1	28
NORMAL ERROR ACTION	28
OVERFLOW	28
THE ERROR TRAP FACILITY	29
Trace level 2	30
ERROR ACTION	30
ARRAY ELEMENTS	31
TRACE 2 WITH STEERING LISTS	31
TRACING EVERY STATEMENT	32
Suppression of trace output	32
Trace level 0	33
Mixed error detection levels	33
Operator intervention	33
Example of error detection in a program with an arithmetic error	33
Example of error trap facility	35
Chapter 8 Overlays	37
THE OVERLAY SYSTEM	37
PROGRAM DESCRIPTION STATEMENTS FOR OVERLAY PROGRAMS	37
The OVERLAY statement	37
The OVERLAY PROGRAM and OVERLAY SEGMENTS statements	38
The DEPTH OF OVERLAY statement	38
Restrictions	39
COMPILATION	39
RUN TIME EFFICIENCY	39
STANDARD FUNCTIONS IN OVERLAYS	39
EXAMPLE OF AN OVERLAY PROGRAM	39
OVERLAYING FROM DISC	40
Chapter 9 Miscellaneous features	43
COMPACT AND EXTENDED DATA PROGRAMS	43
Compiling and running on different size processors	43
Mixed language programs	43
PRIORITY	44

THE OMIT STATEMENT 44
THE COMPRESS STATEMENTS 44
INTEGER AND LOGICAL statements 44
DOUBLE PRECISION statements 45

Chapter 10 Editing 47

THE EDITOR PROGRAM #XMUM 47
Input 48
Summary of facilities 48
THE RESTART FACILITY 49
Output 49
LINE PRINTER OUTPUT 49
Directives 51
BATCH DIRECTIVES 51
JOB DIRECTIVES 52
THE EDITOR PROGRAM #XKYA 57
General description 57
File formats 57
Summary of editor instructions 57
MAIN EDITOR 57
Opening the output file 57
Creating new subfiles 57
Opening input files 57
Positioning input files 57
Transferring subfiles 57
Introducing the line editor 57
Monitoring 58
Closing the files 58
Terminating the run 58
Miscellaneous instructions 58
LINE EDITOR 58
Writing 58
Monitoring 58
Returning to the main editor 58
Reference section 58
INSTRUCTIONS TO THE MAIN EDITOR 58
INSTRUCTIONS TO THE LINE EDITOR 61

Chapter 11 Mixed language programming 63

COMMUNICATION OF DATA 63
TRANSFER OF CONTROL 63
CALLING PLAN SEGMENTS FROM FORTRAN 63
The FPROLOG subroutine 64
The FEPILOG subroutine 65

Arrays 66
THE GETAH SUBROUTINE 66
THE GENAH SUBROUTINE 67
Common storage areas 68
BLOCK DATA SEGMENTS 70
BLANK COMMON AREAS 70
CALLING FORTRAN SEGMENTS FROM PLAN 72
MIXED PLAN/FORTRAN OVERLAID PROGRAMS 73
The subroutine FOVER 73
Restrictions 74
ENTRY POINTS AND MASTER SEGMENTS 75
USE OF WORD 30 75
PERIPHERAL USAGE 75
SUBPROGRAMMING 75
COMPILING MIXED FORTRAN/PLAN PROGRAMS 75
THE LEADER STATEMENT 76

Chapter 12 GEORGE 1 and 2 77

PART 2 OPERATING CONSIDERATIONS

Chapter 13 System description 79
THE 1900 SERIES FORTRAN MAGNETIC TAPE SYSTEM 79
Compilation and consolidation 79
SEGMENTS MODE 79
SINGLE PROGRAM MODE 79
BATCH MODE 79
Object programs 79
TAPE USAGE AND EFFICIENCY 79
Compile time 79
Object time 80
OPTIONAL SYSTEM ALTERATIONS 80
Log analysis message 80
Default listing mode 80

Chapter 14 Batch compilation 81
WORK ASSEMBLY 81
BATCH SYSTEM STATEMENTS 81
Initial statements 82
The terminating statement 83
BATCH INPUT MEDIA 83
Paper tape input 83
Punched card input 83
Switching between basic peripherals 83
Magnetic tape input 83

Magnetic tape output 84
EFFECT OF COMPILING SYSTEM STATEMENTS 84
Listing statements 84
SEND TO statements 84
RUN and DUMP ON statements 84
SEGMENTS statements 84

Chapter 15 Operator's guide to the FORTRAN magnetic tape compiler 85
OPERATORS INSTRUCTIONS FOR THE MAGNETIC TAPE COMPILER 85
Hardware requirement 85
Use of peripherals 85
Priority 85
Operating instructions 86
Exception conditions 88
OPERATORS INSTRUCTIONS FOR EDITOR #XMUM 90
Hardware requirement 90
Use of peripherals 91
Priority 91
Entry points 91
Operating instructions 91
Exception conditions 91
OPERATORS INSTRUCTIONS FOR EDITOR #XKYA 92
Hardware requirement 92
Use of peripherals 92
Priority 92
Operating instructions 92
Exception conditions 92.1

Appendix 1 Compilation error numbers 93

Appendix 2 Execution error numbers 97

Appendix 3 ICL 1900 Series codes 101
8-TRACK PAPER TAPE CODE 101
1900 CARD CODE 102
ATLAS CARD CODE 102

Appendix 4 Program and data formats on magnetic tape files 103
FILE STRUCTURE 103
FORMATTED DATA BLOCKS 103
UNFORMATTED DATA BLOCKS 103

Index 105

Tables

Table 1 Compiling system statements for a single program 3

Table 2 Statements in a STATEMENT TRACE list 31

Table 3 *Removed by Amendment list 1*

Table 4 Location of results for different functions 63

Table 5 The method of storage of FORTRAN variables 71

Table 6 Value of operand in PLAN instructions when calling a FORTRAN segment from a PLAN segment 72

Table 7 System statements for a batch of programs 82

Introduction

1900 SERIES COMPUTERS

Computers in the 1900 Series differ greatly in size, power and the number and type of peripheral units. The smaller machines run only one program at a time and are controlled by a human operator. The larger machines are multiprogramming; that is, they can run several programs simultaneously and may be controlled by one of the various operating systems available for different configurations.

Whether multiprogramming or not, each machine has supplied with it a program called *Executive.* This program may be regarded as a permanent part of the computer and works in conjunction with the operating system or human operator. It controls the multiprogramming activities of the larger machines; it also controls the peripherals and executes requests for the transfer of information between peripherals and the central processor. It also communicates with the operator and executes orders given by the operator. This communication is achieved via the *console typewriter,* which gives a permanent typed record of all communications in the order in which they take place. The operator can type instructions to Executive in order to carry out certain functions, for example to load, activate and delete programs. Executive also can use the typewriter in order to inform the operator of any unusual situations within the machine.

1900 SERIES COMPILERS

Compilers written for use with 1900 Series computers employ a common system of compiling. In this system a compiler does not translate source programs directly into machine code form; instead, programs written in any source language are first converted, by the appropriate compiler, into a series of segments in a common, intermediate language called *semi-compiled.* To produce machine code form the semi-compiled segments are combined by a routine known as the *consolidator.* The advantage of this system is that semi-compiled segments produced on different occasions (and possibly derived from more than one source language) may be combined together at the consolidation stage.

THE FORTRAN MAGNETIC TAPE SYSTEM

The 1900 Series FORTRAN magnetic tape system consists of a compiler #XFAM with a built in consolidator. A set of *compiling system statements,* resembling normal FORTRAN statements in format, is used to supply the compiler with the information necessary to produce an object program. The system is basically oriented towards 'load and go' running; however, it is possible to retain a complete program in binary or segments in semi-compiled. The compiler may also be used to compile a batch of programs for subsequent running. A program or segments of a program may be compiled from paper tape, cards or magnetic tape.

The compiler is overlaid from magnetic tape. Object programs produced may be overlaid or non-overlaid.

During compilation, an optional listing may be output to a line printer or a paper tape punch.

PART 1 PROGRAMMING CONSIDERATIONS

Chapter 1 Program structure

COMPILING SYSTEM STATEMENTS

A FORTRAN program written according to the rules given in the 1900 Series FORTRAN manual is incomplete in that it does not contain sufficient information to enable the compiler to produce an object program. Certain statements must be added to the basic program to provide the compiler with the necessary information; these statements are known as *compiling system statements*. There are three types of compiling system statements: *initial statements, program description statements* and *intersegment statements*. A full list of the compiling system statements is given in Table 1 on page 4 and a complete description of every statement is included in the remaining chapters in Part 1 of this manual.

Initial statements

Initial statements specify the input and output files that are to be used by the compiler (including the listing peripheral, if listing is required). In the absence of an output file specification the system will automatically load and run the object program, providing that the compilation has been error free; the object program is not retained after such a run. If the user does not wish to run the program immediately he may obtain a binary dump of the program by using the DUMP ON statement; the program will then not be executed unless the RUN statement is also specified. A binary dump of a program may be loaded, at a later date, by an operator FInd message.

The user may also obtain a copy of the semi-compiled output from the compiler by using the SEND TO statement. This will normally be required when the input to the compiler consists of a number of segments rather than a complete program; the semi-compiled output may then be consolidated on a later run.

Program description statements

The program description gives information about the object program to the FORTRAN magnetic tape compiling system. Program description statements associate source program channel numbers with 1900 system names, specify the level of error tracing that is to be incorporated in the object program, define the priority of the program and so on. The program description statements taken together constitute a segment (which is given the special name %%%F) and this segment must be the first segment input to the compiler.

A program description must begin with one of the following statements:

PROGRAM *(name)*

OVERLAY PROGRAM *(name)*

SEGMENTS *(name)*

OVERLAY SEGMENTS *(name)*

name is a four character program name which will be used to refer to the program when it is loaded and run.

The PROGRAM statement is used to introduce a non-overlaid program which is to be compiled and consolidated.

The OVERLAY PROGRAM statement is used to introduce an overlay program which is to be compiled and consolidated (see Chapter 8, *Overlays*).

When a source program is not complete, or its consolidated object program is not required, the program description must be introduced by a SEGMENTS statement.

The OVERLAY SEGMENTS statement is used instead of a SEGMENTS statement in order to introduce segments to be compiled which may later be included in an overlay program (see Chapter 8, *Overlays*).

The last statement in the program description must be the statement

END

Other program description statements are optional and may appear in any order between the first and last statements of the program description.

Intersegment statements
Certain initial and program description statements may be input between source segments if it is required to alter the original specifications during compilation. For example, the input medium may be changed between segments by the READ FROM statement.

Terminator
The end of the input to the compiler must be indicated by the terminator statement:

FINISH

Input sequence
The sequence of the input to any compiling run is as follows:

* Initial statements
  PROGRAM/OVERLAY PROGRAM/SEGMENTS/OVERLAY SEGMENTS
  ....
* Program description statements
  ....
  END

  Source or semi-compiled segments
  ....
* Intersegment statements
  ....
  Source or semi-compiled segments
  ....
  FINISH

The sections preceded by an asterisk* are optional.

Most of the compiling system statements are optional and standard default action will be taken if they are omitted.

Format of compiling system statements
All compiling system statements resemble normal FORTRAN statements and are written in columns 7 to 72 on normal coding sheets. Spaces are ignored except in file or subfile names. No continuation lines are permitted within compiling system statements.

Batch compilations
A program may be compiled either singly or as part of a batch. If it is compiled as part of a batch, the effect of certain compiling system statements will be different. A full list is given in Chapter 14, *Effect on compiling system statements.*

Examples

The following examples illustrate two possible combinations of compiling system statements with their effects.

Example 1

LIST	Full listing on a line printer, the program to be loaded and run after compilation and not retained.
PROGRAM (TEST)	Object program called TEST.
INPUT 1 = CR0	Source program channel 1 is identified as card reader 0.
OUTPUT 2 = LP0	Source program channel 2 is identified as line printer 0.
	Tracing at level 1 will be incorporated in the object program by default action.
END	
Segments	Segments written according to the 1900 Series FORTRAN Manual.
FINISH	

Example 2

DUMP ON (MT, PROGRAM MINE)	A binary dump of the object program is to be made to a magnetic tape file named PROGRAM MINE. A short listing is to be output to a line printer.
RUN	The program will be loaded and run after compilation.
PROGRAM (MINE)	
INPUT 2 = TR0	
OUTPUT 4 = LP0	
TRACE 2	Tracing at level 2 will be incorporated in the object program.
END	
Segments	
LIBRARY	The following segments are from a library, they will be included in the program only if needed.
Segments	
FINISH	

Group	*Statement*	*See note*
Initial Statements	LIST/SHORTLIST/NOLIST	1
	RUN	1, 4
	SEND TO	1, 4
	DUMP ON	1, 4
	READ FROM	1, 4
Program description statements	PROGRAM/OVERLAY PROGRAM/ SEGMENTS/ OVERLAY SEGMENTS	2
	COMPACT/EXTENDED DATA/ MIXED SEGMENTS	1, 4
	COMPRESS INTEGER AND LOGICAL	1, 4
	COMPRESS DOUBLE PRECISION	1, 4
	INPUT	1, 3, 4
	OUTPUT	1, 3, 4
	USE	1, 3, 4
	CREATE	1, 3, 4
	OMIT	1, 3, 4
	OVERLAY	1, 3, 4
	DEPTH OF OVERLAY	1, 4
	TRACE*n*	1, 4
	LEADER	1, 4
	PRIORITY	1, 4
	END	2
Intersegment statements	READ FROM	1, 3, 4
	LIST/SHORTLIST	1, 3, 4
	TRACE *n*	1, 3, 4
	LIBRARY	1, 3, 4
Terminator	FINISH	2

Notes:
1 Optional statement.
2 Obligatory statement.
3 More than one such statement may occur within group.
4 Order not essential within group.

Table 1: compiling system statements for a single program.

Chapter 2 Magnetic tape files

This chapter describes the structure of magnetic tape files used by the compiling system or in object programs produced by the system.

SIMPLE FILES, COMPOSITE FILES AND SUBFILES

Information is stored on magnetic tape in the form of a *file*. In the ICL 1900 Series FORTRAN Magnetic Tape System a file occupies one reel of magnetic tape. A file may be a *simple file* containing one category of information, or it may be a *composite file* holding many categories of information, possibly quite unrelated to each other. A simple file can contain information in one of the following categories:

1 Data.
2 Binary program.
3 Consolidated semi-compiled program.

A composite file contains one or more *subfiles*. In the simplest composite file structure one subfile follows another. A subfile may also contain one or more other subfiles, and this is called a *nested* subfile structure. There is no limit on the depth of the nesting.

A subfile which does not contain other subfiles may contain information in one of the following categories:

1 FORTRAN source program.
2 Unconsolidated semi-compiled segments.
3 Binary program.
4 Consolidated semi-compiled program.

Named files and scratch files

On most computer installations magnetic tapes are divided into two groups: *named files* and *scratch files*. Named files are used when information is to be retained. Each named file is given a unique name and is assigned to a particular user or group of users; a named file can be opened only if it is requested by name. A scratch file can be used by any user as a work file during the running of a program; once the program has finished, the scratch file is not considered to contain any useful information and can be used by another program.

File and subfile names

Each file and subfile on magnetic tape is identified by a name consiting of up to 12 characters chosen from the following set:

A to Z

0 to 9

Space

Hyphen (-)

The first character must be a letter.

In the manual, file names are denoted by *f* and subfile names by *s*.

Generation number

All magnetic tape files used in an installation should be uniquely identified. Files or subfiles that contain different versions of the same information may be distinguished by the addition of an integer generation number written in parentheses after the name e.g. MYSOURCEFILE (10).

Magnetic tape file and subfile names and generation numbers are referred to in this manual as follows:

$f(g)$, $f'(g')$, $s(g_s)$, $s'(g_s')$

The specified generation number is stated when a named file is created. When a named file is accessed, the generation number of the file will be checked against the specified generation number. If the generation number is

specified as zero, or is omitted, no check will be made and the first available file or subfile with the correct name will be used.

Retention period

All named files have a *retention period.* This is the length of time (in days) for which a file will be retained as a named file; it is also the number of days which the file will be protected against inadvertent overwriting. When the retention period has expired the file, by definition, becomes a scratch file. A scratch file has zero retention period and may have any name.

Under the FORTRAN magnetic system, it is possible to rename and re-use named files once the information on them is no longer needed. Therefore it is recommended that all named files have the highest possible retention period i.e. 4095 days. When a tape file is created in an object program by means of a CREATE statement (see the section *Magnetic tape channel description statements* in Chapter 3 on page 9) a retention period can be specified by the user. This retention period may be specified as an integer in the range 0 to 4095; the tape is given a retention period of 4095 if no retention period is specified.

Write permit ring

The *write permit ring* is a small plastic ring that must be fitted to a magnetic tape spool before the magnetic tape may be written to; the absence of a write permit ring thus prevents the accidental overwriting of information which is to be retained.

FORTRAN magnetic tape formats

Details of the formats of FORTRAN data on magnetic tape are given in Appendix 4. For details of source program format on magnetic tape see the manual *Compiling Systems* (TP 4241).

These are not normally of interest to programmers writing solely in FORTRAN.

Chapter 3 Object program input/output

CHANNEL DESCRIPTION STATEMENTS

Input/output statements within a FORTRAN source program refer to their related peripheral units by INTEGER *channel numbers.* These must be in the range 1 to 4095 (0 may be used in special cases) in 1900 FORTRAN. When the program is compiled these channel numbers are associated with the 1900 system names of the peripheral units by *channel description statements* in the program description.

There are four types of channel description statement, as follows:

```
INPUT    k = Z  (Input only)
OUTPUT   k = Z  (Output only)
USE      k = Z  (Input and output)
CREATE   k = Z  (Input and output)
```

k is a source program channel number and Z gives details of the peripheral to be associated with *k*. Alternatively *k* could be a list of channel numbers separated by commas, in which case all these channels are associated with the same peripheral Z. The same Z should not appear in more than one channel description statement.

Any number of different channel numbers may be defined. Some peripherals can hold either formatted or unformatted records, others may hold only formatted records. Normally, at least one formatted output channel should be available and this must be defined in the first OUTPUT statement of the program description. This channel is used for the output of diagnostic information. (See Chapter 7, *Program testing.*)

Channels can be defined in the program description even if they may not be used on particular runs of the program. However, the more channels that are defined (particularly if they are on different types of peripheral) the more space is used by the input/output system in the object program.

Part of Z is normally a 1900 *system name* of the relevant peripheral. This consists of a two letter code followed by a *logical unit number,* e.g. LP2 is a line printer with logical unit number 2. These numbers are chosen by the programmer and are then assigned to a particular peripheral or file by the 1900 system. Logical unit numbers must be in the range 0 to 15. Numbers for peripherals or files with the same code must be distinct. Thus, LP2 should not appear twice in the program description (unless on any particular run of the program it will be referred to by only one of the relevant channel numbers) but a card punch could be CP2.

The peripherals and input/output facilities available to the object program are:

- Paper tape reader
- Paper tape punch
- Card reader (1900 or ATLAS code)
- Card punch (1900 or ATLAS code)
- Line Printer
- Magnetic tape
- Exchangeable disc store (serial files)
- Fixed disc store (serial files)
- Console typewriter
- Core store transfers
- Suppression of transfers

If the dump and restart facility is used (see the manual *FORTRAN: Compiler Libraries*), a USE or CREATE statement must specify a file for this purpose. The file used may be on magnetic tape or on E.D.S. or F.D.S.

The facilities for handling E.D.S. and F.D.S. serial files are identical with those described in the manual *FORTRAN: 16K Disc Compiler* (TP4131); no further information is given here.

BASIC PERIPHERALS

Details of programming for basic peripherals appear in the 1900 Series FORTRAN manual. Only formatted records may be processed. Basic peripheral names may appear only in INPUT and OUTPUT statements with one of the following forms:

INPUT	*k* = TR*n*	Paper tape in 1900 code
INPUT	*k* = CR*n*	Cards in 1900 code
INPUT	*k* = CR*n*/ATLAS	Cards in ATLAS code
OUTPUT	*k* = TP*n*	Paper tape in 1900 code
OUTPUT	*k* = CP*n*	Cards in 1900 code
OUTPUT	*k* = CP*n*/ATLAS	Cards in ATLAS code
OUTPUT	*k* = LP*n*	Line printer
OUTPUT	*k* = LP*n*/*l*	Line printer

k is the channel, as defined previously

n is the logical unit number, as defined previously

l is the line printer line-length, i.e. 96, 120 or 160; absence of *l* implies 120 characters.

Examples

1 INPUT 3 = CR0
   Source program channel number 3 is specified as the card reader assigned as CR0.

2 OUTPUT 6 = LP0/160
   Source program channel 6 is specified as a 160 print position line printer assigned as LP0.

Double buffering

Transfers between basic peripherals and FORTRAN object programs are normally single buffered; double buffering may be specified for one basic peripheral of each type, with the exception of punch card devices using the ATLAS card code, which may not be double buffered. A basic peripheral will be double buffered if it is assigned a logical unit number 7.

Double buffering increases the transfer rate of a peripheral to which it is applied and is therefore desirable in any program involving large numbers of transfers; this increased transfer rate is obtained at the cost of an increased core store requirement.

Examples

INPUT	2, 6	= TR7
INPUT	8	= TR0
INPUT	3	= CR7
OUTPUT	4	= LP7
OUTPUT	5	= CP7/ATLAS

At run time units TR7, CR7 and LP7 will be double buffered; TR0 and CP7 will be single buffered.

MAGNETIC TAPE FILES

Simple files held on magnetic tape may be processed using either formatted or unformatted READ and WRITE statements.

Effect of source program statements for magnetic tape files

The following sections describe the effects of various FORTRAN statements when applied to magnetic tape files. General programming information is given in the manual *FORTRAN*.

In the following sections, the channel number is denoted by *c*, meaning an integer variable or constant.

READ AND WRITE STATEMENTS

A READ statement inputs the next available record. A WRITE statement outputs a record after the last record that was read or written (unless the file has been re-positioned by using REWIND or BACKSPACE). A WRITE statement that attempts to overwrite an already written record will destroy that record and any following records on the file.

ENDFILE STATEMENT

Processing of an output file must be terminated by execution of an ENDFILE (or REWIND) statement after the last record has been output by a WRITE statement. If no ENDFILE (or REWIND) statement is executed the file may be left in a non-standard format and may not be readable.

The form of the statement is:

ENDFILE *c*

ENDFILE does not rewind the tape.

REWIND STATEMENT

The REWIND statement will rewind the specified file and position it so that the next READ or WRITE statement applies to the first record of the file.

The form of the statement is:

REWIND *c*

BACKSPACE STATEMENT

The BACKSPACE statement positions the file so that the previous record is available. For a formatted record, physical tape movement may or may not take place.

Repeated backspace operations lead to inefficient tape usage and should be avoided if possible. In some cases REWIND followed by repeated READ statements may be preferable.

The form of the statement is:

BACKSPACE *c*

Opening of magnetic tape files

Before a magnetic tape file is used it must be 'opened', i.e. made available by the 1900 system. In general, this happens automatically the first time a FORTRAN statement referring to the file is executed. If required, the user may open the file earlier by using the ALLOT subroutine.

Alternatively, the user may wish to open the file himself by means of a PLAN subroutine. This is acceptable to the FORTRAN system, which always checks if a file is already open before itself attempting to open it.

Magnetic tape channel description statement

The format for a channel description statement specifying magnetic tape is as follows:

hk = MT*n*/*q* (*f* (*g*,*r*)) /*b*

h is one of INPUT, OUTPUT, USE, CREATE.

k is the channel, as defined previously.

n is a magnetic tape logical unit number (and must not be zero for an overlay program: see page 39).

q is the qualifier defining the format of the records (see below).

f is the file name. Alternatively, it may be a dummy file name, to be replaced at run time by the actual file name. (See the section *File names at run time* on page 10.)

g is the file generation number (optional).

r is the retention period and is used only in CREATE statements.

b is the block size in words (optional - see below).

The different values of *h* are interpreted as follows:

INPUT - use the named magnetic tape file for input only. A write permit ring must not be present.

OUTPUT - use the named magnetic tape file for output only. A write permit ring must be present.

USE - use the named magnetic tape file for input and/or output. A write permit ring must be present.

CREATE - obtain a scratch tape, rename it, use if for output, followed by input if required. A write permit ring must be present.

The file name and enclosing parentheses may be omitted for the USE statement. In this case a scratch tape is obtained and remains scratch at the end of the program. This is the normal procedure if the tape is to be used purely as a work tape.

RECORD FORMAT

The qualifier *q* defining the format of the records on the magnetic tape is normally one of the following:

FORMATTED - indicating formatted records.

UNFORMATTED - indicating unformatted records.

DUMP' - indicates use of file for dump and restart

q may be omitted in which case UNFORMATTED is assumed.

Formatted files used in Basic FORTRAN programs compiled by #XFOM have a different format. If such files are to be read or produced by the FORTRAN magnetic tape compiler, the qualifier OLDFORMAT must be used.

BLOCK SIZE

b is the block size, in words, of the magnetic tape blocks used to hold the FORTRAN records. In general, a large block size is more efficient but occupies correspondingly more core store. The maximum value of *b* depends on the type of tape deck but is at least 4096.

If *b* and the preceding solidus / are omitted then the block size is assumed to be 128 words.

Formatted records are held one or more to a block. Maximum record length is $b - 2$ words. Each unformatted record starts a new block, and occupies as many further blocks as are necessary to hold the whole of the record. There is no maximum record length for unformatted records. Note that four words of each block are used to hold organizational information in unformatted files.

Examples

1 OUTPUT 3 = MT1/FORMATTED (DEPT-15(1))
  Channel number 3 in the program corresponds to the file DEPT-15(1) to be allocated as MT1. The block size will be 128 words.

2 USE 2 = MT3/1024
  The compiler associates channel number 2 in the program with a scratch tape to be opened as MT3 for the input and/or output of unformatted records. The block size will be 1024 words.

File names at run time

Details of a magnetic tape file to be processed may not be known at the time a program is compiled. In this case, the file name should be specified in a peripheral description statement as 'UNKNOWNASYET'. Then, the file name and other details can be provided at run time by a call of the subroutine FILE of the form

CALL FILE (*c*, *H*(*n*), *g*, *r*)

where *c*, *g* and *r* are integer expressions specifying respectively the source program channel number, the file generation number and the retention period. *H(n)* is either the first of two consecutive array elements holding the 12-character file name or is a TEXT constant. The retention period is ignored and should be zero except for a magnetic tape file with a CREATE channel description statement. In the latter case, a retention period of 4095 days is recommended.

The FILE subroutine must be called before the file is opened, i.e. before any READ, WRITE or other input/output statement references the file. A call to FILE does not in itself open the file. Opening will take place as usual on the first other reference to that channel.

Example

A program description contains the statements

```
OUTPUT  7  =  MT1  (UNKNOWNASYET)
INPUT   1  =  CR0
```

The program contains the statements

```
      READ (1,100) FNAME (1), FNAME (2), IGEN
100   FORMAT (2A8, I3)
      CALL FILE (7, FNAME (1), IGEN, 0)
```

The effect is to read a file name and generation number from a card; then, when there is a READ or WRITE for channel 7, the file that is opened is the one specified by this card.

File renaming

It may be convenient to open a named magnetic tape file and give it a new name. In this case, the original file name must be specified in an OUTPUT, USE or CREATE statement (or possibly by a call to the subroutine FILE). Then, the new name and other details can be provided at run time by a call of the subroutine RENAME of the form:

CALL RENAME *(c, H(n), g, r)*

where *c, g* and *r* are integer expressions specifying respectively the source program channel number, the new file generation number and the new retention period. *H(n)* is either the first of two consecutive array elements holding the new 12-character file name or is a TEXT constant. A retention period of 4095 is recommended.

The subroutine can be called at any time. It will open the file under its original name if it is not already open, then rename it. Renaming a magnetic tape file destroys all information currently on that file.

Example
A program description contains the statements

```
OUTPUT  8 = MT2/FORMATTED (OUTFILE)
INPUT   1 = TR0
```

The program contains the statements

```
      READ (1, 100) FNAME (1), FNAME (2), IGEN
100   FORMAT (2A8, I3)
      CALL RENAME (8, FNAME (1), IGEN, 4095)
```

The effect is to read a file name and generation number from paper tape, open the magnetic tape file OUTFILE (if it is not already open), rename it with the new file name and generation number, and give it the standard retention period of 4095 days.

Printing of formatted magnetic tape files

The utility program #XRMF may be used to list, on the line printer, the contents of any magnetic tape formatted file.

There are two ways in which the program may be used:

1 A listing in which a throw to head of form is made followed by each record in the file printed as a new line.

2 A listing in which the first character in each record is interpreted as a FORTRAN carriage control character exactly as for direct line printer output.

The specification of #XRMF contained in the manual *Library Specifications* should be consulted for details of operating.

END OF FILE MARKERS

The end of file markers listed below are recognized on execution of a FORTRAN program. An execution error (see Appendix 2) is flagged if an attempt is made to read beyond the end of the data and an end of file marker is encountered. The marker for magnetic tape and disc files are created automatically by an ENDFILE statement.

Type of file	*Marker*
cards	four asterisks in the first four columns
paper tape	four asterisks in the first four character positions of the record
magnetic tape	a trailer label
disc	an end of file bucket

CHANNEL 0

Channel 0 may be used in the source program but must not be directly defined in the program description. Details are given in Chapter 7, Program testing (see page 29).

CONSOLE TYPEWRITER

FORTRAN programmers may use the console typewriter to output short messages to the operator with formatted WRITE statements. These messages will normally be requests, or reminders for operator action. The channel description statement will be as follows:

OUTPUT *k* = TY0

where *k* is the channel, as defined previously. The message sent to the console typewriter will vary, depending upon the particular machine, but for a multiprogramming machine it will be

0#*name* DISPLAY:– *output record*

where *name* is the four-character name of the program, and *output record* is up to 40 characters. The characters $] ↑ ← must not be used in the message.

CHARACTER TRANSFERS IN CORE STORE

It is often useful to be able to reorganize the contents of a record, in character form, prior to output. For example it may be desirable to introduce space characters into long numbers to improve the legibility of the output. This cannot be done directly using the E, F and I conversion codes; however, it is possible to use an array as if it were a peripheral and 'output' values to it in character form, using WRITE statements. The character handling subroutine COPY may then be used to rearrange the contents of the array before they are output to an actual peripheral using the A conversion code.

Similarly, one or more records may be read into an array using the A code in formatted READ, then re-read using some other format.

A channel number must be associated with an array to allow it to be specified in READ or WRITE statements; this is done by a call to the subroutine DEFBUF, as follows:

CALL DEFBUF *(k, n, arrayname)*

where *k* is the channel number required (INTEGER expression), *n* is the length of the array in characters (INTEGER expression), and *arrayname* is the name of the array (of any type).

When a channel number is to be associated with a particular array at run time, the channel number must first have been defined as an array by a channel description statement of the form

USE *k* = /ARRAY

and then associated with the required array name by a call to DEFBUF.

A channel number may be associated with many arrays during the running of a program, but only one at a time.

Example

The statements below show how the input section of a program might be written to read a variable number of control cards of differing format. The first character determines the type of control card and is a digit in the range 1 to 9. The last control card is blank except for the digit 9 in the first character position.

```
LIST
PROGRAM (EX01)
INPUT 1 = CR0
OUTPUT 2 = LP0
USE 6 = /ARRAY
END
MASTER INPUT
DIMENSION BUFFER (10), X (6)
CALL DEFBUF (6, 80, BUFFER)
```

```
  1 READ (1, 100) BUFFER
100 FORMAT (10A8)
    READ (6, 101) K
101 FORMAT (I1)
    GO TO (51, 52, 53, 54, 55, 56, 57, 58, 59) K
 50 PAUSE 77
    GO TO 1
 51 READ (6, 102) X, Y, N
102 FORMAT (1X, 2F10.6, I6)
    GO TO 1
 52 READ (6, 103) Z
103 FORMAT (1X, 6F10.6)
    GO TO 1
    .....
    etc
    .....
 59 Process data etc.
```

SUPPRESSION OF TRANSFERS

The programmer may wish to suppress transfers on some channels. This can be done by writing /NONE for Z in a channel description statement. A READ or WRITE statement referring to a channel specified in this way has no effect. Variables on an input list are not overwritten, and no action is taken on output.

Examples

```
INPUT   7 = /NONE
OUTPUT 2 = /NONE
```

READ or WRITE statements referring to channels 7 or 2 would have no effect. This feature is useful during program testing.

If a segment being tested has some READ statements, the unit can be specified as /NONE and suitable values of variables in the read list inserted by a test segment, using a COMMON block.

Similarly, it is possible to test an output segment without producing vast quantities of useless output.

Chapter 4 Compiler listing

During compilation a listing is usually output on a line printer. Alternatively, a listing may be obtained on the paper tape punch. The listing consists of a partial or complete list of source program statements together with other information about the program and details of any error found. The type of listing required is specified by means of certain compiling statements known as *listing statements*.

LISTING DESCRIPTION

Listing can be obtained in two degrees of detail or omitted entirely. The more detailed form is known as a *full listing;* the less detailed is called a *short listing*.

Full listing

The first line of a full listing contains the name of the compiler, date of compilation and the time at which compilation commenced (if an internal clock is fitted). Subsequent lines list all the compiling system statements, FORTRAN source statements and compilation error messages. The length of each segment, that is the number of object program instructions generated, is given at the end of each segment. Semi-compiled segments are listed by name only. When reading from subfiles on magnetic tape, the beginning and end of each subfile is recorded in the listings.

Compiling time error messages are described in Appendix 1.

Short listing

A short listing is similar to a full listing but does not list each FORTRAN source statement in detail. A short listing provides segment names, compiling system statements and compilation error messages. In most other respects the listing is similar to a full listing. A short list should not normally be used during the development of a program.

LISTING STATEMENTS

A program is normally short listed on a line printer unless either the programmer includes a listing statement or the program is compiled as part of a batch (see Chapter 13). The possible listing statements are:

LIST

SHORT LIST

SHORT LIST (TP)

NO LIST

LIST or SHORT LIST will output the appropriate type of listing on a line printer. SHORT LIST (TP) outputs a short list on paper tape. NO LIST supresses all listing during compilation; this statement should be used only for fully tested programs, since if an error is found, no useful information can be listed.

LIST or SHORT LIST may be used either as initial statements or as intersegment statements. In the latter case they control the level of listing of all the following segments until the next listing statement is encountered.

SHORT LIST (TP) or NO LIST, if used, must be the first initial statement.

Chapter 5 Compiler output

This chapter describes the compiler output (excluding listing) for a single program compilation; details of the output from a batch compilation are given in Chapter 13.

TYPES OF OUTPUT

Load and go

Unless the user specifies otherwise, a complete FORTRAN program whose program description starts with a PROGRAM (or OVERLAY PROGRAM) statement is compiled and then run immediately. No copy of the object program is retained.

Binary program

If a program is to be run more than once unchanged, then it need not be constantly recompiled; the user may nominate in a DUMP ON statement the name of a file to which a copy of the binary program is to be written. The binary program can then be run a number of times. It can be run immediately after compilation if a RUN statement is used as well as the DUMP ON statement.

Semi-compiled segments

A group of segments may be translated into semi-compiled form without being consolidated if the program description starts with either of the statements SEGMENTS or OVERLAY SEGMENTS (see Chapter 1). The file and subfile to hold the semi-compiled segments must be nominated in a SEND TO statement. Several such groups of segments may be consolidated as a complete program later.

Consolidated semi-compiled program

At one stage during the compilation of a complete program, the program exists in *consolidated semi-compiled* form. This is an intermediate form which normally exists only on a scratch file and is not retained. If the program is required in this form, it can be output to a file named in a SEND TO statement. A non-overlaid consolidated semi-compiled program can be treated in very much the same way as a binary program in that it can be loaded and run as many times as necessary; however, it will take longer to load than a binary program. An overlaid consolidated semi-compiled program is less easy to use since, when an attempt is made to load it, another file (a scratch file or a file named in a DUMP ON statement) is opened and the binary program is written to it.

It is possible to treat a consolidated semi-compiled program in the same way as a group of semi-compiled segments if care is taken. The consolidated semi-compiled program can be input to a later compilation along with other segments in source or semi-compiled form and reconsolidated (see *Input from simple files* on page 23).

INITIAL STATEMENTS

The DUMP ON statement

The DUMP ON statement nominates a file on which a binary version of the program is to be created. It has the general form

DUMP ON (MT,*f (g)*)

The simple file *f* is opened, renamed PROGRAM *name* (0), where *name* is the four character program name specified in the PROGRAM or OVERLAY PROGRAM statement, and is given a high retention period; the binary program is then copied to this file. If *f*, the file name, is already PROGRAM *name*, the file is not renamed and the original retention period and generation number are retained.

The DUMP ON statement is required only if a copy of the binary program is to be retained. It must not be used if the program description starts with a SEGMENTS or an OVERLAY SEGMENTS statement.

Example

DUMP ON (MT, PROGRAM ABCD)

PROGRAM (ABCD)

. . . .

END

. . . .

FINISH

A binary copy of the program is made on a simple file already named PROGRAM ABCD. The program is not run immediately.

The RUN statement

The RUN statement indicates that the program is to be loaded immediately after compilation; it has the general form

RUN

The RUN statement is required only if a DUMP ON statement is present and if the object program is to be run immediately. The RUN statement is superfluous if used when no DUMP ON statement is present for a complete program but it is allowable. However, it must not be used if the program description startes with a SEGMENTS or an OVERLAY SEGMENTS statement.

Example

LIST

DUMP ON (MT, PROGRAM PXAB)

RUN

OVERLAY PROGRAM (JACK)

. . . .

END

. . . .

FINISH

The file PROGRAM PXAB is renamed PROGRAM JACK; the binary program is written to the file and is then loaded ready for running.

The SEND TO statement for semi-compiled segments

When a set of segments not comprising a complete program is compiled with a SEGMENTS or an OVERLAY SEGMENTS statement a SEND TO statement must be present to nominate the file and subfile to which the segments are to be written. The following are forms of the SEND TO statement for use with semi-compiled segments.

1 CREATE NEW COMPOSITE SUBFILE. If the format

SEND TO (MT, $f(g)$, $f'(g')$. $s(g_s)$)

is used, the file $f(g)$ is renamed $f'(g')$ and is given a high retention period. If $f(g)$ is the same as $f'(g')$, no renaming occurs. The file is created as a composite file containing a single subfile $s(g)$, of semi-compiled segments; Any previous contents of $f(g)$ are destroyed.

2 ADD TO EXISTING COMPOSITE FILE. If the format

SEND TO (MT, $f(g)$. $s(g_s)$)

is used, the file $f(g)$ is scanned for a subfile $s(g_s)$. If $s(g_s)$ is found, its contents are overwritten with the semi-compiled segments; any later subfiles in the file are destroyed. $s(g_s)$ must not be contained in any other subfile. If $s(g_s)$ is not found, a subfile with that name is created at the end of the file and the semi-compiled segments are written to it.

3 REPLACE EXISTING SUBFILE. If the format

SEND TO (MT, $f(g)$. $s(g_s)$,. $s'(g'_s)$)

is used, the resulting action is similar to that in 2 above except that if $s\,(g_s)$ is found it is renamed $s'\,(g'_s)$. If it not found, $s'\,(g'_s)$ is created at the end of the file and an error is flagged on the listing.

Example

```
LIST
SEND TO (MT, OLDFILE, NEWSEGFILE, NEWSEGS)
SEGMENTS
END
..........
Program segments
..........
FINISH
```

This is an example of the use of format 1. The file OLDFILE is renamed NEWSEGFILE; the file NEWSEGFILE is a composite file containing one subfile, NEWSEGS to which the semi-compiled segments are written.

The SEND TO statement for consolidated semi-consolidated program

For a complete program, a SEND TO statement is not normally used. Either the program is to be run immediately or a binary version of the program is created using a DUMP ON statement. A consolidated semi-compiled program is created on a scratch file during the compilation. However, a named file may be nominated instead in a SEND TO statement. The file is normally a simple file and one of the following formats of the statement is used.

1 If the format

SEND TO (MT, *f (g)*)

is used, the file with the name *f (g)* is renamed PROGRAM *name* (0), where *name* is the four character program name; the file is given a high retention period. If *f* is already PROGRAM *name* no renaming occurs.

2 If the format

SEND TO (MT, *f (g), f'(g')*)

is used, the file *f (g)* is renamed *f' (g')* and is given a high retention period.

Alternatively, a consolidated semi-compiled program can be copied to a subfile of a composite file by using any one of the forms of the SEND TO statement given in the section *The SEND TO statement for semi-compiled segments* above.

Example

```
SEND TO (MT, LOCALFILE)
DUMP ON (MT, PROGRAM MINE)
RUN
PROGRAM (MINE)
..........
END
..........
FINISH
```

The file LOCALFILE will be renamed PROGRAM MINE (0) and during compilation, the consolidated semi-compiled program will be written to this file. At the end of compilation, a binary version of the program will be written to the file PROGRAM MINE and the program will be run.

COMPILATIONS WITH ERRORS

If, during compilation, an error is encountered, no more semi-compiled program will be generated but the source program will still be read and any errors found will be reported. Compilation will end when the FINISH statement is encountered; the program will not be run even if a RUN statement is present.

If errors are found, subfiles and simple files will still be created or renamed as specified in SEND TO statements;

however, a file named in a DUMP ON statement will not be renamed.

In general, no useful output except the listing, is obtained from a program with errors. But, if a file has been nominated in a SEND TO statement to receive the semi-compiled program, any segments compiled before the one containing the error will be on the file and may be used as input to a subsequent re-compilation (see *Input from simple files*).

SUPPRESSION OF COMPILER OUTPUT

Output of semi-compiled program, consolidated or otherwise, may be suppressed by

SEND TO (NONE)

No magnetic tapes are required for output. This statement is useful when the first compilation is be used to locate errors. No attempt is made to consolidate a PROGRAM compilation.

Chapter 6 Compiler input

This chapter describes the various media from which the FORTRAN magnetic tape compiler will accept input. It also describes the different forms that input may take and describes the system in detail.

COMPILER INPUT MEDIA

The FORTRAN magnetic tape compiler will accept input from punched cards (punched in 1900 or ATLAS code), paper tape or magnetic tape. The input may consist of FORTRAN source segments, semi-compiled segments or a mixture of both.

Most programs presented to the compiler are held on only one basic input medium. However it is possible to switch the input medium between segments of a program. Thus one segment may be input on cards, the next segment on magnetic tape, etc. A switch in medium is specified by means of a READ FROM statement which may be present between segments of a program as in intersegment statement. The statement may also appear as an initial statement or immediately after the program description.

Initial input medium

Initially, the input medium must be paper tape or cards. The initial input medium is indicated by the entry point to the compiler. There are three alternative entry points corresponding to the following three cases:

1 Start the compilation, reading initially from 8-track paper tape punched in 1900 code.

2 Start the compilation, reading initially from cards punched in 1900 code.

3 Start the compilation, reading initially from cards punched in ATLAS code.

Subsequently a switch in the input medium may be made whenever a correct READ FROM statement is encountered.

COMPILING FROM BASIC PERIPHERALS

If a complete program is not all held on the same basic medium or punched in the same code, (cards only), a switch between peripherals and/or card codes may be made by a READ FROM statement. A change in basic peripheral causes the previously used peripheral to be released by the compiler and the requested one assigned. The compiler continues reading from the new peripheral until the end of the program is reached or until another READ FROM is encountered.

In this context, the possible forms of the READ FROM statement are:

READ FROM (TR)	-	continue, reading from paper tape.
READ FROM (CR/ATLAS)	-	continue, reading from cards punched in ATLAS code.
READ FROM (CR/S1900)	-	continue, reading from cards punched in 1900 card code.
READ FROM (CR)	-	continue, reading from cards. The card code is taken to be the same as the most recent card input, established either by initial entry point or by a READ FROM (CR/ATLAS) or READ FROM (CR/S1900) statement. If all previous input was from paper tape, then 1900 card code is assumed.

Example

Consider a source program, the master segment of which, MAST, is held on paper tape. MAST calls two subroutines one of which, SEG1, is on cards; the other, SEG2, was originally written for another machine and is held in ATLAS card code.

The input consists of the following:

```
LIST
PROGRAM (MIXI)
INPUT    2 = TRO
OUTPUT   1 = LPO
END
MASTER MAST
source statements
..........
END
READ FROM (CR)
```

These statements are held on paper tape

When READ FROM (CR) is read, the tape reader is released and card reader assigned.

```
SUBROUTINE SEG1
source statements
..........
END
READ FROM (CR/ATLAS)
```

These statements are held on cards in 1900 Series card code. The same card reader remains assigned to the complier.

```
SUBROUTINE SEG2
source statements
..........
END
FINISH
```

These statements are held on cards in ATLAS card code.

COMPILING FROM MAGNETIC TAPE

Both source and semi-compiled segments on magnetic tape may be input to the compiler. Magnetic tape files input to the compiler can be either composite or simple files. A FORTRAN source program or source segments must be held within a subfile in a composite file. Semi-compiled segments may also be held in a subfile which may be generated by the FORTRAN magnetic tape compiler or some other magnetic tape compiler. The corresponding forms of the READ FROM statement are described in the following sections.

Input from subfiles

It is possible to compile segments held in subfiles on magnetic tape. The subfiles may contain source segments or semi-compiled segments.

The programmer specifies the required subfiles to the compiler by one of the following forms of the READ FROM statement:

READ FROM (MT, $f(g)$. $s(g_s)$)

READ FROM (MT,. $s(g_s)$)

When a file name f is present, as in the first type of READ FROM statement, the compiler normally releases any input file already open, opens f and searches for the subfile s. If f is already open, it is rewound and the search for s is made from the beginning.

If the file name f is omitted, as in the second form of the READ FROM statement, the search for subfile s starts from the current position on whatever input file is already open.

The subfile s may contain other subfiles nested to any depth. In this case all subfiles in the nest are compiled.

On reaching the end of the subfile to be compiled, the compiler reverts back to the reader from which the READ FROM statement was input.

Subfiles containing information other than FORTRAN source or semi-compiled segments are ignored, and if a subfile of the correct name containing FORTRAN source or semi-compiled segments is not found before the end of the composite file is reached, an error message is indicated on the line printer (see Appendix 1, *Compilation error numbers*). The compilation continues in error mode i.e. semi-compiled output is suppressed.

Compiling system statements on magnetic tape

READ FROM statements must not be input from magnetic tape (with one exception, see Other system libraries). Other compiling system statements (with the exception of SHORT LIST (TP) and NO LIST statements), may be held on magnetic tape together with the source program and input at the same time. However, it is recommended that all compiling system statements are input from a basic peripheral as this allows far greater flexibility particularly if a compiling system statement is to be amended subsequently.

Examples

1 A source program is contained on cards except for two segments held in a subfile SEGS in a composite file ANALYSIS, on magnetic tape with file generation number 1.

Input consists of the magnetic tape file together with the following statements on cards:

```
LIST
PROGRAM (FRED)
INPUT    3 = CR0
OUTPUT   8 = MT0/FORMATTED (RESCHAP)
END
MASTER SEGA
source statements
..........
END
READ FROM (MT, ANALYSIS (1). SEGS)
FINISH
```

The compiler will read from cards until the READ FROM statement is encountered. Then it will compile from magnetic tape, until the end of the subfile SEGS is reached. At this stage it will revert to reading from cards, i.e. the FINISH Statement.

2 Each segment of a program is contained in a separate subfile on magnetic tape. The names of the subfiles in the order in which they appear on the composite file INFILE are PROGDES (containing compiling system statements), SEG1, SEG2, SEG3. The input consists of the magnetic tape file together with the following statements on punched cards:

```
LIST
READ FROM (MT, INPFILE.PROGDES)
READ FROM (MT,.SEG1)
READ FROM (MT,.SEG2)
READ FROM (MT,.SEG3)
FINISH
```

Input from simple files

A consolidated semi-compiled program written to a simple file by a previous compilation (see *The SEND TO statement for consolidated semi-compiled programs* on page 19) may be input for reconsolidation with a different program description, possibly with some segments replaced. When two or more segments having the same name are input to the compiler, only the first is consolidated, whether this is source or semi-compiled.

A segment may be replaced therefore by inputting the required replacement segment before the segment to be replaced and with the same name. When using this technique it is essential that a new program description is provided.

The READ FROM simple file statement takes the form

READ FROM (MT, *f(g)*)

where *f* is the filename and *g* is the generation number, which may be omitted together with its surrounding parentheses.

Unpredictable results may occur if different issues of the compiler and/or its library are used for the initial and later compilations.

Example
A consolidated semi-compiled program is contained in a simple file PROGRAM KANE. It is to be reconsolidated with a different program description and with the subroutine segment TRYTWO replaced by a later version that has to be compiled.

```
LIST
PROGRAM (KAN2)
INPUT    2 = CR0
OUTPUT   3 = MT1/FORMATTED (RESKAN2)
END
SUBROUTINE TRYTWO
..........
..........
END
READ FROM (MT, PROGRAM KANE)
FINISH
```

Editing of source programs on magnetic tape
Source programs may be created and updated on magnetic tape by means of one of the editing programs #XKYA or #XMUM. (See Chapter 10.)

INCORPORATION OF LIBRARY SUBROUTINES INTO A PROGRAM

FORTRAN compiler library
The FORTRAN compiler library, which is called SRF7, contains the standard functions and subroutines available to FORTRAN programs; routines contained in the SRF7 group are automatically incorporated into a program whenever they are referenced. Full details of the routines contained in the SRF7 group are given in the *FORTRAN Compiler Libraries* manual.

An installation may add functions or subroutines to the SRF7 group, if required. These additional routines will also be automatically incorporated when referenced.

Other system libraries
Other system libraries may be held on the *system library tape;* the tape holding the FORTRAN magnetic tape compiler. These libraries may be standard libraries supplied by ICL or special libraries created by the installation. If a program references a routine contained in any library other than SRF7, then the compiler must be informed as to which library is to be scanned for the required routine. This is done by means of a special form of the READ FROM statement, which must be preceded by the LIBRARY statement. If several libraries are to be scanned then a READ FROM statement must appear for each, preceded by one LIBRARY statement. The sequence of READ FROM statements must be followed by the FINISH statement. The LIBRARY statement must follow the last segment of the program.

The special form of the READ FROM statement is as follows

READ FROM (MT,-.*library*)

where *library* is the four-character name of the library to be scanned. This is the only form of the READ FROM statement that is itself allowed to be held on magnetic tape.

The FORTRAN compiler library, SRF7, is scanned automatically, if necessary, after any other libraries specified in READ FROM statements. If it is required that the compiler library is scanned before any other libraries then it must be explicitly specified in a READ FROM statement preceding any other library READ FROM statement.

When libraries are scanned for a routine, the first routine with the required name will normally be incorporated into the program; any subsequent routines with the same name will be ignored.

Example

```
Initial statements
..........
Program description statements
..........
FORTRAN source program
..........
LIBRARY
READ FROM (MT, –.USRL)
FINISH
```

In addition to standard routines, the FORTRAN source program in the example above references routines in a user's library, USRL, which is held on the system library tape. The READ FROM statement causes the compiler to scan this library, and then the SRF7 library is automatically scanned.

Libraries held on other tapes

A user may nominate one or more subfiles of any composite tape file to be scanned as library. Such a library need not be a system library. It could contain a subfile of semi-compiled segments produced by a 1900 Series compiler or it could contain FORTRAN source segments, in which case each segment is compiled but only these that are required are incorporated; this is obviously less efficient than the direct incorporation of semi-compiled segments.

Library subfiles not on the FORTRAN system tape are nominated by standard READ FROM statements, which must be preceded by a LIBRARY statement following the last segment of the program.

Example

```
Initial statements
..........
Program description statements
..........
FORTRAN source segments
..........
LIBRARY
READ FROM (MT, HOLDFILE.PRIVATELIB)
READ FROM (MT,              SECONDLIB)
FINISH
```

The effect of these statements is to make the contents of the two subfiles PRIVATELIB and SECONDLIB in the composite file HOLDFILE available to the program as libraries.

Paper tape and card libraries

It may be convenient to hold libraries that are accessed relatively infrequently on either paper tape or cards in 1900 or ATLAS code. Libraries held in this way may be in the form of either semi-compiled or source segments, although the latter is less efficient as each segment is compiled whether or not it is used.

Libraries held on paper tape or cards must be input to the compiler following the end of the source program and preceded by a LIBRARY statement; it is recommended that a LIBRARY statement should occupy the first card or paper tape record of such a library to ensure that the statement is not omitted.

Recommended order of searching libraries

When the FINISH statement is read by #XFAM, a search of the FORTRAN library, SRF7, automatically takes place in order to satisfy cues in the compiled FORTRAN segments or in any semicompiled segments that have also been supplied to the compiler.

In general, however, it is advisable to arrange for a search of the SRF7 block to take place immediately following the search of any private libraries and before searching ICL written libraries (for example the S-RS BLOCK). Thus the recommended order is as follows:

1 Search private libraries containing segments written by the programmer or installation.

2 Search the FORTRAN library SRF7.

3 Search any other library containing ICL written segments required (for example, S-RS block).

4 Search the FORTRAN library again (this is automatic on reading the FINISH statement).

If step 3 is omitted, step 2 is not required. If step 1 is omitted, step 2 is still advisable as otherwise wrong versions of certain routines may be included (for example if SRA1 is scanned before the SRF7 groups then the Algol version of SIN will be included).

Example
If a source program contains calls to private routines in a library, which in turn call routines in the S-RS block, the directives should be:

LIBRARY

READ FROM (MT, *library name. subfile name*)

READ FROM (MT,–.SRF7

READ FROM (MT,–.S–RS)

FINISH

ACCEPTANCE RULES

The compiler can input any mixture of source and semi-compiled segments from any medium, except that the program description segment must always be in source form.

During compilation of the main program, any segment is accepted and incorporated into the program as long as no segment of the same name has previously been encountered. When two segments have the same name, the first is accepted and the second ignored.

When a library is being read (after a LIBRARY statement) or when the standard library is being read (as the result of a FINISH statement) only those segments that have.been called for in a previous segment are incorporated.

There may be some qualifications to these rules if the program incorporates segments in other languages (see Chapter 9).

Chapter 7 Program testing

COMPILATION ERROR DIAGNOSTICS

During compilation of a FORTRAN program, a partial or complete list of source program statements is output on the listing peripheral. When an error is found, an error message is output. If a full listing has been specified, the error message follows the complete statement. In a short listing, the error message is preceded by the first line only of the statement in error; the error could be in a continuation line. The detection of an error causes semi-compiled output to cease; the compiler continues so that further errors may be detected. A list of compilation error numbers is given in Appendix 1.

Error messages

The format of the error message is normally:

> ERROR *n* IN LAST STATEMENT, LINE *p*, CHAR *q*
> *n* is the error number
> *p* is the line number within the statement (starting at line 0)
> *q* is the character number within the line (starting at character 1)

The error is mostly likely to have occurred at or before the position specified by *p* and *q*.

The occurrence of one error may lead the compiler to produce further spurious errors later in the program. Consider the following sequence

```
DIMENSION A(10), Z(5, 70), AMISH (7,10*, LOOK (5)
..........
READ (5,500) A, LOOK
..........
WRITE (6,600) A (1) LOOK (3)
```

The character) has been mispunched as *. The DIMENSION statement will be faulted because of the *; the compiler will not register LOOK as an array but will treat it as a variable in the READ statement; then in the WRITE statement an error will be noted because a subscript is attached to a variable. The reasons for such spurious errors are not always easy to find. If the cause of certain error lines cannot be ascertained, then obvious errors should be corrected and the program recompiled; any spurious errors will then disappear.

LABEL ERRORS

Errors 15 and 16 occur when a label referred to in a segment is not present. These errors will not be found until the end of the segment is reached. Messages of this type have a different format:

> ERROR *n* - LABEL *m*

where *m* is the label not found. Messages referring to label 0 should be ignored as they only indicate that an error has already been found in some statement that refers to a label.

MISSING SEGMENTS

If some segments of a program are not input to the compiler, a loadable program is still produced but the message SEGMENTS MISSING will be output on the listing. An incomplete program should be run only if it is known that the missing segments will not be entered on that run. Any attempt to actually enter a missing segment will cause an error.

OBJECT PROGRAM ERROR DIAGNOSTICS

The discovery of the causes of run-time error conditions can be an expensive process, involving wastage of machine time and programming effort. For this reason it has become standard practice to incorporate monitoring routines into programs; these routines carry out tests during run-time and output details of any error conditions that occur. All compiled FORTRAN programs contain monitoring routines and are said to be *traced;* the necessary routines are

incorporated by the compiler and no extra programming is required.

The advantages of tracing are obtained at the cost of increased run-time and core store requirements. The amount by which these two factors are increased is determined by the level of tracing used; i.e. by the comprehensiveness of the tests carried out by the monitoring routines. The higher the level of tracing the longer will be the run-time and the greater will be the core store occupied.

The level of tracing that it is desirable to incorporate into a program is determined by the relative importance of the two conflicting requirements: run-time efficiency and ease of error finding. During the early stages of the development of a program ease of error finding will be the important requirement and a high trace level will be desirable. When the program has been fully tested and proved, a lower trace level may be used with a corresponding increase in run-time efficiency. To provide for this variation three levels of tracing are available, 0, 1 and 2, being low, medium and high levels respectively. It is envisaged that trace level 2 will be used during program development and trace level 1 during normal production runs. Trace level 0 will be used only for well-established programs and when core store and execution time are at a premium.

Trace output channel
Trace information will normally be output on the first output channel specified in the program description. The user should ensure that this channel will accept formatted output, as trace information must not be output to an unformatted magnetic tape file, for instance. The trace output channel may be used for other, non-trace information, as the user chooses. The user may output non-trace information on the trace output unit, without knowing its channel number, by using the channel number 0 in WRITE statements.

Example

```
PROGRAM (JIM5)
INPUT    9 = TR0
OUTPUT   3 = LP0
OUTPUT   5 = TP1
END
```

The trace information would be output on the line printer, also used as the programmer's unit 3.

Trace level 1
Level 1 is the normal mode of tracing and is automatically used unless the programmer includes a TRACE statement in the program description (see below), specifying a different level. The effects of tracing at level 1 are described below.

NORMAL ERROR ACTION
The occurrence of a detectable error causes the following message to be output on the trace output device:

EXECUTION ERROR *n* PROGRAM TERMINATED

where *n* is an error number. A full list of error numbers is given in Appendix 2. This line is followed by a list of the last 25 segment entries and exits under the heading SEGMENT TRACE. An entry to a segment is marked by the first eight characters of the segment name and an exit is marked by RETURN. In this context, the term 'segment' means a segment compiled from FORTRAN source, a Basic External Function or any PLAN segments incorporated in a FORTRAN program using FPROLOG and FEPILOG. (See Chapter 11, *Mixed language programming.*) The program then halts, displaying the characters EE on the console typewriter.

OVERFLOW
Overflow may occur if the result of an arithmetic expression or the result of evaluating part of an arithmetic expression is out of range, or if the program attempts to carry out some mathematically undefined action such as division by zero. Overflow may also occur during the evaluation of a logical expression that compares two arithmetic expressions if the difference between the two is out of range; this is most likely to occur during the evaluation of logical IF statements.

If overflow occurs, an overflow indicator is set and the program continues. A test on the state of this overflow indicator is made prior to an entry to or an exit from every compiled subroutine or function segment and every Basic External Function. If the overflow indicator is set at such a point, execution error number 50 is output.

USER'S CHECK

A subroutine is provided in the compiler's library that enables the user to check the state of overflow in his program. The subroutine, called OVERFL, requires one integer argument and is called in the standard way:

```
CALL OVERFL (j)
```

j may be any integer variable or array element. The actual argument is assigned the value 1 if overflow is currently set, or the value 2 otherwise. The subroutines leaves overflow clear. This subroutine is provided chiefly to allow programs written for another system to be run on a 1900 machine. It may also be used to test for overflow more frequently than the automatic facility of trace level 1.

THE ERROR TRAP FACILITY

The user may augment the standard trace facilities by providing a subroutine to be entered whenever an error is discovered by the trace routines. Such a subroutine is called an *error trap*. An error trap may be written to output specialized information chosen by the user rather than by the system. Alternatively, in certain cases the subroutine may be written to carry out some remedial action enabling the program to continue.

An error trap is set by calling the library subroutine FTRAP with the user's error subroutine as an actual argument. Thus, if the user's subroutine is called PMORTM, the user would write:

```
      EXTERNAL PMORTM
      ..........
      ..........
      CALL FTRAP (PMORTM)
```

The subroutine FTRAP need be called once only in a program and will normally be called at the beginning of the program.

The subroutine PMORTM must be written with one dummy argument of type INTEGER. When an error is detected, the subroutine will be entered with this argument set equal to the error number as listed in Appendix 2. (There is one special case dealt with below.) Thus, the subroutine PMORTM might be written.

```
      SUBROUTINE PMORTM (ERR NO)
      INTEGER ERR NO
      COMMON/ABCD/ARRY(10),X,Y,Z
      WRITE (0,101) ERR NO, ARRY,X,Y,Z
  101 FORMAT (1H  , 6HERROR  , I2/1H  , 10E10.4/1H  , 3E10.4)
      STOP
      END
```

This subroutine outputs the error number, followed by the current value of an array and three variables, on the trace output channel (i.e. channel0).

If the error consists of an unacceptable character in a numeric input field (i.e. corresponding to error 0 '*x*' as listed in Appendix 2) then the argument of the error subroutine is not set to an error number, but is set to the negated internal representation of the character; i.e. if the argument is negative, it should be made positive and its text value will then be 4H000*x*, where *x* is the erroneous character. The above example could be expanded to take care of this as follows:

```
      SUBROUTINE PMORTM (ERR NO)
      INTEGER ERR NO
      COMMON/ABCD/ARRY(10),X,Y,Z
      IF (ERR NO.GT.0) GO TO 5
      ERR NO = - ERR NO
      WRITE (0,100) ERR NO,ARRY,X,Y,Z
      STOP
    5 WRITE (0,101) ERR NO,ARRY,X,Y,Z
  100 FORMAT (1H , 7HERROR' ,A4, 1H'/1H,10E10.4/1H ,3E10.4)
```

```
101 FORMAT (1H  , 6HERROR  , I2/ 1H,  , 10E10.4/1H  , 3E10.4)
    STOP
    END
```

If the programmer requires the program to continue after an entry to the error trap subroutine, the subroutine can end with a RETURN statement rather than a STOP statement. For non-fatal errors, control will then return to the point in the program where the error was detected and the operation that caused the error will have a meaningless result. For fatal errors it is not possible for the program to continue and a RETURN statement will cause the program to output the standard post mortem information and halt. The distinction between fatal and non-fatal errors is given in Appendix 2. The user may set up a different error trap at any point in the program by calling FTRAP with a different argument. Alternatively the standard error procedure may be reinstated by calling the subroutine FRESET as follows:

```
CALL FRESET
```

This subroutine cancels any previous call to FTRAP.

Trace level 2

Trace level 2 is the highest level of tracing available. All trace level 1 facilities are also provided in trace level 2; they will not be described again in this section.

Trace level 2 is incorporated by including in the program description the statement:

```
TRACE 2
```

ERROR ACTION

The post mortem information output on the detection of an error is the same as for trace level 1 with the addition of a STATEMENT TRACE, which gives details of the last 100 statements executed preceding the error.

STATEMENT TRACE output

The STATEMENT TRACE list is output following the standard error line and contains a line for each of the 100 statements examined. Each line contains the following details in the order shown:

The line number, in the range -100 to -1 inclusive.
The statement label, if one is present.
An abbreviation of the statement type.
The result of the statement, if applicable.

The end of a DO loop, although not written as a statement in the source program, is treated as a statement (type LOOP).

The statements, their abbreviations and their results are shown in Table 2 on page 31.

Whenever control has been passed to a new segment, the name of the segment is printed on a separate line. In the case of an overlaid object program (see page 37) these names will not be correct if the segment entered has subsequently been overlaid by a segment in another unit of the same overlay area.

Statement	*Abbreviation*	*Result*
Arithmetic assignment	ARTH	Value of left-hand side
Logical assignment	LOCC	Value of left-hand side: TRUE or FALSE
Arithmetic IF	IF	Value of expression in parentheses
Logical IF	LIF	Value of logical expression: TRUE or FALSE
GO TO	GO	
DO	DO	Initial parameter
	LOOP	(terminal parameter) - (control variable)
Computed GO TO	CGO	Control variable
Assigned GO TO	AGO	
READ	READ	
WRITE	WRTE	
PAUSE	PAUS	
STOP	STOP	
CALL	CALL	
RETURN	RETN	
ENDFILE	ENDF	
CONTINUE	CONT	
BACKSPACE	BACK	
REWIND	REW	
ASSIGN	ASGN	

Table 2 Statements in a STATEMENT TRACE list

ARRAY ELEMENTS

The start and end addresses of each array are held in store at object program run time. At trace level 2 a check is made that the generated address of an array element falls within the bound of the array. No check is made on individual subscripts. For example, if ARRAY(3,2) is declared, then a reference to ARRAY(5,5) would be faulted. However, ARRAY(4,1) would not be detected as an error. The corresponding error number is 01. (See Appendix 2.)

TRACE 2 WITH STEERING LISTS

It may be desirable to obtain detailed trace information about selected parts of a program. If the program was compiled at trace level 2 it is possible to selectively output information about parts of the program by specifying these parts in a *trace steering list.* The trace steering list is read from the trace input channel, usually a card or paper tape reader, at the start of the program run. The trace input channel is the first input channel specified in the program description. The user must ensure that it is capable of inputting formatted data.

Example

```
PROGRAM (FRED)
INPUT    5 = TR0
INPUT    7 = CR0
OUTPUT   6 = LP0
TRACE 2
END
```

This program description will cause the steering list to be read, at run time, from a tape reader allocated as TR0. The steering list would be input from this unit before any data associated with the programmer's channel 5.

FORMAT OF TRACE STEERING LISTS

A trace steering list consists of one or more lines, each of which specifies a continuous sequence of statements, within a segment, that is to be traced. Each sequence of statements is defined by specifying the beginning of the sequence relative to a *trace triggering statement,* which must be labelled, and the length of the sequence.

Each line of a trace steering list has the following format:

$l \quad d \quad c$

l is the label of a statement that is to be the trace triggering statement for the traced statements. The same label must not be listed more than once. d, where $=99 \leqslant d \leqslant 2047$, indicates at which statement, relative to the triggering statement, tracing is to commence. For example, if d is 50, trace information is output when 50 statements following the triggering statement have been executed.

It should be noted that d does not refer to the number of statements *written* after the label, but to the number that have actually been executed. If d is 0 trace output starts with the triggering statement itself. Negative values of d indicate that statements executed before the triggering statement are to be traced; however, no trace information is actually output until the triggering statement itself is executed.

c, where $0 \leqslant c \leqslant 4095$, is the total number of statements to be traced for that triggering statement.

If there is an overlap between the ranges specified by two steering lines, fewer than c lines may be output, since the output from the first statement will cease when the second triggering statement is encountered. If the value of d for the second triggering statement is negative, and overlaps the scope of a previous triggering statement, some information may be repeated.

The format of a trace steering list on a FORTRAN coding sheet is as follows:

Segment name $l_1 \quad d_1 \quad c_1$ $l_2 \quad d_2 \quad c_2 \qquad c_2$ $l_3 \quad d_3 \quad c_3$ . . . . $l_n \quad d_n \quad c_n \quad c$ END FINISH	Repeated for each segment

The *segment name*, END and FINISH must begin in or after column 7.

Columns 73 to 80 are ignored and column 6 should be blank.

l should be written in columns 1 to 5.
d should begin in column 7.
c should begin in column 13 or be preceded by a tab character.

No spaces are allowed in c and d. Tabs are allowed on paper tape in the same way as in source program.

A STATEMENT TRACE list is output for each series of statements specified in the trace steering list. Line numbers are consecutive in ascending order, in the normal manner. The trigger statement line is always numbered 0; line numbers for statements executed before the trigger statement are given negative numbers.

TRACING EVERY STATEMENT

The user can obtain trace information for every statement that has been compiled with trace level 2; without specifying a trace steering list, by activating the program at entry point 27. (See the section *Loading and running the object program* in Chapter 15 on page 87).

Suppression of trace output

Trace output during the running of a program may be suppressed by switching off Switch 0, either by a call to the subroutine SWOFF or by operator action. The normal error information will still be output if an error occurs while Switch 0 is switched off.

Trace level 0

If core store is at a premium, a zero level of tracing is available that saves about 200 words of core store compared with TRACE 1. The limited facilities provided from a subset of the trace level 1 facilities. On the detection of an error, the standard error line is output giving the error number. However, no SEGMENT TRACE list follows. There is no user error trap facility and no test on the state of the overflow register. The detection of any error always causes the program run to be terminated. Trace level 0 may be requested by writing, in the program description, the statement

```
TRACE 0
```

Mixed error detection levels

During development of a program, it may be convenient to run part of it at level 1 and part at level 2. This can be done by specifying TRACE 2 in the program description and then TRACE 2 or TRACE 1 between program segments as required. Each segment is compiled at the level set by the previous TRACE statement. A trace steering list must not refer to a segment that has been compiled at level 1. Statement lines output under the heading STATEMENT TRACE will include only those statements traced at level 2.

Operator intervention

The operator may intervene at program run time and cause tracing information to be output. The normal reasons for this are:

1 The program has exceeded its time limit.

2 Executive has detected an illegal machine code instruction causing an ILLEGAL message to be printed on the console typewriter.

The trace information output takes the same form as if an error had been detected by a trace routine in the program except that no execution error line appears. Details of the necessary console messages are given in the section *Loading and running the object program* in Chapter 15 on page 90.

Example of error detection in a program with an arithmetic error

The following program EREX reads a list of numbers J_i, A_i, B_i (i = 1, 2,.....n) into the blank COMMON block. A check is made to set $A_i = -1$ if $\sqrt{(A_i - B_i)} < 3.142$. Subroutine PROCESS is then called to operate on these numbers (the action of the subroutine is immaterial to the example).

```
        LIST
        PROGRAM (EREX)
        OUTPUT  2 = LP0
        INPUT   1 = TR0
        TRACE 2
        END

        MASTER READ
        COMMON J(10), A(10), B(10), N
2       READ (1,100) N
        IF (N.EQ.0) STOP
        WRITE (2,200) N
        DO 1 I = 1, N
        READ (1,101) J(I), A(I), B(I)
        IF (SQRT(A(I) - B(I)).LE.3.142) A(I) = -1.0
1       CONTINUE
        CALL PROCES
        GO TO 2
100     FORMAT (I2)
101     FORMAT (I2, 2F6.1)
```

```
200     FORMAT (14H1PROGRAM #EREX//13H NO OF VALUES, I4//)
        END
        SUBROUTINE PROCES
        ..........
        END
        FINISH
```

The following output on the line printer shows what may happen if no error trap is used and there is an error in the input data. Error 57 is reported (negative argument in SQRT), STATEMENT TRACE shows how far the program has gone, and SEGMENT TRACE shows that SQRT was the last routine called.

```
PROGRAM #EREX                                           } Output by the program
NO OF VALUES 8                                          }
EXECUTION ERROR 57 - PROGRAM TERMINATED                   Output by TRACE
STATEMENT TRACE
READ
-22          2  READ
-21             LIF    FALS
-20             WRITE
-19             DO                     1
-18             READ
-17             LIF    FALS
-16          1  CONT
-15             LOOP                   6
-14             READ
-13             LIF    FALS
-12          1  CONT
-11             LOOP                   5
-10             READ
- 9             LIF    TRUE
- 8             ARTH   -1.00000000000E 0
- 7          1  CONT
- 6             LOOP                   4
- 5             READ
- 4             LIF    FALS
- 3          1  CONT
- 2             LOOP                   3
- 1             READ
SEGMENT TRACE
SQRT
RETURN
SQRT
RETURN
SQRT
RETURN
SQRT
RETURN
SQRT
```

Example of error trap facility

An error trap subroutine is incorporated in the program. The program description is as in the previous example. The MASTER segment begins:

```
      MASTER READ
      COMMON J(10), A(10), B(10), N
      EXTERNAL ERROR
      CALL FTRAP (ERROR)
    2 READ etc. as in the previous example
```

The error subroutine is

```
      SUBROUTINE ERROR(K)
      COMMON J(10), A(10), B(10), N
      WRITE (2,100) K, (J(I), A(I), B(I), I = 1,N)
      RETURN
C     RETURN IN ORDER TO CHECK REST OF DATA
  100 FORMAT (// 6H ERROR, I3, 9H DETECTED // 22H VALUES IN COMMON AREA
     1/// (3X, I2, 2F6.1/))
      END
```

The output consists of the following on the line printer:

```
PROGRAM #EREX                                   } output by the program
NO OF VALUES 8                                  }
ERROR 57 DETECTED                                 output by subroutine ERROR
VALUES IN COMMON AREA
      74            153.1       78.6
      13            102.8       84.7
      25             - 1.0      97.1
      31            177.9       53.8
      65             94.5       98.4
       0              0.0        0.0
       0              0.0        0.0
       0              0.0        0.0
```

The program continues with the rest of the data.

Chapter 8 Overlays

This chapter describes the overlay system which may be used in a 1900 FORTRAN program using magnetic tape.

THE OVERLAY SYSTEM

Programs may be written which are too large to be held in the core store of the computer. Such a program is run by *overlaying* parts of the program, from a backing store medium That is, only part of the program is in the core store at a particular time, the remainder being held on the backing store and copied into the core store whenever it is required.

To use the overlay system, the programmer must divide his program into units, one unit which is not to be overlaid and a number of *overlay units.* The section of program which is not to be overlaid is called the permanent unit and is kept in the *permanent area* of core store. It contains the MASTER segment and any other segment that the programmer chooses. The permanent unit will also contain various subroutines that are inserted into a FORTRAN program by the compiler and not referred to explicitly by the programmer (e.g. the input and output subroutines). Each overlay unit of the program must be assigned to an *overlay area* of core store. There may be several overlay areas and each one may have several overlay units assigned to it. At any one time an overlay area may contain only one of the units assigned to it.

The structure of an overlaid program is defined in OVERLAY statements in the program description. This type of statement is described below.

1900 FORTRAN does not require the user of the overlay system to call any special subroutines. The source program is written in the normal way and segments are called by CALL statements or function references exactly as if the whole program was to be in the core store.

If the segment called is in an overlay unit, the system checks to see whether that unit is already in the core store. If it is not, it will be copied in from the backing store, then entered normally. A RETURN statement in the segment will have a similar effect, checking to see whether the segment containing the calling statement is in the core store, and copying it in if it is not. Decisions on overlay organizations are not, therefore, required at the time of writing a source program. This organization need not be defined until the program is about to be compiled.

PROGRAM DESCRIPTION STATEMENTS FOR OVERLAY PROGRAMS

The OVERLAY statement

The form of this statement is as follows:

OVERLAY $(a, u) s_1, s_2, s_3 ....s_n$

a is the overlay area number in the range $1 \leq a \leq 255$.
u is the overlay unit number in the range $1 \leq u \leq 1023$.
$s_1, s_2, s_3, ...s_n$ is a list of the names of segments to be included in unit u of area a.

The numbers used for a and u need not be consecutive. The same unit numbers can be used for units in different areas. Since the OVERLAY statement is a program description statement, no continuation lines are possible. However, if the same area and unit numbers appear in more than one statement, then the unit will comprise all the segments named in all such statements. For example, the statements:

```
OVERLAY (3, 7) SUBA1, SUBA2, FUNC7
OVERLAY (3, 7) SUBB3, SUBB5
```

will have the same effect as:

```
OVERLAY (3, 7) SUBA1, SUBA2, FUNC7, SUBB3, SUBB5
```

All segments that have not been assigned to an overlay unit will be assigned to the permanent unit. The permanent unit is not specified explicitly by the programmer.

The OVERLAY PROGRAM and OVERLAY SEGMENTS statements
The OVERLAY PROGRAM statement must be used in place of a PROGRAM statement in order to introduce an overlay program. It has the following form

OVERLAY PROGRAM (*name*)

The OVERLAY SEGMENTS statement is used instead of a SEGMENTS statement in order to introduce segments to be compiled which may later be included in an overlay program. It has the following form

OVERLAY SEGMENTS (*name*)

where *name* is as above. The area and unit number of each overlay segment is not required until the complete program is compiled and therefore OVERLAY statements should not appear in a program description introduced by OVERLAY SEGMENTS.

OVERLAY PROGRAM may be used to introduce a program with no overlays if it is wished to include segments which were compiled under an OVERLAY SEGMENTS statement, but a slight loss in core store usage is involved in doing this. This situation of overlay segments in a non-overlay program would arise if a library of segments is required for use in either overlay or non-overlay programs.

The DEPTH OF OVERLAY statement
At run time, when a segment A of an overlay program in one unit calls a segment B in another unit, the following three actions take place:

1 The new overlay unit is brought into core store (if not already there).
2 Control is transferred to segment B.
3 A record is kept of the unit and area numbers of segment A so that when a return is made to segment A then the correct unit can be brought in.

This situation becomes more complex if B calls a further segment C in another unit (possibly even the unit that segment A is in). In this case records of the unit and area numbers of A and B must be kept. This process can go much further in some programs and a list of these records must always be kept.

The space reserved by the compiler to contain this list of records must be sufficient to contain the maximum number of segment entries required. The size of the list is specified by means of a DEPTH OF OVERLAY statement. The *depth of overlay* is defined to be the maximum number of calls (or function references) which can be obeyed *in sequence*, and which each transfer control to a new overlay unit. A record is kept in the list for each of these calls. Here 'unit' includes the permanent unit.

The statement

DEPTH OF OVERLAY *n*

in the program description, where *n* is an integer, implies that the list will contain a maximum of *n* records i.e. the statement will reserve storage for *n* entries in the list. In the absence of the DEPTH OF OVERLAY statement, *n* is set equal to the total number of units in the program. This is usually an over estimate so that the statement is obligatory only in the case of highly segmented programs. On the other hand, it is useful to save core store by specifying an appropriate length for the list when this is very much less than the number of units.

Examples

1 A program contains 40 overlay units and one overlay area. All calls to overlay units are made from the permanent unit and an overlay unit does not call another overlay unit, but may call standard functions in the permanent unit. The programmer may save the space of 38 entries in the list of records by including the statement

   DEPTH OF OVERLAY 2

   in the program description.

2 A program contains two overlay units and two overlay areas. A segment in the permanent unit calls a segment 1 of unit A, which calls segment 1 of unit B, which in turn calls segment 2 of unit A etc. There are 20 calls in all between unit A and unit B. Segments in units A and B call standard functions which are held in the permanent unit. In order to reserve enough store to hold the list of records the programmer *must* include the statement

   DEPTH OF OVERLAY 22

   in the program description.

Restrictions

The following restrictions must be observed in overlay programs:

1 A CALL or function reference must not bring down another unit to overwrite the unit containing the CALL or function reference, whether this is done directly or indirectly via a chain of CALLS or function references. There is one exception: a CALL with no arguments may bring down a unit to overwrite itself. In this case, however, a RETURN may not be made unless the calling segment also has no arguments.

2 No subroutine or function whose name is listed in an EXTERNAL statement may be in an overlay unit.

3 No segment name may appear in two OVERLAY statements.

4 MT0 must not be used by an overlay program.

It should be noted that non-common variables, array elements, and arrays will be overlaid if assigned values in DATA statements. These variable array elements and arrays will be reset to the specified values each time the overlay unit is brought in.

COMPILATION

The structure of an overlay program must be defined in the program description each time the program is compiled, whether or not the structure has changed. If the structure is to be different from the previous compilation it is possible to re-compile by providing a new complete program description and re-inputting the semi-compiled program produced by the previous run (see Chapter 6, *Input from simple files*, page 23).

RUN TIME EFFICIENCY

The most efficient utilisation of available core store will be obtained if each overlay unit associated with a particular overlav area is approximately the same length. It is not possible to estimate exactly the number of machine code instructions that will be generated from a FORTRAN source segment, but during compilation the compiler outputs to the listing peripheral a line at the end of each segment, which specifies the program area used by the segment. To this must be added the space taken up by constants in DATA statements. (Note that if a single element of an array occurs in a DATA statement, the whole space of the array is included in the size of the overlay unit.)

Lastly, the structure of overlays should be designed so that units seldom need be interchanged.

STANDARD FUNCTIONS IN OVERLAYS

Standard functions and subroutines will normally be in the permanent area. The programmer can, if he wishes, assign a standard function or subroutine to an overlay unit.

If some standard functions are required only by segments in one overlay unit, then core store is saved if subroutines and functions are inserted in the unit where they are required.

EXAMPLE OF AN OVERLAY PROGRAM

A program with the following structure is to be overlaid.

The MASTER segment calls subroutines S1, S2, S3 in that order.

In addition subroutine S3 calls functions S4 and subroutines S5, S6 in that order.

It is assumed that the MASTER segment and each subroutine are of roughly the same length and only enough core store is available for the MASTER segment and two subroutines.

In this case the program could be organized as follows:

```
LIST
DUMP ON (PROGRAM ABCD)
OVERLAY PROGRAM (ABCD)
OVERLAY (1, 1) S1
OVERLAY (1, 2) S2
OVERLAY (1, 3) S3
OVERLAY (2, 1) S4
OVERLAY (2,2) S5
```

```
OVERLAY (2,3) S6
OUTPUT  1 = LP0
USE     2 = MT1
INPUT   3 = TR0
END
```

OVERLAYING FROM DISC

Overlay programs compiled by #XFAM are normally overlaid from magnetic tape during execution. If it is required to overlay the program from disc, the following statement must be included in the source program (a suitable place is immediately after the MASTER statement):

```
EXTERNAL OLAYDISC
```

The object program will be produced on magnetic tape (as specified in a DUMP ON statement). Before the program can be run it must be transcribed to disc using the program #XPEU, described in the manual *Library Specifications* (TP 4011).

Chapter 9 Miscellaneous features

COMPACT AND EXTENDED DATA PROGRAMS

1900 FORTRAN may be used on medium and large processors in the 1900 series. Medium processors are defined as having not more than 32768 words of core store; large processors normally have at least 65536 words of core store. Object programs that are to utilize fully the core stores of large processors must be *extended data* programs; these programs cannot be run on medium processors. Other programs are known as *compact* programs; compact programs may be run on medium or large processors; in the latter case the program area may not occupy more than 32768 words of core store.

A program that is to be compiled and run on a large processor may contain any amount of data, subject to the limitations of the core store, in the form of arrays or COMMON variables. The core store required for holding other parts of the program must not exceed 32768 words; no individual COMMON block may be longer than 32768 words, (i.e not more than 16384 REAL variables).

Compiling and running on different size processors

The compiler will normally produce a compact program on a medium processor and an extended data program on a large processor. The following compiling system statements allow for running or re-compiling on different sized processors.

EXTENDED DATA

COMPACT

MIXED SEGMENTS

If a program is compiled on a medium processor for running on a large processor it must contain the EXTENDED DATA statement in the program description. However, this statement may be omitted if the program will not occupy more than 32768 words of core store.

If a program is compiled on a large processor for running on a medium processor it must contain the COMPACT statement in the program description.

Mixed language programs

This section provides additional information needed only by users wishing to incorporate PLAN subroutines into a FORTRAN program or otherwise to mix languages. (This includes reading from other libraries.) Chapter 11 describes mixed language programming in detail.

Compilers mark individual semi-compiled segments to indicate that they can either run only in a compact program, run only in an extended data program or run in either type of program. Those segments which can be run in either type of program are marked as 'either' by the respective compiler.

FORTRAN, Algol and EMA produce segments marked as capable of running in either type of program. For a PLAN segment it is up to the programmer whether or not the segment will work in compact programs or extended data programs. Segments produced by all PLAN 1, 2 and 3 compilers are marked as compact segments. PLAN 4 compilers allow the user to specify how the segments should be marked.

The consolidator will check the mode of all semi-compiled segments against the mode of the program. Segments marked 'either' will be accepted under the usual conditions. Compact segments in an extended data program or extended data segments in a compact program will normally be faulted; however, if the segments appear in a subroutine library they will be ignored. It is thus possible to have alternative versions of the same subroutine for working in extended or compact programs.

The programmer can override the checks described above by including the statement

MIXED SEGMENTS

in the program description. Segments are then accepted regardless of their mode. The main reason for including this statement is to allow segments which will work in extended data programs to be included even though they are marked as compact only.

PRIORITY

Some of the larger machines in the 1900 Series are multiprogramming; that is, several programs can be in core store at any one time. When the execution of one program is suspended whilst a peripheral transfer is taking place, Executive will continue with another program. This program may in turn be suspended and then control will be passed to a third program and so on. This system permits the simultaneous use of a large number of peripherals at maximum efficiency, as well as permitting the most effective use of the central processor. To facilitate the organisation of time sharing between programs, each program is given a priority, which indicates the ratio of peripheral activity to actual computing time. A program with high peripheral activity comparing to actual computing time requires a high priority and vice versa. A priority is assigned to a program by including in the program description a statement of the form

PRIORITY *n*

where *n* is an integer in the range 1 to 99. If this statement is omitted, a priority of 50 is assumed. The statement will have no effect on a single programming machine.

THE OMIT STATEMENT

If a program is under test, some segments may not be required when the object program is run; some may not even be written. Segments not required can be omitted at the consolidation phase by including the following statement in the program description

OMIT *seg 1, seg 2, ... seg n*

seg 1, seg 2, ... seg n is a list of the names of segments not to be consolidated into the object program. The segments may or may not be presented to the compiler. The program will be terminated if a call to a missing segment is encountered at run time.

THE COMPRESS STATEMENTS

INTEGER AND LOGICAL statements

This statement may be written anywhere in the program description as follows:

COMPRESS INTEGER AND LOGICAL

It causes one word of core store instead of two to be assigned to each INTEGER and LOGICAL variable or array element used in the source program. The range of INTEGER quantities is not affected, and any operation on INTEGER quantities will yield the same results. Two words are used normally only for compatibility with other versions of FORTRAN. The user is normally recommended to use this statement only when store shortage makes it essential or when some segments in a program are written in another language and these segments require compressed FORTRAN INTEGER quantities.

In COMMON, EQUIVALENCE and DATA statements it is advisable to match items of the same type; otherwise erroneous results may be obtained when a COMPRESS statement is used. For example, in one segment the statement

COMMON/MAT/A,B(100), I(10,10), C

appears and in another

COMMON/MAT/ING, BB(100), J(10, 10), CC

If both segments are compiled without the COMPRESS statement; ING and A will occupy the same storage; as will B and BB, I and J, C and CC, whereas if both segments are compiled in COMPRESS INTEGER AND LOGICAL mode, the variables and arrays following A and ING will no longer correspond. If the common area is being used to pass values from one segment to the other, incorrect results will be obtained.

Semi-compiled segments presented to the compiler are not affected by COMPRESS statements.

A side effect of the COMPRESS INTEGER AND LOGICAL mode of compilation is that all TEXT constants used in the program are rounded up by the addition of spaces to the next multiple of four characters, instead of eight, even if TEXT quantities are to be held in REAL, DOUBLE PRECISION, or COMPLEX variables and array elements.

DOUBLE PRECISION statement

The statement

COMPRESS DOUBLE PRECISION

included anywhere in the program description, causes all DOUBLE PRECISION variables and array elements used in the source program to be treated as type REAL. This limits the accuracy but does not change the range of DOUBLE PRECISION variables and array elements. When a compressed DOUBLE PRECISION variable or array element is to contain TEXT information, the maximum number of characters that can be stored is 8 instead of 16. Care should be exercised when using COMMON, EQUIVALENCE, and DATA statements. Functions that require DOUBLE PRECISION arguments may still be used but the accuracy obtained may be slightly less than that obtained for a REAL variable. This is because two arbitary words will be taken to lengthen each argument to the required four words. The inaccuracy caused will be so slight as to be insignificant in the majority of cases. DOUBLE PRECISION working uses more core store and takes considerably longer than REAL working, even on machines with floating point hardware. If data is not accurate enough to justify DOUBLE PRECISION working, or if highly accurate results are not required, the COMPRESS statement can be used to reduce run time in a program that normally requires DOUBLE PRECISION working. Similarly, when a program is under test, the COMPRESS statement can be used to reduce run time and subsequently be removed for production runs.

This statement is of use when programs written for other machines and using DOUBLE PRECISION are to be run on 1900 Series machines, and the accuracy of the REAL working of the 1900 is found to be adequate. The COMPRESS statement is inserted once, and saves changing each type statement.

Chapter 10 Editing

When a source program has been written, it will normally be punched on to paper tape or cards. It could then be input directly to the FORTRAN compiler, and the object program obtained ready for running on the computer. However, it will usually be necessary to make several changes to the program and to recompile it several times before it is finalized. This could be done by repeatedly amending the source cards or paper tape, and re-presenting them to the compiler, but such a system has the following disadvantages:

1 Repeatedly reading input from a basic peripheral is very wasteful of valuable computer time.

2 Program manipulated on paper tape or cards are bulky and prone to accidental damage.

These disadvantages can be overcome by a system in which the program is transferred from paper tape or cards to magnetic tape backing store. Any necessary changes to the program are then made by amending it on the magnetic tape file. After the inital transfer to magnetic tape, and after each amendment, the program may be input to the compiler from the magnetic tape file at high speed.

Programs which perform the tasks of transcription and amendment are called *Editors*. In addition to their main functions described above, Editor programs commonly provide other useful facilities, such as listing of programs, and production of basic peripheral copies of programs for use as reference documents.

The editor program normally used for editing FORTRAN programs held on magnetic tape files is #XMUM. This program can be used to create a batch of FORTRAN programs on a magnetic tape file, and subsequently to update the file by adding, deleting or amending the programs. If the file is created, and is always updated, by #XMUM, the user need not be concerned with the details of magnetic tape file formats; usually he refers to each program by name, and #XMUM automatically handles the file structure.

If there is a magnetic tape file of FORTRAN programs that does not observe the same conventions as #XMUM files, and it is required to update this file, the editor program #XKYA must be used. Both #XMUM and #XKYA edit magnetic tape files which conform to the 1900 Series composite file structure as described in the manual *Magnetic Tape* (TP 4091). Subfiles containing source program may be added, deleted or amended by #XKYA, but it is necessary for the user to understand the structure of the composite file, since, for example, he must specify the sentinels to be set up in the file. However, certain facilities, such as merging of subfiles and input of subfiles from other magnetic tape files are available only with #XKYA.

The editor programs #XMUM and #XKYA are described below.

THE EDITOR PROGRAM #XMUM

General

The editor #XMUM is used to create, and subsequently to update, jobs on a magnetic tape file. A *job* consists of a subfile containing a source program optionally followed by other subfiles containing any associated semi-compiled segments, a steering list and data. Each job normally constitutes a program, but it is also possible to compile two or more jobs together as one program. The source segments in a file used by #XMUM may be written in FORTRAN, Algol or EMA, but this Chapter will only consider the facilities relevant to FORTRAN.

COMPILATION

After a file of jobs has been created by #XMUM, an attempt will normally be made to compile and run the programs comprising the file by use of #XFAM. The facilities for compiling a batch of programs are described in Chapter 14; the file named in the BATCH FROM statement (page 82) will be the file created by #XMUM.

If errors are discovered in the compilation or running of any of the jobs on the file, the appropriate lines of source program may be altered in an update run of #XMUM and the updated file recompiled. Facilities are also available for creating a selective output file in an update run, so that only those jobs which have been altered need to be recompiled. The selective output file instead of the updated file would be named in a BATCH FROM statement.

Alternate runs of #XMUM and #XFAM would follow until all the programs on the file are correct. Individual programs from a file created by #XMUM can be compiled by using the file name and subfile name in READ FROM statements (see page 22).

WORK ASSEMBLY

The programs held on the file used by #XMUM may belong either to one programmer or to a number of programmers. In the latter case a *work assembler* will be required to control any editing and compilation performed on the batch of programs which make up the file. His role will be to collect the editing requirements of the programmers and to submit them to the editor program. If compilation is required for some but not all of the programs he will provide the statements for the creation of a selective output file and for suppression of compilation where necessary. He will collect line printer output from editing and compilation, and will return it to the appropriate programmers. The functions of the work assembler in batch compilation are described on page 81.

Input

For a file creation run, the input consists of the *editing file* which is on punched cards (in 1900 code or ATLAS code or both) or paper tape or both and consists of various #XMUM directives, described below, together with the information to be placed in the file. For a file updating run, the input consists of the *input file* to be updated, and an editing file similar to that required for creating a file. The input file, which must always be on magnetic tape, contains source programs in the format used by #XMUM, which conforms to the ICL 1900 Series composite file structure. The input file should, therefore, have been itself created by #XMUM.

The editing file contains some directives referring to the file as a whole, and a series of groups of directives referring to particular jobs; each such group is called a *job reference.* The jobs on the input file will be in alphanumeric order of job name and the job references should normally also be in that order. If the job references are out of order, an extra file will be required as a work file. If a work file is not required, this must be indicated at the start of the editing file by the *ORDER directive (see page 52).

The following example shows the form of the input for the insertion of a new program on the file:

> *FORTRAN *name*
>
> *one or more source segments*
>
> *DATA
>
> *lines of data*
>
> ****

Summary of facilities

A run of #XMUM is normally used to produce an updated file suitable for input to subsequent runs of the program under the control of directives as specified in the section *Directives* on page 51. Alternatively by use of the *OFFLINE directive the run may be used for offlining only, in which case the output file is not suitable for subsequent input to the program. Unless the program is informed by the *ORDER directive that the jobs are referenced in the editing file in alphanumeric order, a work file is created, which may be either a scratch file or a named file designated by a *WORKFILE directive.

An individual job may be copied to the output file and to any selective output file (see page 51) without amendment by giving only start of job and end of job directives. If these are not given for a specific job the job will be copied to the full output file but not to the selective output file. Other operations which may be requested for individual jobs, and which require further directives are:

1. A whole job or the semicompiled segments and/or steering line and/or data may be deleted. The *DELETE directive (page 53) is used.
2. Lines of source program may be deleted and new lines may be inserted. The *ALTER directive (page 53) is used.
3. Semicompiled segments may be added or the existing semicompiled segments may be replaced by new ones. The *SEMI directive (page 54) is used.
4. The trace steering list may be line edited, or a completely new steering list may be written to a magnetic tape file, but as this list is not meaningful on disc it will be processed but not written to any disc output files. The *STEER and *ALTER directives (pages 54 and 53) are used.
5. Lines of data may be deleted and new lines may be inserted. The *DATA and *ALTER directives (pages 54 and 53) are used.
6. A copy of all or part of a job may be produced on cards or paper tape by means of the *CPCOPY and *TPCOPY directives (page 56).

7 If any reference is made to a job, that job will normally be compiled by a batch monitor or batch development system when the new batch is processed. However, it is possible to perform editing operations on a job, or to offline a new job, and to prevent that job from being compiled just as if no reference had been made to the job. The *NOCOMPILATION directive (page 56) is used.

8 The editing file may contain a mixture of cards and paper tape and whenever a change of input peripheral is necessary a *SWITCH directive (page 54) is used. A change of card input from 1900 code to ATLAS code or vice versa may be specified by the *ATLAS and *1900 directives (page 54) respectively.

9 Full listings may be obtained by a *LIST directive (page 54). Some listing is normally obtained without this directive but all listing may be suppressed by the *NOLIST directive.

10 An alternative job terminator may be specified by means of the *LAST directive (page 56).

Any or all of operations 1, 2, 3, 4 and 5 may be combined but must be performed in the order given, as the constituents of all jobs in the input file will be in the standard order of source segments, followed by semicompiled segments, if any, the steering list, if present, and data, if any. New jobs may be inserted in an existing batch and are given in the editing file.

The directives requesting operations 6, 7, 9 and 10 may be input in any order immediately following the start of job directive.

Note: In certain job directives, the initial word is followed by one or more parameters that specify the precise editing operation required. In such cases, the initial word must be followed by at least one space or horizontal tabulation character.

THE RESTART FACILITY

If a failure occurs during the latter stages of an #XMUM run, and provided a work file has been successfully created and retained it will not be necessary to repeat the entire run as, in most circumstances, the program may subsequently be restarted at a point about midway through the run. This also avoids having to re-input the editing file from cards or paper tape.

The successful creation of a work file is indicated by the line printer message:

AMENDMENT FILE HAS BEEN SUCCESSFULLY CREATED

and at the same time by the console typewriter message DISPLAY:AM.

In order to employ the restart facility the following conditions must apply during an unsuccessful run:

1 The directives *ORDER or *OFFLINE should not be present.

2 The program must proceed past the printing of the workfile creation messages mentioned above.

3 The run should not terminate as a result of a workfile failure.

4 The work file should not be a scratch disc file.

If the workfile is a magnetic tape scratch file, the operator will be able to restart the run (see page 92 of Chapter 15).

If the workfile is a named file, a separate restart run is required; all the files including the preserved work file are put on line except for the editing file which is replaced by a new file containing only the *RESTART directive (page 52), an end of batch directive and a blank record.

Output

The principal output file contains the complete batch of jobs in their amended form. An additional selective output file may also be created containing only those jobs which have been explicitly referenced during the run. Magnetic tape or disc may be used for either of the output files and both files conform to the ICL 1900 Series composite file structure. At least one of these output file must be specified for each run. If output is to a disc file the file could be used for subsequent input to the disc compiler #XFAE or the disc editor program #XMED described in the manual *FORTRAN: 16K Disc Compiler* (TP 4131).

Some line printer output (see below) is always produced. The other possible form of output is the production of a paper tape or card copy of all or part of a job.

LINE PRINTER OUTPUT

Some listing is always produced for each job referenced. This is discussed in the section *The* *LIST *and *NOLIST directives* on page 54.

If an error in the editing file is detected, an error message is output on the line printer. If the error is caused by a batch directive, a magnetic tape or tape deck failure, or if the input file has an incorrect format, the run will be abandoned; otherwise the run will continue but further processing of the job reference that caused the error may not be possible. The error messages are self-explanatory and are not described here but the actions taken by #XMUM on the detection of certain of the most common errors are given below:

1 If a job reference or an inserted job is out of order when an *ORDER directive has been used, the job will not be written on the output tape; any reference and associated input will be ignored.

2 If editing operations are attempted in an incorrect order (see page 49), the first attempt out of order and all subsequent attempts in that job reference will be ignored. If, for example, an attempt were made to edit the data and then the semicompiled segments, the source and semicompiled segments would be copied unchanged and the data editing would be performed but the reference to the semicompiled segments would be ignored.

3 If an attempt is made to insert a new job that has the same name as an existing job, the new job will be ignored, except when the directive *DELETE∇ALL appears in the new job, i.e. when the old job is being replaced.

4 If a checksum error or a card sequence error is found in a semicompiled segment, further semicompiled input in that job reference is ignored.

5 If a card sequence error is found in source program, no action is taken but SEQ ERR appears after the listing of the contents of the card concerned.

6 An attempt to edit data when the job concerned has no data causes any further reference to the job to be ignored.

When a run of #XMUM ends, or is terminated, a list of all the jobs on the tape output files is given. Jobs that have been referenced during the run and not set latent by the *NOCOMPILATION directive are indicated by asterisks. For a disc output file the subfile names are given in preference to the job names and the bucket length of each subfile is also provided.

An example of this output follows for a run producing a selective file and a full output file:

page n

	07/01/70	LISTING BY EDITOR XMUM MARK 3A
JOBS APPEARING IN THE OUTPUT FILE CSBSOURCE1(5)		
JOB	LANGUAGE	DATE LAST REFERENCED
*A3	FORT	07/01/70
EA6	FORT	29/12/69
FIOT	FORT	29/12/69
*TEMPTEST	FORT	07/01/70
*TEST1	FORT	07/01/70

page n + 1

	07/01/70	LISTING BY EDITOR XMUM MARK 3A
SUBFILES APPEARING IN THE SELECT FILE COMPSOURCE1(3)		
SUBFILE NAME	LENGTH IN BUCKETS	DATE LAST REFERENCED
*FORTA3	7	07/01/70
SEMIA3	3	07/01/70
*FORTTEMPTEST	11	07/01/70
DATATEMPTEST	6	07/01/70
*FORTTEST1	13	07/01/70
DIRECTORY SUBFILES	2	
TOTAL BUCKETS USED	42	
FILE LENGTH	80	
BUCKETS SPARE	38	

Directives
There are two groups of directives: *batch directives* which control the complete run of #XMUM, and *job directives* which control the processing of individual subfiles. Except for the *end of batch directive* which terminates the editing file, all the batch directives used for a particular run must be given at the start of the editing file, in any order.

Each directive must be punched at the start of a new card or a line or paper tape. Only the first four characters of any of the directives need be punched, and no line of program or data submitted may start with any of these groups of characters.

BATCH DIRECTIVES

*The *IN directive*
If the purpose of the run is to update a file, this file is used as the input file and must be opened by an *IN directive, which takes the following form:

*IN(*F*(*g*))

where *F* is a file name and *g* a file generation number.

This file must always be on magnetic tape.

If there is no input file, this directive is omitted.

*The *OUT directive*
This directive specifies the file which is to be opened as the principal output file, and must always be present unless the *SELECT directive is used. The directive takes the following form:

*OUT *T*(*F*(*g*)=*F'*(*g'*))

T indicates the device type, and takes one of the following forms:

MT for a magnetic tape file.

ED for an E.D.S. or T.E.D.S. file.

FD for an F.D.S. file.

If *T* is omitted, the device type is assumed to be MT. *F* and *F'* represent filenames consisting of from 1 to 12 alphanumeric, space or minus characters, the first of which must be a letter, and *g* and *g'* represent file generation numbers.

The file *F*(*g*) will be opened as the output file and will then be renamed *F'*(*g'*). The name and generation number of the existing file may be retained for the output file, in which case the directive takes the form:

*OUT *T*(*F*(*g*))

If only the generation number of the output file is to be altered, the following form of directive may be used:

*OUT *T*(*F*(*g*)=(*g'*))

The output file will then retain the name *F* but will have its generation number updated to *g'*.

In any of the above forms (*g*) may be omitted resulting, in the case of a disc file, in the highest generation number in the system being assumed, and, in the case of a magnetic tape file, in no check being made.

A magnetic tape file which has been renamed is also given a new retention period of 4095 days. However, this does not preclude its being specified in a *OUT or *SELECT directive in a subsequent #XMUM run.

*The *SELECT directive*
This directive specifies the file which is to be opened as the selective output file and to which only those jobs specifically referenced during the run are to be written. It may be used in addition to or as an alternative to the *OUT directive.

The directive may take any of the following forms:

*SELECT *T*(*F*(*g*)=*F'*(*g'*))

*SELECT *T*(*F*(*g*))

*SELECT *T*(*F*(*g*)=(*g'*))

Where each of these forms is similar to that specified for the *OUT directive.

*The *ORDER directive*
This directive indicates that the job directives are in alphanumeric order of job name so that a work file is not required. The order of precedence is space, 0 to 9, A to Z. The directive takes the following form:

*ORDER

If no *ORDER directive appears amongst the Batch directives, a work file will automatically be obtained, unless an *OFFLINE directive is used. The work file will be a scratch magnetic tape unless the *WORKFILE directive is used. A file will be created on the work tape and will be given the name XMUMTAPE generation 0 but will have a zero retention period.

*The *OFFLINE directive*
This directive is used when the run is for the purpose of offlining only and the jobs to be offlined are not presented in alphanumeric order of name and are not required in order on the output file. A work file will not be used and the jobs will appear on the output file in the order in which they are input from the editing file. The file produced, if on magnetic tape, cannot subsequently be edited by #XMUM, although if it is on disc it can be edited by #XMED. The directive takes the following form:

*OFFLINE

No *IN directive will appear with *OFFLINE, but an *OUT directive is obligatory.

If subsequent editing on magnetic tape is required, the jobs must be input in alphanumeric order or a work file must be used.

*The *WORKFILE directive*
Normally the work file, when used, will be a scratch magnetic tape but, if a named file is to be used as the work file, this directive must appear at the start of the editing file. The directive takes the following form:

*WORKFILE *T*(*F*(*g*))

where *T*, *F* and *g* are as defined for the *OUT directive.

F and *g* are the name and generation number of the file that is to be used as the work file. This file will not be renamed and will not be given a new retention period. A scratch work file on disc may be obtained by the use of the following directive:

*WORKFILE *T*

*The *LAST directive*
This directive permits the use of a four character job terminator directive other than the normal **** for all the jobs in the run. The directive takes the following form:

*LAST *abcd*

abcd represents the alternative terminator and must consist of any four non space printing-set characters, but must not begin with any of the four characters $,], ↑, or ←, and must not be the same as the first four characters of any #XMUM directive.

*The *RESTART directive*
If a job is to be restarted from a named work file the editing file for the restart run should contain a *RESTART directive followed by the end of batch directive and a blank record. The *RESTART directive takes the form:

*RESTART *T*(*F*(*g*))

where *T*, *f* and *g* are as defined for the *OUT directive.

The end of batch directive
The end of the editing file must be indicated by the following directive:

////

This should be followed by a blank card or a blank line on paper tape, i.e. two adjacent FE_2 (newline) characters.

JOB DIRECTIVES
After the initial batch directives further directives will apply to individual jobs. Each job is referenced only once and each reference starts with the start of job directive of the job concerned.

The start of job directive
For a FORTRAN job, this takes the form:

*FORTRAN *name,identification*

The job *name*, which will normally be the program name, may consist of up to eight alphanumeric or space characters and must start with a letter . *identification*, which is an optional entry, may take any form required; for example, it may take the form of the name and accounting code of the user. If there is no *identification*, the preceding comma may also be omitted.

*The *DELETE directive*
Whole jobs or parts of jobs may be deleted from an input file by the use of the *DELETE directive. The directive takes the form:

*DELETE∇*parameters*

The *parameters* entry is optional. If *parameters* is not present, or if it takes the form ALL, the complete job is deleted; that is it does not appear in the output file. Alternatively *parameters* may take the form of any or all of SEMI, STEER and DATA, separated by commas. These entries delete any semicompiled segments, steering list and data respectively that form part of the job on the input file. The source program may only be deleted by deleting the entire job.

*The *ALTER directive*
The *ALTER directive has three general forms, which are:

*ALTER *m,d*

*ALTER *m–n*

*ALTER *m*

m, d and *n* are integers. If the first form is used, the job will be copied from the input file to the output file until the *m*th line of the source program, steering list or data is reached. *d* lines will then be deleted, starting with the *m*th line. Any lines to be inserted at that point should follow the *ALTER directive in the editing file. If the second form is used, the *m*th to the *n*th lines of the source program, steering line or data in the job will be deleted. If any lines are to be inserted between the (m–1)th and (n+1)th lines, they should follow the directive.

If an insertion is to be made without a deletion the third form is used. The lines to be inserted should follow the directive; they will be inserted between the (m–1)th and the *m*th lines.

Any required number of *ALTER directives may be used with a job but they must be input in ascending order of line number. No reference should be made in a directive to a line deleted by another directive and lines deleted or inserted by a previous directive should not be taken into account when the values of *m* and *n* are determined. The line numbers given in a listing of all the jobs in their current form should be used. Listings giving line numbers are automatically given each time a new job is written to the output file and each time it is amended. Line numbers are not written to the output file; they are generated afresh each time. If a record on magnetic tape or disc contains more data than can be listed on one line of a line printer, a > character will appear between the line number and the line itself, and the remaining data is not listed.

Lines of the steering list and data are manipulated in the same way as lines of source program but the *ALTER directive, or group of such directives must be immediately preceded by the *STEER and *DATA directive respectively. If the job contains no data but data is to be added by the editing run, the *DATA directive alone is used and is followed by the data to be inserted. The same applies to a steering list. The *ALTER directive must not be used to alter lines of semi-compiled segments.

As an example, the necessary job reference in the editing file is given below to perform the following editing:

1 A job CAPHY is to have associated source program amended by replacing the 2nd line and 22nd line each with a DIMENSION and COMMON statement.

2 A PAUSE statement is to be inserted before line 30 and lines 50 to 58 are to be deleted.

3 A set of data is to be included in the job for the first time.

```
*FORTRAN CAPHY,JIM∇SMITH,PHYSICS∇LAB∇3B
*ALTER 2, 1
        DIMENSION∇A(5,5),B(10,10),C(15,15)
        COMMON/SHARE/A,B,C
```

```
*ALTER 22,1
        DIMENSIONVH(5,5),G(10,10),K(15,15)
        COMMON/SHARE/H,G,K
*ALTER 30
        PAUSE 55
*ALTER 50-58
*DATA
15  18  2  5  8 7
****
```

*The *SEMI directive*
When the *SEMI directive appears in a reference to a job all the semicompiled segments currently in that job will be deleted. Any semicompiled segments may be incorporated for the first time in the same way. The directive takes the following form:

*SEMI

At least one new segment must be presented. If the existing segments are still required and a new one is to be added, the existing segments must be re-input. This directive, if used, must follow any references to source segments and must precede any reference to the steering list or to data.

*The *STEER and *DATA directives*
These directives are used to introduce any alterations to the steering list or to data respectively. They take the forms:

*STEER

*DATA

If both are present for a job they must occur in the above order. Either must follow any references to source program or to semi-compiled segments.

*The *1900 and *ATLAS directives*
Cards may be input to #XMUM in either 1900 or ATLAS code. It may be required to change the code between source segments or between source and data. For this reason directives are available for switching 1900 and ATLAS card codes. These may appear anywhere in or before source, steering information or data. At the beginning of each job the card code specified by the entry point is assumed.

The directives have the following format:

*ATLAS

*1900

Cards input after the *ATLAS directive must be in ATLAS card code and after the *1900 directive in 1900 Series card code.

*The *SWITCH directive*
Parts of the editing file may be input on cards and other parts on paper tape. Any number of changes of input peripheral may be made and a change may occur at the end of any line or card, either within a reference to a job or between jobs. The same directive is used to switch from cards to paper tape as is used to change from paper tape to cards; it takes the following form:

*SWITCH

blank record

The blank record is a blank card if the change is from cards to paper tape and is a blank line, i.e. an extra FE_2 (newline) character, if the change is from paper tape to cards.

*The *LIST and *NOLIST directives*
Some listing is always produced when a job is referenced. The amount of listing normally depends on the amount of editing performed but can be controlled by the *LIST and *NOLIST directives.

The first part of the listing of each job gives the job name and indicates whether the job has been inserted, deleted, amended, replaced or merely referenced, i.e. copied unchanged from the input file to the output file; this listing is always given. Unless the *LIST or *NOLIST directives have been used:

1 If the job has been deleted, no more listing is produced for that job.

2 If the job has been copied unchanged, only the job directives present in the job and in the editing file are printed.

3 If a new job has been inserted or has replaced an old job, a complete listing is given, including all job directives. Source program is printed with line numbers (not statement numbers) at the left; deleted lines are indicated by a message of the form *n* LINES DELETED, where *n* is the number of lines. Semicompiled segments are listed by segment name only.

4 If the job has been amended, each amended section or newly inserted section, i.e. SOURCE segments, semi-compiled segment names, steering line or data, is listed in full. Job directives are printed.

The *LIST directive is used to increase the amount of listing when a job is not new and has not been amended in each section. The directive takes the following forms:

Directive	*Result*
*LISTVALL	A full listing is obtained.
*LIST	The source segments are listed.
*LISTVSEMI	The semicompiled segment names are listed.
*LISTVSTEER	The steering list is listed.
*LISTVDATA	The data is listed.

If two sections of the job are to be listed, only one statement is necessary; SEMI, STEER and DATA may appear in the same statement, separated by commas. If the source segments and another section are required, SOURCE must appear in the statement, e.g. *LISTVSOURCE,SEMI would obtain a listing of the source and semicompiled segments.

The *NOLIST directive is used to reduce the amount of listing obtained when a job is new or has been amended. The directive takes the following forms:

Directive	*Result*
*NOLISTVALL	The only listing obtained is of the job directives, as for a job that has been copied unchanged from the input file.
*NOLIST	No listing of the source segments will be produced.
*NOLISTVSEMI	Semicompiled segment names will not be listed.
*NOLISTVDATA	The data will not be listed.
*NOLISTVSTEER	The steering list will not be listed.

SEMI, STEER and DATA may be combined with each other and with SOURCE, as in the *LIST directive.

The *LIST and *NOLIST directives. when used, must appear in the first group of directives within the job reference, i.e. between the start of job directive and the *ALTER, *SEMI, *STEER and *DATA directives, if any.

The following is an example of the listing produced for the insertion of a newjob consisting of a source segment and data into a file when no *NOLIST directive has been used.

```
PAGE n

          time       date     LISTING BY EDITOR XMUM
          *FORTRAN SIDNEY, M.G.H.LAB7
          *TPCOPY   ALL
          JOB SIDNEY INSERTED INTO FILE JACK (6)
           *FORTRAN SIDNEY, M.G.H.LAB7
0001                MASTER JACK
```

```
0002            DIMENSION A(2, 10)
                .........
                .........
0059            END
        *DATA
0001    2   5   9
0002    10
0003    11
0004    14
0005    29
        ****
```

*The *NOCOMPILATION directive*

This directive is normally used in connection with the Batch Monitor and Batch Development systems to suppress the compilation of a job which has been amended or otherwise referenced during the #XMUM run. However the use of this directive will also have the effect of omitting the job from a selective output file. The directive should appear between the start of job directive and the *ALTER, *SEMI and *DATA directives, if used, or before the first line of source program of a new job. The directive takes the form:

*NOCOMPILATION

*The *CPCOPY and *TPCOPY directives*

A copy of the whole of a job or of one or more sections may be obtained on cards by the *CPCOPY directive or on paper tape by the *TPCOPY directive. The directive must precede any *ALTER, *SEMI, *STEER or *DATA directives or lines of source program.

A copy of the whole job is produced by one of the following directives:

*CPCOPY∇ALL

*TPCOPY∇ALL

The job will be punched in the order in which it is written on the output file, and will start with a start of job; directive and end with an end of job directive.

If only sections of the job are to be punched, the *CPCOPY or *TPCOPY should be followed by SOURCE for a copy of the source segments, DATA for the data, etc. If only the source segments are to be punched, SOURCE may be omitted but if other parts are also to be punched, SOURCE must be present and should be followed by a comma; if two or more of SEMI, STEER and DATA are present they should be separated by commas. Semicompiled segments, when punched, will be preceded by the *SEMI directive, the steering list by the *STEER directive and data by the *DATA directive but the end of job directive will not appear unless the whole job is punched.

*The *LAST directive*

This directive permits the use of a four character job terminator directive other than the normal ****. It is similar to the batch directive *LAST except that it applies only to the job whose start of job directive it immediately follows. The directive takes the following form:

*LAST *abcd*

abcd represents the alternative terminator and must consist of any four non space printing-set characters, but must not be the same as the first four characters of any #XMUM directive and must not begin with any of the four characters $,], ↑, or ←.

The end of job directive

This takes the form:

****

unless it has been altered by a batch or job *LAST directive. Details of permissible alternative end of job directives are given in the section *The *LAST directive* above.

THE EDITOR PROGRAM #XKYA

This section describes the use of the editor program #XKYA. The general description of the program is followed by details of input and output file formats and a summary of all the available editor instructions. The instructions are described in details, in alphabetical order, in the reference section.

General description

#XKYA edits magnetic tape files at several levels. At the highest level, #XKYA is used to rearrange, delete and add subfiles; at the lowest level it is used to amend individual lines of a program. Instructions to #XKYA are input from a basic peripheral and are addressed either to the *main editor* or to a subeditor called the *line editor.* The main editor deals with complete subfiles, header labels, trailer labels and sentinels. The line editor inserts, deletes, or changes individual lines within a subfile. Lines are reference by line number.

Basically, a run of the editor program produces an updated file from an old file by means of a file of instructions, known as the *editing file,* input from a basic peripheral. The new file should be created on a named file, however, a scratch tape may be specified, in which case any available expired tape will be opened. In this case it should be renamed and given a retention period. In addition to the principal input file, up to four additional input files may contribute complete subfiles, which are copied into specified positions in the output file; subfiles are copied selectively. Individual lines or complete programs that are to be inserted are included in the editing file.

File formats

Source programs can be held on tape only in standard 1900 Series subfile formats. Magnetic tape files for editing must, therefore, conform to the 1900 Series composite file structure.

Source program sentinel formats are given in the manual *Compiling Systems.* The maximum size of blocks output by #XKYA is controlled by a BLO instruction. For input to the FORTRAN compiler the maximum block size is 128 words (see *The BLOCKSIZE instruction* on page 57). Trailing spaces are removed from each record before it is written to the output file. However, since each record is an integral number of words, up to three spaces are added if necessary.

Summary of editor instructions

All instructions starts with a mnemonic that is recognized by #XKYA. The rest of the instruction contains relevant information in a format determined by the first mnemonic. Mnemonics are used with the main and line editors. All line editor instructions are preceded by a warning character, chosen by the programmer.

The facilities available with each of the editors are described below, and the mnemonic that is used to obtain each facility is given. A more detailed description is given in the reference section.

MAIN EDITOR

Opening the output file

The OFW instruction opens the output file; if this file is to be renamed, the RENAME instruction will write a new header label as required.

Creating new subfiles

The OSF instruction starts a new subfile by writing a start-of-subfile sentinel. When the required contents have been transferred from the editing file by the line editor, the new subfile is closed by a CSF instruction.

Opening input files

The OFR instruction opens a specified file for reading

Positioning input files

The POSITION instruction positions any specified input file at the start of the required subfile; if the required subfile is on the principle input file, and it has already been passed, the BACKSPACE instruction should be used. Any file may be positioned at its head by the REWIND instruction.

Transferring subfiles

One or more subfiles are copied from any input file to the output file by the COPY instruction. The OMIT statement is used to skip over the next subfile of the principal input file but can also be used to transcribe the whole file except for specified subfiles. Each subfile omitted requires a separate OMIT instruction. The RSF instruction can be used in conjunction with the COPY or LINEDIT instructions to rename a subfile copied from the principal input file.

Introducing the line editor

The LINEDIT instruction introduces a set of instructions to the line editor. The BLOCKSIZE instruction is used to specify the size of the blocks to be produced by the line editor.

Monitoring

As each instruction is executed, progress is indicated on a line printer. A printout of all subfile sentinels in the output file is normally obtained but the EPS instruction may be used to inhibit the printing of sentinels. The SPS instruction restarts this printing.

Closing the files

TRAILER writes a trailer label on the output file; CLOSE closes any specified file. More than one output file can be used in an editing run, but each must be closed before the next one can be opened.

Terminating the run

A FINISH instruction is used at the end of a run.

Miscellaneous instructions

The CPCOPY and TPCOPY instructions cause the next subfile written by the line editor to be output on a card punch or paper tape punch respectively. The DELETE instruction causes the editor program to delete itself.

LINE EDITOR

Writing

The ALTER instruction can be used to write lines of source program from the editing file, after, or in place of, specified lines of a subfile being copied from an input file. Alternatively the whole of the contents of a newly created subfile can be supplied from the editing file. If the lines being written are supplied on cards, the card sequence numbers will be copied unless a LOP instruction is used.

Monitoring

Normally every line written on the output file by the line editor will appear in the line printer progress report. If the SUPPRESS instruction is used, the number of lines printed will be restricted to the line immediately preceding any deleted lines and to lines that are inserted, together with the preceding line.

Returning to the main editor

The end of the application of the line editor is indicated by an END instruction. If a subfile is being transferred from an input file to the output file, the transfer is completed and the end-of-subfile is automatically updated before being written. If the current subfile is to be merged with a subfile yet to be found, the TERMINATE instruction can be used to return to the main editor in order to find the required subfile. The first subfile will not be closed and the second will be written by a further batch of line editor instructions; any number of subfiles may be merged in this way.

Reference section

In this section all the available instructions are described in detail. The main editor and the line editor are treated separately. Instructions are in alphabetical order.

The same input medium must be used for all the instructions and all the program in the editing file. Instructions to the main and line editors should not exceed 72 characters in length. On paper tape, every instruction must be terminated by a newline character; on cards, each instruction should start on a new card. The end of the editing file should be marked by a card or line that starts with four asterisks.

Only the first three characters of each instruction are essential. Thus CLO and CLOSE are synonymous.

When an editing file on paper tape is read, a space character will be inserted for each tape character encountered within any main editor instructions. The effect of tab characters within line editor instructions is defined in the section *Instructions to the line editor.*

INSTRUCTIONS TO THE MAIN EDITOR

BACKSPACE

This instruction positions the principal input file at the beginning of a specified subfile which has previously been passed in the file. The instruction has the following format, where *s* is the subfile name:

BACKSPACE *s*

BLOCKSIZE

This instruction is used to specify the block size that the line editor is to use when moving program from the editing file into a newly created subfile. For files which are to be read by the FORTRAN compiler, the following BLOCKSIZE directive must appear.

BLOCKSIZE , 128

(The comma is obligatory.) The BLOCKSIZE instruction does not affect subfiles being written by the main editor, for example by the COPY instruction.

CLOSE
This instruction closes the specified file. The file may then be re-opened with a new unit number, if necessary, and the old unit number of this file may be used again. The instruction takes the following form, where *n* is the unit number:

CLOSE MT*n*

If MT*n* is omitted, MT0 will be closed.

COPY
This instruction copies complete subfiles from an input file to the output file.

The instruction has the format:

COPY MT*n m s*

where *n* is the unit number, *m* is the mode and *s* is the subfile name. Each of the three parameters is optional.

If MT*n* is omitted, MT0, the principal input file, will be copied. *n* must not be 1 since MT1 is the output file

If the mode is not specified, or is zero, the input file is copied from where it is currently positioned, up to, but not including, the specified subfile, to the output file. If the mode is specified as 1, the action is the same except that the subfile named is also copied to the output file.

If a subfile name is not specified, mode should not be present. In this case the input file is copied to the output file, up to but not including the trailer label.

CPCOPY
This instruction causes the next subfile written by the line editor to be output on a card punch. The instruction has the format:

CPCOPY

CSF
This instruction writes an appropriate end-of-subfile sentinel to the output file. The instruction has the format:

CSF

A CSF instruction must be given to match every OSF directive. If CSF attempts to write an end-of-subfile sentinel for which there is no matching start-of-subfile sentinel on the output file, an error will be reported and #XKYA will delete itself.

DELETE
This instruction causes the editor program to delete itself and, normally, to output the message DELETED 99. If any minor errors, that is errors that have not caused premature deletion, have occurred, the message output is DELETED ME. The instruction has the format:

DELETE

EPS
This instruction inhibits the printing of sentinels in the listing. Printing of sentinels can be restarted by an SPS instruction. The instruction has the format:

EPS

FINISH
This instruction terminates the run and deletes #XKYA. If the file being created is completely new and not an edited form of another file, that is if there is no principal input file, an appropriate trailer label will automatically be written to the output file. Otherwise, the remainder of the principal input file will be copied on to the output file before the trailer label is written. All files are closed and a DELETED message is output on the console typewriter. The instruction has the format:

FINISH

LINEDIT
This instruction introduces a batch of line editor instructions and passes control to the line editor. The instruction has the format:

LINEDIT *s*

where *s*, subfile name is optional. If the subfile name is not present, the subfile before which the principal input file

is currently positioned will be edited by the line editor. If the subfile name is present, the principal input file will be copied to the output file until the specified subfile is reached. The subfile to be edited must be simple.

OFR

This instruction opens the specified file as an input file and gives it the unit number *n*. The instruction has the format:

OFR MT*n f(g)*

If MT*n* is omitted, the file is given unit number 0; *n* must not be 1.

The file generation number is optional, and may be omitted together with its associated parentheses, in which case no check is made on the file generation number.

All references to an input file must be to one that has been opened by OFR and not yet closed.

An attempt to open a file for reading as unit *n* when a unit *n* has already been opened will be ignored and an error will be signalled.

OFW

This instruction opens the specified file as the output file, giving it unit number 1. The instruction has the format:

OFW *f (g)*

The file generation number is optional.

If the file has an expired retention period, the file name should be specified as SCRATCH TAPE; in this case any available expired file or scratch file may be obtained.

The first reference to the output file in the editing file must be preceded by an OFW instruction. On receiving an OFW instruction, the program will throw a page and then print the date of the run followed by Words 1 to 8 of the file's header label; the tape serial number, the reel sequence number, the file generation number and the retention period are printed in octal.

More than one file may be written in a single run by issuing further OFW instructions. If a previous output file is still open when another OFW is issued (that is a CLOSE MT1 instruction has not been given) #XKYA will automatically close the existing output file and open a new one as specified.

OMIT

Each OMIT instruction causes one subfile to be omitted from the version of the principal input file copied to the output file. If no subfile is specified the input file is merely repositioned after the next subfile and nothing is written to the output file. If a subfile name *s* is specified, the input file is copied from its present postion to the end of the subfile before the subfile *s* and repositioned after subfile *s*. The instruction has the format:

OMIT *s*

OSF

This instruction starts a new subfile on the output file by writing an appropriate start-of-subfile sentinel. The instruction has the format:

OSF *m s, t*

where *m* is the mode, *s* is the subfile name, and *t* is the subfile type. For a simple subfile containing FORTRAN source, *m* must be zero or omitted, and *t* must be B2F4. For a composite subfile, *m* must be 1 and *t* must be C100. Each subfile created by an OSF instruction must be closed by a corresponding CSF instruction.

When a simple subfile is being created, the instruction after OSF should transfer control to the line editor so that the contents of the subfile can be transferred from the editing file.

An OSF instruction creating a composite subfile will be followed by instructions (such as COPY) that transfer complete subfiles to the output file, or by other OSF instructions.

POSITION

This instruction positions the specified input file at the start of the sentinel of the specified subfile. Nothing is, written to the output file. The instruction has the format:

POSITION MT*n s*

If MT*n* is omitted, MT0 will be positioned.

POSITION ignores changes of subfile level.

RENAME

This instruction renames the output file (MT1) by overwriting the header label. The instruction has the format:

RENAME *f(g//r)*

The new header label is printed on the line printer.

If the file generation number *g* is omitted the file will be given a file generation number of zero. If the retention period *r* is omitted or specified as zero, it will be taken as 4095 days. When present, the retention period must always be preceded by two solidi: if the retention period is omitted no solidi should appear. If both parameters are omitted, the solidi and parentheses should be omitted.

REWIND

This instruction positions the file on the specified unit number at its head, and has the format:

REWIND MT*n*

If MT*n* is not specified, MT0 will be positioned.

RSF

This instruction renames the specified subfile when it is transferred from the principal input file to the output file. If no subfile is specified, the next subfile to be copied is renamed; if a subfile is specified, all subfiles that occur between the current position of the input file and the specified subfile are copied to the output file before the subfile is renamed. The instruction has the format:

RSF *s'*, *s*

where *s* is the old subfile name and *s'* is the new subfile name.

SPS

All sentinels written to the output file are normally output as part of the listing, but the listing of sentinels may be suppressed by EPS instruction. The SPS instruction cancels the last EPS instruction and the printing of sentinels is continued. The instruction has the format:

SPS

TPCOPY

This instruction causes the next subfile written by the line editor to be output on a paper tape punch. The instruction has the format:

TPCOPY

TRAILER

This instruction writes an appropriate trailer label to the output file. Normally this will be done by a FINISH instruction but if another output file is to be used the TRAILER instruction should be used so as not to terminate the run. The instruction has the format:

TRAILER

INSTRUCTIONS TO THE LINE EDITOR

The LINEDIT main editor instruction causes control to be passed to the subeditor which edits by reference to line number. The line editor edits the subfile on which the principal input file (MT0) is currently positioned according to instructions in the editing file, producing an output subfile on the output file. The only input file that can be used by the line editor is the principal input file (MT0). The line editor can be used without an input subfile to produce a new subfile on the output file.

When first entered, the line editor writes the sentinel of the subfile about to be edited to the output file, prints it on the line printer(unless an EPS instruction is in force) throws to a new page and then prints the sentinel again. It then prints lines written to the output file according to the instructions. On leaving the line editor, another throw to a new page is given and the end-of-subfile sentinel is printed (unless suppressed) when it is written to the output file.

All line editor instructions must be preceded by a warning character. Any character may be used and is set, for any batch of instructions, by the character preceding the first instruction after the LIN instruction. The character used must not be the first character of any of the lines in the editing file that are to be inserted into the subfile. The batch of line editor instructions must be terminated by an END or TER instruction.

Any horizontal tabulation characters in a paper tape editing file are expanded into spaces according to the PLAN field format.

The line editor can accept 1900 or ATLAS card code. 1900 code is assumed on entry to the line editor.

In this chapter an asterisk is used as the line editor instruction warning character. Any suitable character in the 1900 Series 64 character set may be chosen by the user.

ALTER

This instruction writes lines from the editing file into the subfile currently being written. The line of the existing subfile after which the insertion is to be made is specified and that line and any number of the following lines of this subfile can be deleted. The instruction can also be used to insert information into a newly created subfile. It has the format:

*ALTER *n, d*

where *n* is the line number and *d* is the number of deletions.

This instruction copies lines from the input file to the output file up to, but not including, the line number specified. Starting from this point, it omits the number of input lines specified by *d*, and then reads lines from the editing file, writing them to the output tape until the next instruction, recognized by the warning character, is encountered.

If the line number is zero or is omitted, the new lines are inserted at the beginning of the subfile. If this parameter is omitted, the commas must remain.

If number of deletions is zero or is omitted, then new lines from the editing file will be inserted *after* the line number specified in the instruction and there will be no deletions. (Note that the corresponding directive to #XMUM would perform the insertions *before* the line number specified).

If a new subfile is being created both the line number and the number of deletions should be omitted.

Any number of ALTER instructions may be used with the same subfile but they must appear in ascending order of line number. If two successive ALTER instructions specify line numbers that are the same or of which the second is less that the first, the line editor will report an error and ignore further instructions until it reaches an END instruction.

ATLAS

This instruction informs the line editor that succeeding cards are in ATLAS code up to the end of the current batch of line editor instructions or up to a 1900 instruction. The instruction has the format:

*ATLAS

END

This instruction returns control to the main editor. The transfer of the current subfile is first completed and the end sentinel is automatically updated unless a new subfile has been created, in which case the CSF instruction must be used. If an END instruction immediately follows a LINEDIT instruction, no editing is carried out but a listing of the subfile is obtained. The instruction has the format:

*END

LOP

This instruction prevents the sequence numbers of the following cards in the editing file from being copied to the output file. It has no effect on paper tape. The instruction has the format:

*LOP

SUPPRESS

This instruction restricts the number of lines printed in the monitor printout. It has the format:

*SUPPRESS

Normally, every line written to the output file by the line editor will appear in the printout, but if the SUPPRESS instruction is used, only the following lines will be printed.

1 When deletions are made, the preceding lines.
2 When insertions are made from the editing file, the preceding line and the inserted lines.

When an insertion and a deletion are made by a single ALTER instruction the preceding line is printed only once.

The SUPPRESS instruction, when used, should be the first instruction after the LINEDIT instruction.

TERMINATE
This instruction returns control to the main editor but, unlike END, does not close the subfile. The end sentinel of the subfile being transferred is not copied and the next LINEDIT instruction will omit to copy the start sentinel of the subfile to be edited. In this way subfiles can be merged. The main editor positioning instructions such as BACK-SPACE and POSITION can be used after the TERMINATE instruction to position the input file as required before control is returned to the line editor to continue writing the subfile. Once part of a subfile has been written by the line editor, no main editor instruction may output to that subfile before the next LINEDIT instruction. The instruction has the format:

*TERMINATE

1900
This instruction informs the line editor that succeeding cards are in 1900 code up to the end of the current batch of line editor instructions or up to an ATLAS instruction. It has the format:

*1900

Chapter 11 Mixed language programming

A mixed language program may be created by consolidating together a number of semi-compiled segments. Nearly all 1900 compilers output segments in this form and thus FORTRAN segments may be mixed with PLAN segments, Algol segments etc. This chapter is concerned mainly with the inclusion of PLAN segments in a FORTRAN program.

COMMUNICATION OF DATA

The communication of data between PLAN and FORTRAN segments may be achieved either by passing over arguments or via a COMMON area. Table 5 on page 71 gives details of the number of words used to store FORTRAN variables.

TRANSFER OF CONTROL

The transfer of control to a segment in the other language is effected by the execution of a *calling sequence* within the calling segment. Control is then passed to the called segment, which will, normally, carry out the following:

1 Preliminary operations such as storing the link address to which control is to be returned on completion of the segment.
2 The main body of operations.
3 Terminal operation - returning to the calling segment via the link.

CALLING PLAN SEGMENTS FROM FORTRAN

A PLAN segment may be called from a FORTRAN segment either as a SUBROUTINE segment or a FUNCTION segment, the difference being that a SUBROUTINE segment may have any number of arguments, including zero, and a FUNCTION segment must have at least one argument. Furthermore a FUNCTION segment must return a result in a location that depends on the type of function. Table 4 lists the location of results for different types of functions.

Type of function	*Location of result*
Integer or Logical	X6
Real	Floating-point accumulator
Double-precision	Floating-point accumulator: first two words of result. X4, X5: Second two words of result.
Complex	Floating-point accumulator: real part of result. X4, X5: imaginary part of result.

Table 4: Location of results for different functions

Both a SUBROUTINE segment and a FUNCTION segment may return results via a common area or via the arguments.

When writing a FUNCTION or SUBROUTINE segments in PLAN, programmers are advised to use the subroutines FPROLOG and FEPILOG, which are included in the FORTRAN library. The function of FPROLOG is to pick up and store the line and the address of each argument in an area of store specified by the programmer; the function of FEPILOG is to execute the return of control to the calling segment.

Using these subroutines, a PLAN segment will consist of the following:

CALL to FPROLOG
Main body of segment
CALL to FEPILOG

and should start with a #PROGRAM directive of the form:

#PROGRAM /*name*

where *name* is the name of the segment, as in the CALL statement or function reference.

The FPROLOG subroutine

FUNCTION
To provide communication between a FORTRAN calling segment and a PLAN segment.

DESCRIPTION
This subroutine should be called by PLAN segments that are to be incorporated into a FORTRAN object program as functions or subroutines. The subroutine transfers a fixed or variable number of arguments from a FORTRAN calling sequence to a storage area specified in the PLAN segment, and passes the name of the called segment to the FORTRAN trace system.

The first word of the storage area specified in the PLAN segment is used to store link information for the return to the FORTRAN segment. This word must not be altered in any way during the segment. The return must be effected by a call to the FEPILOG subroutine (see the section *The FEPILOG subroutine).*

No instructions should precede the call to FPROLOG in the segment.

INPUT ARGUMENTS
The PLAN segment should commence with the following calling sequence:

```
CALL  3 FPROLOG
        8Hname
        c
        /addr
```

name is the name of the PLAN segment as it is to appear in any FORTRAN trace output. This name is held in two words, which should be space-filled if necessary, e.g. 8HPSEGVVVV

For PLAN segments requiring a *fixed* number of arguments, $c = n$, the number of arguments to be transferred. For PLAN segments requiring a *variable* number of arguments, $c = -m$, where m is the maximum number of arguments to be transferred, excluding the first. The first argument must specify n, the number of actual arguments for the particular call to the segment. If n is found to exceed m then a fatal execution error is signalled.

addr is the start address of the area, in lower or upper data, that is to receive the link information and arguments. This area must contain $n + 1$ words if n is fixed or $m + 2$ words if n is variable.

When the PLAN segment is to deal with a variable number of arguments, the first argument in the FORTRAN argument list must be an additional argument specifying the number of subsequent arguments; this first argument must be of type INTEGER and it may be a constant, a variable or an array element.

RESULTS
The result of a call to FPROLOG is as follows:

For a fixed number of arguments:

addr contains link information for use by FEPILOG

addr + 1 contains the address of the first argument.

addr + 2 contains the address of the second argument.

. . . .

addr + n = contains the address of the nth argument.

For a variable number of arguments

addr contains link information for use by FEPILOG

addr + 1 contains *n* the value of the first argument (giving the number of arguments to follow).

addr + 2 contains the address of the second argument.

. . . .

addr + *n* + 1 contains the address of the last argument.

USE OF OVERFLOW
If overflow is found to be set when FPROLOG is called then FORTRAN execution error 50 will be reported.

SPECIAL POINTS
1 No instructions should be obeyed before the call to FPROLOG.
2 The address of the argument as given by FPROLOG is contained in the modifier part of the word; the remaining bits may or may not be zero.
3 The word containing the link address should not be altered in any way.

The FEPILOG subroutine

FUNCTION
To return control from a PLAN SUBROUTINE or FUNCTION segment to the FORTRAN calling segment.

DESCRIPTION
This subroutines utilises link information generated by the subroutine FPROLOG, to effect a return from a PLAN segment to the FORTRAN calling segment; in addition the subroutine passes information to the FORTRAN trace system. The return of control is made directly from FEPILOG to the FORTRAN segment so that no EXIT instruction is required in the PLAN segment.

For every call to FPROLOG, within a program, there must be, dynamically, a corresponding call to FEPILOG.

INPUT ARGUMENTS
The calling sequence is as follows:

```
CALL 3 FEPILOG

        /addr
```

addr is the address of the location containing the link information stored by the corresponding call to FPROLOG.

RESULTS
Control will be returned to the calling segment and this will be noted by the FORTRAN trace system.

USE OF OVERFLOW
If overflow has been set during the PLAN segment and is still set on the calling of FEPILOG then FORTRAN execution error 50 is signalled.

Example

A FUNCTION segment is to be written for the purpose of adding together two complex numbers. The segment could be written in PLAN as follows:

I.C.T. 1900 SERIES PLAN CODING SHEET			TITLE / PROGRAMMER	SEGMENT / DATE
LABEL 1–6	OPERATION 7–12	ACC. 13–15	OPERAND 16–72	PROG. IDENT. 73–75 / SEQUENCE 76–80
#PROGRAM			/CADD	
#LOWER				
			LINK(3)	
#PROGRAM				
	CALL	3	FPROLOG	
			8HCADDΔΔΔΔ	
			2	
			/LINK	
	LDX	1	LINK+1 [ADDRESS OF 1ST ARGUMENT	
	LDX	2	LINK+2 [ADDRESS OF 2ND ARGUMENT	
	LFP		2(1)	
	FAD		2(2)	
	SFP		4 [X6, X5 HAVE IMAGINARY SUM	
	LFP		0(1)	
	FAD		0(2) [F.P.A. HAS REAL SUM	
	CALL	3	FEPILOG [RETURN	
			/LINK	
#END				

SHEET OF

This segment would be equivalent to the FORTRAN segment:

```
COMPLEX FUNCTION CADD (A,B)
COMPLEX A, B
CADD = A + B
RETURN
END
```

ARRAYS

In a FORTRAN program arrays are stored with their elements in consecutive storage units, which may consist of one, two or four words of storage, so that the left most subscript varies most rapidly and successive subscripts vary correspondingly less rapidly. In order that complete arrays may be 'handled' easily, information about each array is stored separately in an *array header.*

Two PLAN subroutines, GETAH and GENAH, are available for dealing with the array headers of FORTRAN arrays that are passed over into PLAN segments.

THE GETAH SUBROUTINE

Function

The subroutines unpacks a FORTRAN array header into an information block accessible to a PLAN segment.

Input arguments

The calling sequence is as follows:

```
CALL  1 GETAH
LDX   3 a
LDX   3 b
```

a contains the address of the array header.

b contains the address of the first word of the area of store that is to receive the information block. For an *n*-dimensional array the length of the information block must be *n + 4* words.

Results
The results returned to the information block are as follows:

WORD 0	contains the number of dimensions of the array (n).
WORD 1	contains the number of words per array element (m).
WORD 2	contains the Base address - the address of element (0, 0, 0, ..0).
WORD 3	contains the address of the first element of the array - element (1, 1, 1, ...,1).
WORD 4	contains the address of the first word past the end of the array.
WORD 5	contains the first partial product P_1 (only present for $n>2$).
WORD 6	contains the second partial product P_2 (only present for $n>3$).
WORD (n + 3)	contains the last partial product P_{n-1}.

Notes
P_k is defined by $P_k = D_1.D_2.D_3 . . . . .D_k$
where D_k is the number of elements in demension k, and k may take any of the values 1, 2, 3,...,n-1. The base address is found by subtracting m.N from the address of the first element, where $N = 1 + P_1 + P_2 . . . . +P_{n-1}$.

THE GENAH SUBROUTINE

Function
The subroutine generates a standard FORTRAN array header from an information block produced by GETAH.

Input arguments
The calling sequence is as follows:

```
CALL  1 GENAH
LDN   3 a
LDN   3 b
```

a contains the address of the first word of the area at which the array header is to be generated and *b* contains the address of the first word of the information block. If *a* and *b* are specified as the same the effect will be to condense the information block into a standard array header.

It is recommended that the same number of words are set aside for both the information block and the array header.

Results
A standard FORTRAN array header is generated at the address specified.

Example
It is desired to write a PLAN subroutine, to be called by a FORTRAN segment, to copy one array into another.

The arrays are assumed to have a similar structure and not more than three dimensions. Two argument, being two array names are required by the subroutine and the corresponding information blocks are called IB1 and IB2.

A possible version of the segment is as follows:

I.C.T. 1900 SERIES — PLAN CODING SHEET

LABEL	OPERATION	ACC.	OPERAND	PROG. IDENT.	SEQUENCE
#PROGRAM			/COPYARRY		
#LOWER					
			LINK(3),IB1(7),IB2(7)		
#PROGRAM					
	CALL	3	FPROLOG		
			8HCOPYARRY		
			2		
			/LINK		
	CALL	1	GETAH		
	LDX	3	LINK+1 [ADDRESS OF 1ST ARRAY HEADER		
	LDN	3	IB1		
	CALL	1	GETAH		
	LDX	3	LINK+2 [ADDRESS OF 2ND ARRAY HEADER		
	LDN	3	IB2		
	LDX	3	IB1+4		
	SBX	3	IB1+3 [SET X3 = NUMBER OF WORDS IN 1ST ARRY		
	LDX	7	IB2+4		
	SBX	7	IB2+3		

SHEET 1 OF 2

I.C.T. 1900 SERIES — PLAN CODING SHEET

LABEL	OPERATION	ACC.	OPERAND	PROG. IDENT.	SEQUENCE
	BXL	7	3,ER1 [ERROR IF 2ND ARRAY TOO SMALL		
	LDX	1	IB1+3		
	LDX	2	IB2+3		
L2	BXGE	3	'513',L1 [ALLOW FOR MORE THAN 512 WORDS		
	MOVE	1	0(3) [TRANSFER ELEMENTS		
	CALL	3	FEPILOG [RETURN		
			/LINK		
L1	MOVE	1	512		
	SBN	3	512		
	ADN	1	512		
	ADN	2	512		
	BRN		L2		
ER1	SUSWT		2HE1		
	BRN		*-1		
#END					

SHEET 2 OF 2

Common storage areas

In the previous examples information has been passed over to the called segment via arguments. An alternative, or complementary, method is to pass information via a common storage area.

4159(7.69)

When establishing common areas care must be taken to ensure that the differences in the sizes of the storage units used by FORTRAN and PLAN are allowed for. Details of the number of words used to store FORTRAN variables are given in the Table 5 on page 71. When an array is specified in a FORTRAN COMMON statement, the number of words occupied by the array is the product of the number of elements and number of words per element.

It must be remembered that when arrays are being dimensioned, the number enclosed in parentheses in FORTRAN refers to the number of elements, whereas in PLAN the number enclosed in parentheses refers to the number of words.

Thus the following FORTRAN statements:

```
REAL B
COMMON/AREA/B(7)
```

and the PLAN directive:

I.C.T. 1900 SERIES — PLAN CODING SHEET

TITLE	SEGMENT
PROGRAMMER	DATE

LABEL (1–6)	OPERATION (7–12)	ACC. (13–15)	OPERAND (16–72)	PROG. IDENT. (73–75)	SEQUENCE (76–80)
#UPPER			COMMON/AREA/		
			C(14)		

establish a common area named AREA in which B corresponds to C and both occupy 14 words. The PLAN directive # LOWER could be used in place of #UPPER to force the area to be placed in lower storage, although the FORTRAN segment(s) could not take advantage of this.

EXAMPLE OF THE USE OF COMMON

A FORTRAN segment contains the following statement:

```
LOGICAL BOOLEAN
COMMON/POOL/J, A(5), LIST (5,2), BOOLEAN
```

These establish a common area of 34 words called POOL, as follows:

INTEGER	variable	J - 2 words
REAL	array	A - 10 words
INTEGER	array	LIST - 20 words
LOGICAL	variable	BOOLEAN - 2 words

A PLAN segment called by the FORTRAN segment contains the directive:

I.C.T. 1900 SERIES — PLAN CODING SHEET

TITLE	SEGMENT
PROGRAMMER	DATE

LABEL (1–6)	OPERATION (7–12)	ACC. (13–15)	OPERAND (16–72)	PROG. IDENT. (73–75)	SEQUENCE (76–80)
#UPPER			COMMON/POOL/		
			ITEM,NULL,RES(10),K1(10),K2(10),Y(3)		

This establishes a common area of 35 words called POOL as follows:

ITEM	-	1 word
NULL	-	1 word
RES	-	10 words
K1	-	10 words
K2	-	3 words
Y	-	3 words

During consolidation the two areas will be combined into one storage area named POOL consisting of 35 words (the larger of the sizes) of which the FORTRAN segment may use only the first 34 words as follows:

J will correspond to ITEM and NULL

A will correspond to RES

LIST will correspond to K1 and K2

BOOLEAN will correspond to the first two words of Y

BLOCK DATA SEGMENTS

The equivalent of a FORTRAN BLOCK DATA segment may be written in PLAN. The initial # PROGRAM directive must be of the form:

PROGRAM *progname/segname*

where *progname* is the program name and *segname* is any unique segment name.

The segment must contain only # UPPER or # LOWER COMMON directives that set up common areas containing preset values.

BLANK COMMON AREAS

The blank common area may be used by both PLAN and FORTRAN segments, for example the blank common block established by the statement:

COMMON FRED, BERT

could be accessed by a PLAN segment containing the directive:

I.C.T. 1900 SERIES PLAN CODING SHEET			TITLE / PROGRAMMER		SEGMENT / DATE
LABEL 1–6	OPERATION 7–12	ACC. 13–15	OPERAND 16–72	PROG. IDENT. 73–75	SEQUENCE 76–80
#UPPER			COMMON//		
			JIM(4)		

Type	*Number of words*	*Format*
NORMAL COMPILATION MODE		
REAL	2	Word 1 Bits 0 to 23 signed mantissa Word 2 Bit 0 0 Bits 1 to 14 continuation of mantissa Bits 15 to 23 exponent + 256 (i.e. as PLAN floating point number)
INTEGER	2	Word 1 Bits 0 to 23 signed binary integer Word 2 Bits 0 to 23 unused
LOGICAL	2	Word 1 Bits 0 to 22 0 Bit 23 0 for FALSE 1 for TRUE Word 2 Bits 0 to 23 unused
DOUBLE PRECISION	4	Word 1, Word 2 } as for REAL Word 3 Bit 0 0 Bits 1 to 23 continuation of mantissa Word 4 Bit 0 0 Bits 1 to 9 final part of mantissa
COMPLEX	4	Word 1, Word 2 } as for REAL Word 3, Word 4 } as for REAL The first pair give the real part and the second pair the imaginary part of the complex number.
COMPRESS INTEGER AND LOGICAL MODE		
INTEGER	1	Word 1 Bits 0 to 23 signed binary integer
LOGICAL	1	Word 1 Bits 0 to 22 0 Bit 23 0 for FALSE 1 for TRUE
COMPRESS DOUBLE PRECISION MODE		
DOUBLE PRECISION	2	Word 1, Word 2 } as for REAL

Table 5 The method of storage of FORTRAN variables.

CALLING FORTRAN SEGMENTS FROM PLAN

The following calling sequence should be used wherever a FORTRAN segment is called from a PLAN segment.

```
CALL  1 segname
LDN   3 a1
LDN   3 a2
..........
..........
LDN   3 an
```

$a_1, a_2, ...a_n$ are locations in the LOWER data store containing each argument in turn.

If any of the arguments are held in the UPPER data store then the

LDN 3 *a*

instruction should be replaced by an instruction of the form

LDX 3 *b*

where *b* is the word in the # LOWER data store containing the address of the argument.

Both forms are such that when obeyed, the address of the argument is placed in X3; Control is returned to the word following the last LDN or LDX instruction.

Table 6 gives the values of the operand and associated instructions for the various types of arguments.

Type of argument required by FORTRAN segment	*Instruction*	*Operand*
Constant or variable	LDN 3	Address of constant or variable
	LDX 3	Address of word containing address of constant or variable
	LDN 3	Address of array header
	LDX 3	Address of word containing address of array header
Array	LDX 3	Address of word containing 128/ address of first element
Function or Subroutine	LDX 3	Address of word containing a branch instruction (BRN) to function or subroutine

***Table 6:** Value of operand in PLAN instructions when calling a FORTRAN segment from a PLAN segment.*

Example

The FORTRAN subroutine PIVOT has been written to determine the element of a one dimensional array that contains the lowest value and to return the element subscript POOL and the lowest value MIN as follows:

```
SUBROUTINE PIVOT (POOL, A, N, MIN)
REAL MIN
DIMENSION A(N)
```

```
      POOL  = 1
      MIN   = A(1)
      DO 100 I = 2, N
      IF (MIN.GT.A(1)) GO TO 100
      MIN   = A(1)
      POOL  = 1
  100 CONTINUE
      RETURN
      END
```

It is required to call this subroutine from a PLAN segment in order to process the array SHADOW; the calling sequence might be written as follows:

I.C.T. 1900 SERIES PLAN CODING SHEET		TITLE / PROGRAMMER	SEGMENT / DATE
LABEL (1–6)	**OPERATION** (7–12)	**ACC.** (13–15)	**OPERAND** (16–72)
#UPPER			SHADOW(50)
#PROGRAM			
	LDX	1	'50'
	STO	1	LENGTH
	CALL	1	PIVOT
	LDN	3	PCOU
	LDX	3	'128/SHADOW'
	LDN	3	LENGTH
	LDN	3	MIN

MIXED PLAN/FORTRAN OVERLAID PROGRAMS

PLAN segments included in a FORTRAN program can be overlaid in the same way as FORTRAN segments, provided they are mentioned in the OVERLAY statement in the program description and are entered via the subroutine FOVER (see below). A PLAN steering segment specifying overlay organization is not required and must not be input to the compiler. If the PLAN segment itself calls other segments it is advisable that this is done with FOVER.

The subroutine FOVER

FUNCTION

To call FORTRAN overlay system in PLAN segment in a mixed language overlaid program.

DESCRIPTION

In a FORTRAN/PLAN mixed language overlaid program that uses the FORTRAN overlay system, calls from a PLAN segment must normally be achieved by calling FOVER. (For the exceptional case in which this is only optional, see note 1 below).

The function of this subroutine is to ensure that the called segment is in core store (i.e. bringing in the overlay unit containing the called segment, if necessary), and then to exit to the segment (or cue, see note 2 below) specified. The normal RETURN statement (or call to FEPILOG) in the called segment causes the overlay system to transfer control to the instruction following the calling sequence for FOVER, after ensuring that it is in core store (i.e. bringing in the overlay unit that contains the instruction if it is an overlay area that has been overwritten since FOVER was called).

If the logic of the program causes control to pass more than once to the section of the program that contains a call to FOVER, on the second and subsequent occasions control will pass more quickly to the called segment.

INPUT ARGUMENTS
The calling sequence is as follows:

```
CALL   1  FOVER
ENTER     calledseg
LDN    3  a1
LDN    3  a2
......................
......................
LDN    3  an
BRING     callingseg .
```

where *calledseg* is the name of the called segment
$a_1, a_2, ...a_n$ are the *n* arguments of the called segment
callingseg is the name of the calling segment

This sequence occupies (n + 6) words, since ENTER and BRING are PLAN Macro instructions occupying 3 and 2 words respectively. (They are never executed as such but all the information they contain is used by FOVER.)

USE OF OVERFLOW
Unchanged

RETURN LOCATIONS
On exit from the called segment return (either direct or via the overlay system) is to CALL + (n +5) (where the CALL referred to is that to FOVER).

Notes:

1 If the calling and the called segments are to be consolidated into the same overlay unit, or both into permanent store, then the use of FOVER is only optional. The advantage of using it in this case is that subsequent reorganization of the overlay structure can be achieved without having to recompile any of the PLAN segments involved.

2 References to CALLEDSEG could equally be to CALLEDSEG + *c*, CUENAME, CUENAME + *c* (where CUENAME is the name of a cue) when the called segment is written in PLAN.

Restrictions
When using the FORTRAN overlay system the following three restrictions should be noted:

1 Not more than 20 arguments may be passed over to any called segment.

2 If the called segment and the calling segment are in different units of the same overlay area, then preset area, which is itself overlaid, should not be used for storing argument addresses or the arguments themselves.

3 If an EXIT instruction is used, the segment referred to in the link must currently be in store.

4 The PLAN instructions BRING, RECALL and ENTER should not be used except as specified with the FOVER subroutine.

Example

A PLAN segment POLAY with no arguments is to be included in area 1 unit 2 of a FORTRAN program. POLAY calls the FORTRAN segment FSEQ. The program description might include the statements

```
OVERLAY (1, 2) POLAY, OTHERSEGS
OVERLAY (1, 3) FSEQ, MORESEGS
```

and the PLAN segment would have the structure

I.C.T. 1900 SERIES PLAN CODING SHEET		TITLE / PROGRAMMER		SEGMENT / DATE
LABEL (1–6)	**OPERATION** (7–12)	**ACC.** (13–15)	**OPERAND** (16–72)	**PROG. IDENT** (73–75) / **SEQUENCE** (76–80)
#PROGRAM			/POLAY	
	CALL	3	FPROLOG	
			8HPOLAY▽▽▽	
			0	
			/LINK	
	CALL	1	FOVER	
	ENTER		FSEQ	
	BRING		POLAY	
	CALL	3	FEPILOG	
			/LINK	
#END				

ENTRY POINTS AND MASTER SEGMENTS

There must be one and only one master segment in a program. If written in FORTRAN is must start with a MASTER statement. If written in PLAN it must contain entry point 0. In general, use of entry point 9 should be avoided in a PLAN segment. If any of the FORTRAN segments have been compiled in TRACE 2 mode then no entry points should appear in a PLAN segment.

USE OF WORD 30

In a FORTRAN program the following use of bits in Word 30 is made:

Bit 0	Used to control statement tracing of any segment compiled in TRACE 2 mode.
Bits 1 to 8	Used by programmer as sense switches.
Bits 10 - 23	Reserved for future use by FORTRAN system.

In a mixed PLAN/FORTRAN program care should be taken to ensure that a bit in Word 30 is not used for more than one purpose.

PERIPHERAL USAGE

It is strongly recommended that input or output on a particular peripheral be programmed in one language only. If this is not done, unpredictable events may occur.

If a PLAN program incorporates FORTRAN segments containing FORTRAN input/output statements, the peripherals must be defined in a FORTRAN program description in the usual way and the semi-compiled segment produced from the program description (given the name %%%F) must be included in the PLAN program.

The FORTRAN input/output system will not be incorporated into the object program unless either READ or WRITE statements are present or at least one FORTRAN segment was compiled at trace level 2.

Peripherals nominated in a FORTRAN program description are assigned dynamically the first time they are used by the FORTRAN system.

SUBPROGRAMMING

There are no subprogramming facilities in FORTRAN but subprogramming can be used in PLAN in the normal way.

COMPILING MIXED FORTRAN/PLAN PROGRAMS

This section gives some general guidance on how semi-compiled PLAN segments can be produced and how they can be incorporated into a FORTRAN program. The information given here is not comprehensive and further information will be found in the *PLAN Reference Manual.*

For a 16K configuration, it is recommended that one of the PLAN 3 compilers #XPLG (for paper tape or card input) or #XPLV (for magnetic tape input) be used to compile the PLAN segments. The output will be an incomplete consolidated semi-compiled program containing the semi-compiled PLAN segments and any library subroutines from the PLAN library S-RS that are called by these segments.

All the segments are input to the FORTRAN compiler by a statement of the form

READ FROM (MT,*filename*)

where *filename* is the name of the PLAN compiler output file (as nominated by the programmer).

For a configuration of 32K or more, it is recommended that the PLAN 4 compiler #XPLN be used either to produce a consolidated semi-compiled program as before or to produce unconsolidated semi-compiled segments in named subfiles. In the latter case, the subfiles must be input to the FORTRAN compiler by nominating the subfiles in READ FROM statements (see page 22). Any library subroutines required from the PLAN library must then be obtained by using a LIBRARY statement and a READ FROM as described in Chapter 6.

THE LEADER STATEMENT

The presence of the LEADER statement in the program description of a mixed FORTRAN/PLAN program will cause information to be output on the listing peripheral at the end of the source listing. The information output gives a consolidated program summary; this consists of details of the actual areas of core store allocated to the program storage areas at run time. These details are output in a highly compressed and economical form (see Chapter 8 of the *PLAN Reference Manual*).

Chapter 12 GEORGE 1 and 2

Details of the use of the FORTRAN magnetic tape compiler under the GEORGE 1 and 2 operating system will be supplied as an amendment to this manual.

PART 2 OPERATING CONSIDERATIONS

Chapter 13 System description

This Chapter is designed for those responsible for the running of an installation and gives a general outline of the new system together with recommendations for optimizing magnetic tape usage.

THE 1900 SERIES FORTRAN MAGNETIC TAPE SYSTEM

The basis of the system is the FORTRAN magnetic tape compiler, #XFAM, which is held on a *system tape* together with its associated library group, SRF7, and a search program. The compiler accepts FORTRAN source segments or semi-compiled segments, either from one or more basic peripherals or from magnetic tape, and outputs semi-compiled segments to magnetic tape.

Compilation and consolidation

The compiler has three alternative *modes* of compilation: *segments* mode, *single program* mode and *batch* mode. The user specifies the mode that is required by an *initial compiling system statement* input at the beginning of the run.

SEGMENTS MODE

In segments mode the compiler output unconsolidated semi-compiled segments to a magnetic tape; consolidation is impossible.

SINGLE PROGRAM MODE

In single program mode the input to the compiler consists of a single complete program comprising a number of segments in FORTRAN source or semi-compiled form. Normally the compiler will output the consolidated semi-compiled program to a magnetic tape from which the program is automatically loaded into store, ready to be activated, the compiler deleting itself as the program is loaded. The user may prevent the loading of the program optionally, retaining the semi-compiled form, and he may obtain a binary dump of the loaded program. If the program is an overlay program the loading procedure involves making a binary dump of the program to a second output tape; loading then takes place from this tape.

BATCH MODE

In batch mode, the input to the compiler consists of two or more complete programs, which are compiled and consolidated and output in semi-compiled form to a magnetic tape. In addition, the search program held on the system tape is copied across to the beginning of the output tape so that the programs may be accessed.

Object programs

Once loaded into store, an object program is activated in the normal manner by a GO *#name* console message, where *# name* is the program name.

A program that is not automatically loaded is held on tape either as a single binary program, a single consolidated semi-compiled program or one of a batch of consolidated semi-compiled programs. In the first two cases the program is loaded by a FI*#name* console message, in the third case the program is loaded by a FI*#name#file* console message; where *#name* is the program name and PROGRAM *file* is the name of the tape containing the batch.

The peripherals used by an object program are specified in the *program description.* The 1900 Series FORTRAN System allows the use of magnetic tape and disc files (both E.D.S. and F.D.S.) in addition to all basic peripherals. More than one peripheral of a particular type may be used if required.

TAPE USAGE AND EFFICIENCY

Compile time

The FORTRAN magnetic tape compiler is oriented towards a named tape system. The tapes to be used during a compilation are nominated by the user in the initial compiling system statements; the absence of a nominated tape will be signalled at the console. The user need not nominate tapes for output; in the absence of a nominated output

tape the compiler will look for a scratch tape. The compiler will always require one output tape and an extra output tape will be required for overlay programs and where a binary dump is required. The compiler is itself overlaid and will retain one tape deck throughout a compilation. In order to minimize tape movement and search time the system tape should be organized as follows:

beginning of the system tape

#file - search program

#XFAM - magnetic tape compiler

SRF7 - compiler library

frequently used libraries

. . . .

The system tape is created using standard library manipulation programs. The file is called PROGRAM *file*.

It is considerably faster to load programs that are held in binary form rather than in consolidated semi-compiled form. Thus frequently used FORTRAN programs should be compiled and retained as binary dumps. The batch compilation facility may be used to maximize job throughput.

Object time

1900 Series FORTRAN users are enabled to specify the block size for formatted data transferred at object time. In general, the larger the block size, the faster will be transfers to and from magnetic tape at object time although this is achieved at the expense of an increase in program size.

OPTIONAL SYSTEM ALTERATIONS

Log analysis messages

When a batch of jobs is compiled in batch mode a DISPLAY message appears on the console log giving the program name and the mill time used. If #XFAM is used as supplied, this message does not appear when single programs are compiled. If the installation manager wishes the DISPLAY message to be produced for compilation of single programs as well as for batches he may achieve this by using the program #XPMW (see the manual *Compiling Systems*) to alter Word 30 of #XFAM to the value #10000000.

Default listing mode

If #XFAM is used as supplied, and no listing statement (see page 15) is given in the initial statements, the compiler assumes the SHORT LIST option. This default action can be changed so that the LIST option is assumed by using the program #XPMW (see the manual *Compiling Systems*) to alter Word 30 of #XFAM to the value #10000, or, if the log analysis message is also required, to #10010000.

Chapter 15 Operator's guide to the FORTRAN magnetic tape compiler

This chapter gives the operating instructions for the FORTRAN magnetic tape compiler. Though written as a reference document for operators, it is also intended as a programmer's guide for writing the operator's instructions.

The operating instructions are divided into two parts: the first lists the normal procedures and the second lists exception conditions that can arise. Each section is further divided into compilation instructions and run time instructions.

The operating instructions given will not apply when the compilation is to be performed under the control of GEORGE 1, in this case the GEORGE 1 System Macros may be used, or the programmer may write his own System Macros.

OPERATORS INSTRUCTIONS FOR THE MAGNETIC TAPE COMPILER

Hardware requirement

Processor	Any 1900 Series central processor with floating point facilities, performed either by extracode or hardware, and at least 12032 words of core store.
Basic Peripherals	At least one reader: paper tape or card. Optional listing device: line printer or paper tape punch.
Magnetic Tapes	The Library Tape. One or two named or scratch files for compiler output. Other magnetic tapes to be used for input (optional).

Use of peripherals

LP1	Listing peripheral.
TP1	Alternative listing peripheral.
TR0	Paper tape input.
CR0	Card input.
MT0	FORTRAN Library Tape containing the compiler and associated library SRF7. This tape remains assigned throughout the compilation, since the compiler is overlaid.
MT1	The magnetic tape nominated to receive 'final' semi-compiled output.
MT2	Input magnetic tapes in succession (optional).
MT3	The magnetic tape used as a work tape during the compilation of an overlay program or extended data program.

Note: If a binary dump of the object program is required, it will be made automatically after the compiler has deleted itself to a magnetic tape assigned as MT1.

Priority

The compiler as supplied has a priority of 50.

Operating instructions

Compilation and consolidation: Single program mode

Narrative	*Console message*
1 Load the FORTRAN library tape, and other tapes requested by the programmer. Load the compiler #XFAM into store	FI #XFAM #*yyyy*
where the FORTRAN library tape is called program *yyyy*. This tape remains assigned to #XFAM throughout the run.	
When the compiler is loaded, the message will be output.	O #XFAM; HALTED:-LD
2 Activate the compiler by one of the following:	
(a) begin, reading initially from paper tape	GO #XFAM 20
(b) begin, reading initially from cards in 1900 code	GO #XFAM 21
(c) begin, reading initially from cards in ATLAS code	GO #XFAM 22
A paper tape or card reader is assigned as unit 0 Switching between input peripherals may occur.	
3 The program will be read in, compiled, and in most cases consolidated and loaded into store.	
(a) If there are errors in compilation the compiler will halt	O #XFAM; HALTED:-ZZ
(b) If the SEGMENTS or OVERLAY SEGMENTS statement has been used then no consolidation will take place and an error free compilation will result in	O #XFAM; HALTED:-EC
(c) In the normal case an error free non-overlay program, *#name* say, is compiled, consolidated and loaded into store. The message sequence is	O #XFAM; DELETED:-LO *#name p* O *#name:* HALTED:- LD
where *p* is the number of the deck holding the tape with the consolidated, semi-compiled program.	
(d) With an error free overlay program a binary dump is always produced and messages are	O #XFAM; DELETED:-LO *#name p* O *#name;* DELETED:-LO *#name q* O*#name;* HALTED:–LD
where *q* is the number of the deck holding the binary program and *p* is as in case (c) above	
(e) If the DUMP ON statement is used the program will not be loaded into core (unless RUN is also present). Console messages at the end of compilation will be	O #XFAM; DELETED:- LO *#name p*
The binary dump is then produced and the process terminates with the message	O *#name;* DELETED:-AE

Loading and running the object program: Single program mode

Narrative	*Console message*
4 Load the object program, if not done automatically	FI *#name*
When the object program has been loaded, the message will be output.	O *#name* HALTED:-LD
5 Activate the object program as indicated by the programmer:	
(a) if no TRACE output is required	GO *#name* 20
(b) If TRACE output is required for every statement	GO *#name* 27
(c) if a TRACE steering list is supplied	GO *#name* 28
Load the steering list on the basic peripheral assigned. The steering list will be read in, and the program entered.	
6 On request from the programmer: TRACE output may be suppressed by turning Switch 0 off	OFF *#name* 0
TRACE output may be turned on again by	ON *#name* 0
7 The program will delete itself with the message	0 *#name* DISPLAY:- *abcde* DELETED:-00

Compilation and consolidation: batch processing mode

Narrative	*Console message*
1 Load the FORTRAN library tape and other tapes as requested by the work assembler. Load the compiler into store where the FORTRAN library tape is called PROGRAM *yyyy*. This tape remains allocated to #XFAM throughout the batch.	FI#XFAM*#yyyy*
When the compiler is loaded, the message will be output.	O #XFAM; HALTED:-LD
2 Activate the compiler as requested by the work assembler by one of the following:	
(a) begin, reading initially from paper tape	GO#XFAM 20
(b) begin, reading initially from cards in 1900 code	GO#XFAM 21
(c) begin, reading initially from cards in ATLAS code	GO#XFAM 22
A paper tape or card reader is assigned as unit 0. Switching between input peripherals may occur.	
3 Program will be read in, compiled, and in most cases consolidated. For each program in the batch, the console message will be output, where *m* is a measure of the mill time, and may be used for log analysis, *name* is the four character program identifier and	O #XFAM; DISPLAY:- COMPILED *m name x*

Narrative	*Console message*
x is: EC no errors found SM segments missing ZZ errors found	
4 At the end of the batch the final message is	O#XFAM; HALTED:- END OF BATCH
5 To initiate another batch return to step 2.	

Loading and running the object program: batch processing mode

Narrative	*Console message*
6 Load a compiled object program, by the message where PROGRAM *xxxx* is the name of the file chosen by the work assembler for compiler output.	FI#*name*#*xxxx*
7 Activate the object program:	
(a) if no TRACE output is required from every statement	GO#*name*20
(b) if TRACE output is required from every statement	GO#*name*27
(c) if a TRACE steering list is supplied Load the steering list on the basic peripheral assigned	GO#*name*28
8 On request from the programmer:	
TRACE output may be suppressed by turning Switch 0 off	OFF#*name* 0
TRACE output may be turned on again by	ON#*name* 0
9 The program will delete itself with the message	0 #*name* DISPLAY:- *abcde* DELETED:- 00
10 Repeat from step 6 for each successfully compiled program in the batch.	

Exception conditions

Compilation and consolidation

Message	*Reason*	*Action*
1 O#XFAM; HALTED:-CE	Checksum error when reading semi-compiled program.	If input is on paper tape or cards, the tape should be moved back to the beginning of a block or one card moved back to be re-read. Continue by typing in GO#XFAM If semi-compiled program was being read from magnetic tape, either abandon the compilation run by GO#XFAM 27 or continue the compilation in error mode by GO#XFAM 26 The latter is most useful in a batch compilation when it is not required to terminate the batch by GO#XFAM27. The compilation in error should be repeated

Message	Reason	Action
		from the beginning possibly after the regeneration of the semi-compiled program.
2 O#XFAM; HALTED:- CR LP TR TP	The compiler requires a peripheral that is not available	Make the necessary peripheral available and type in GO#XFAM
3 P#XFAM; HALTED:- NL	The FORTRAN subroutine library has not been found on the library tape. This may be because it is not present on this tape, or has already been skipped in the search for another library.	Abandon the run.
4 O#XFAM; HALTED:- PF	This message occurs after a single program compilation has been abandoned by typing GO#XFAM 27	To initiate another compilation, return to step 2 of Compiling and Consolidating operating instructions.
5 O#XFAM; HALTED:- ST	The compiler requires more core store and none is available.	a) On a multiprogramming processor, delete an idle program and continue by typing GO#XFAM b) On a single programming processor, abandon the compilation by typing GO#XFAM 27 The program is too large and should either be compiled on a machine with more core or be re-written in smaller segments.
6 Operator action is required if the compiler attempts to read from a slow device more input than the user has supplied.	(a) if the message in step 3 of the operating instructions has not been output, it is likely that a FINISH statement has been omitted. (b) If message of the type in step 3 of the operating instructions have been output, then the compiler is working in batch mode and an END OF BATCH statement is very likely omitted.	(a) Simulate a FINISH statement by typing GO#XFAM 28 (b)Simulate an END OF BATCH statement by typing GO#XFAM 27
7 O#XFAM; HALTED: END OF BATCH - OUTPUT TAPE FULL	There is no room for the remaining programs on the output tape.	Batch compile the remaining programs to a different output file.

Loading and running the object program

Message	*Reason*	*Action*
1 O#name; HALTED:- EE	The program run has been terminated. This message occurs on the detection of an execution error or as a result of GO#*name* 29	No operator action required. Post mortem information will be output automatically. The program cannot be restarted.
2 O#*name*;HALTED:- *abcde*	The program has reached the statement PAUSE *abcde* where *abcde* consists of up to five octal characters.	As specified by the programmer.
3 O#name; HALTED:- CR CP LP TR TP	The peripheral required by the program is not available.	Make the necessary peripheral available and type GO#*name*
4 O#*name* ILLEGAL	Illegal occurrence	Input GO#*name*29 Post mortem information will be output and the program will halt with the message O#*name;* HALTED:-EE
5	The program loops indefinitely	Input GO#*name* 29 Post mortem information will be output and the program will halt with the message O#*name;* HALTED:- EE
6 0#*name*; HALTED:- E*x or* 0#*name*; HALTED:-LD ERR-*x* where *x* is one of A, C, I, N, P, Q, R, S, T, U or X	An error has been detected during the loading process	Reload the program; if this fails, recompile the program.
7 0#*name*; HALTED:- TM	Failure when dumping binary program to magnetic tape.	Recompile the program.
8 0#*name*; HALTED:- EM	An attempt has been made to load an EXTENDED DATA program to a machine which is too small.	Abandon the run.

OPERATOR'S INSTRUCTIONS FOR EDITOR #XMUM

Hardware requirement

Processor: Any 1900 processor with floating point facilities, extracode or hardware, and console typewriter.

Core Store: 8512 words

Slow Peripherals: One paper tape or card reader. Switching between slow input peripherals may occur during the course of an #XMUM run.
One line printer
One card punch (optional)
One paper tape punch (optional)

Magnetic Tapes: One or two named tape(s) for #XMUM output
One named tape for #XMUM input (optional)
One named or scratch tape for #XMUM work tape (optional)

Use of peripherals

TR0	Paper tape input
CR0	Card input
TP0	Paper tape punch
CP0	Card punch
LP0	Line printer for listing
MT0	Input magnetic tape
MT1	Output magnetic tape
MT2	Work magnetic tape

Priority
#XMUM as supplied has a priority of 80.

Entry points

GO # XMUM 20	Paper Tape entry
GO # XMUM 21	Card/1900 entry
GO # XMUM 22	Card/ATLAS entry
GO # XMUM 23	Restart
GO # XMUM 28	Abandon the run

Operating instructions

Narrative	*Console message*
1 Load the Library Tape containing the editor #XMUM and one to three other magnetic tapes as specified by the programmer. Load #XMUM into store.	FI#XMUM#TAPE
2 Activate #XMUM by one of the following:	
(a) begin, reading from paper tape	GO#XMUM 20
(b) begin, reading from cards in 1900 code	GO#XMUM 21
(c) begin, reading from cards in ATLAS code	GO#XMUM 22
Switching between input peripherals may occur during the run.	
3 At the end of a run, #XMUM will halt	O#XMUM; HALTED:- OK
4 #XMUM may be restarted for another run at step 2.	

Exception conditions

Message	*Reason*	*Action*
1 O#XMUM; HALTED:- CR TR CP TP LP	The program has tried to allocate a peripheral which is not available	Make the necessary peripheral available and type GO #XMUM
2 O #XMUM; HALTED:- ER	An unusual hardware failure has occurred on the input magnetic tape	Either abandon the run by GO #XMUM 28 or continue past the error by GO #XMUM (some input records may be lost)
3 Operator action required if #XMUM attempts to read from a slow device more input than the user has supplied.	1 If all input from paper tape and cards has been read, it is likely that a batch terminating directive //// has been omitted. 2 If some input has still to be read from the other slow medium it is likely that a user	1 Supply a batch terminator consisting of //// in columns 1 to 4. 2 Satisfy the transfer by feeding in a blank card or newline character.

Message	*Reason*	*Action*
	has omitted a blank card or a line following a *SWITCH directive.	
4 O#XMUM; HALTED:–ZZ	The output magnetic tape has for some reason not been finished in good order. The reason for this (given on the listing) could be magnetic tape failure or an input magnetic tape in the wrong format.	Abandon.

OPERATORS INSTRUCTIONS FOR EDITOR #XKYA

Hardware requirement

6912 words of core store

1 paper tape reader or 1 card reader

1 or more magnetic tape decks (maximum 6)

1 paper tape punch or 1 card punch (optional)

1 line printer

Use of peripherals

TR0 or CR0	parameters and data	The reader is assigned at the start of the run and released at the end.
LP0	error codes and messages	The printer is assigned at the start of the run and released at the end.
TP0 or CP0	subfile copies (optional)	The punch is assigned in the program and released at the end of the run.
MT1	output data	The file is opened when specified and closed at the end of the run.
MT0	main input tape	The file is opened when specified and closed at the end of the run.
MT2 to MT5	input tapes (optional)	The files are opened when specified and closed at the end of the run unless specifically closed earlier.

Priority

#XKYA as supplied has a priority of 80

Operating Instructions

Narrative	*Console message*
1 Load the library tape containing #XKYA.	
2 Load the input and output magnetic tapes.	
3 Load the editing file in a card or paper tape reader.	
4 Load #XKYA into core store with the message:	FI #XKYA #TAPE
5 When loading is completed the following message will be output:	HALTED:- LD
Begin the run by typing:	

Narrative	*Console message*
(a) if the editing file is on paper tape:	GO #XKYA 20
(b) if the editing file is on cards:	GO #XKYA 21
6 When the run has been completed without error, the following message will be output:	0#XKYA; DELETED 99

Exception conditions

Message	*Meaning*	*Action*
1 0#XKYA; HALTED:-CP	A card punch is required.	When a card punch is available, restart the run by inputting: GO #XKYA
2 0#XKYA; HALTED:-TP	A paper tape punch is required.	When a tape punch is available, restart the run by inputting: GO #XKYA
3 0#XKYA; HALTED:- NOT ENOUGH CORE	More core store is required.	When sufficient storage space is available, restart the run by inputting: GO #XKYA
4 0#XKYA: DELETED:-ME	The run has been completed but minor errors have occurred. These are indicated separately on the line printer.	
5 0#XKYA: DELETED:-OH	A major error has caused the run to halt prematurely. The reason is indicated on the line printer.	
6 0#XKYA; DISPLAY:- MT*n* READ ERROR	A magnetic tape failure has been detected when reading the input tape MT*n*; *n* is the unit number.	The run is automatically abandoned, and deleted OH is output.
7 0#XKYA; DISPLAY:- MT WRITE ERROR	A magnetic tape failure has been detected when writing on MT1.	The run is automatically abandoned, and deleted OH is output.

Appendix 1 Compilation error numbers

Error number	*Interpretation*
1	Mis-matched parenthesis
2	Missing parenthesis
3	/), expected, but another character found.
4	Error in format of expression.
5	Constant out of range.
6	Integer variable expected, but something else found.
7	Array, not declared, or declared twice.
8	First character of name non-alphabetic, or name omitted.
9	Statement too long.
10	Statement incomplete.
11	Label set twice or error in columns 1 to 5.
12	Error in label references.
13	Incorrect subscript: for example, too many or too few subscripts. In particular, the array name may have been used as an unsubscripted variable.
14	Incorrect name, e.g. subroutine has incorrect name.
15	DO label not found.
16	Label missing.
17	Statement not recognised or out of context.
18	Error in compiling system statement.
19	Expression too complicated or Too many overlay segments (as a guide this may occur if more than 150 segments are specified).
20	Compiler error (should not occur).
21	Expression on the left hand side of an arithmetic assignment statement or In a relational expression "=" has been used instead of ".EQ."
22	Array not declared in a declaration statement or Statement Function definition out of place.
23 - 25	Reserved.

Error number	*Statement*	*Interpretation*
26	Initial statements	Incompatible use of file or subfile names e.g. file specified in SEND TO statement is same as file specified in DUMP ON statement.
	CALL	Subroutine called has same name as the segment containing the CALL statement.
	FUNCTION	Segment name omitted.
	Function reference	Function reference has the same name as the master or subroutine segment which contains the function referenced.
	SUBROUTINE	Segment name omitted.
	GO TO	Error in GO TO statement following the characters GO TO.
	DO	DO statement incomplete.
	IF	DO statement in 'logical IF'.
	FORMAT	Unlabelled.
	READ WRITE	Channel number not integer constant or integer variable.
	INTEGER REAL LOGICAL DOUBLE PRECISION COMPLEX	Type already specified for same item or type statement in wrong position.
	COMMON	Incorrect name for common block
	DIMENSION	No dimensions given
	EQUIVALENCE	Dummy argument in Equivalence List
	DATA	Incorrect name in list e.g. a subroutine name.
27	FUNCTION	No arguments after function name
	Function reference	An argument of the function has the same name as the master or subroutine segment in which the function is referenced.
	GO TO	Error after) of 'computed GO TO'.
	CALL	An argument of the subroutine called has the same name as the master or subroutine segment in which the CALL statement appears.
	READ WRITE	Format reference error.
	COMMON	Incorrect item in list e.g. dummy argument.
	EQUIVALENCE	Incorrect item e.g. constant or segment.
	DATA	Incorrect form of constant.

Error number	*Statement*	*Interpretation*
28	DO	Parameters not integer expressions.
	READ WRITE	Error in list
	EQUIVALENCE	Array in list not declared as such.
	DATA	Incorrect form of complex constant.
29	EQUIVALENCE	Item already equivalenced or otherwise defined so that equivalence is impossible.
	DATA	Too may or too few constants in list.

Error number	*Interpretation*
31	MT0 has been declared in an overlay program.
35	Program too large: either the lower storage area exceeds 4K, the core requirement exceeds 32K in compact mode or the core requirement exceeds 256K in extended data mode.
36	Incomplete program: either an overlay segment has been omitted, or a source program description segment has been omitted or is not the first segment imput.
37	Reserved.
38	Subfile of the correct name and containing Fortran source or Semi-compiled not found.
39	File name has been omitted from the first READ FROM (MT,) statement.
40	The operator has intervened wrongly, e.g. the operator has typed GO #XFAM 24.
41	A block has been read from magnetic tape containing more than 128 words.
42	A semi-compiled segment of the wrong mode has been read prior to a FINISH or LIBRARY statement i.e. a segment marked as suitable for extended data mode is being included in a compact data program, or vice versa.
43	Output tape full i.e. end of tape detected on writing to an output tape. In batch mode a message Halted 'End of batch - output tape full' will be printed.

Appendix 2 Execution error numbers

This table gives the interpretation of run-time error numbers. Those marked with a '*' are *fatal*. In the event of a fatal error, the program is forcibly halted by the system even if the program has passed through an error trap (see Chapter 7).

	Error number	*Interpretation*
	0'*x*'	This error number indicates that a prohibited character *x* has been found in an input field.
	01	Array subscript error. The generated address of an array-element is outside the bounds of the array. This error check is made only if the program is compiled in TRACE 2 mode.
*	02	The TRACE steering-list has incorrect format or is too long.
*	03	The overlay depth has been exceeded.
*	04	Reserved.
	05	Reserved.
*	06	Reserved.
*	07	Not enough arguments have been supplied for the subroutine called. This error is detected only when the object program is an overlay program.
*	08	This error can occur only in mixed-language programs. Too many arguments have been specified for a PLAN segment.
	09	Reserved.
*	10	Error in the structure of a format specification, e.g. field separator missing, character out of place. unrecognized character found.
*	11	'F' or 'E' conversion code is not associated with REAL or COMPLEX variable.
*	12	'I' conversion code is not associated with an INTEGER variable.
*	13	'G' conversion code is not associated with a REAL or COMPLEX variable.
*	14	'L' conversion code is not associated with a LOGICAL variable.
*	15	'D' conversion code is not associated with a DOUBLE PRECISION variable.
*	16	Zero or undefined field width when using the A,L,H, or X codes.
*	17	When using the 'L' code, either whole of the input-field is blank, or the first non-space character is neither T nor F.
	18,19	Reserved.
*	20	The channel number used in the program has not been associated with a peripheral in the program description.
*	21	A formatted input record is not long enough.
*	22	An unformatted input record is not long enough.
*	23	Illegal operation attempted, e.g. read from card punch, or write to an input magnetic tape.
	24 to 26	Reserved.
*	27	An attempt has been made to use RENAME with an input file.
*	28	An attempt has been made to use FILE or RENAME but a dummy filename has not been given in the program description statement.

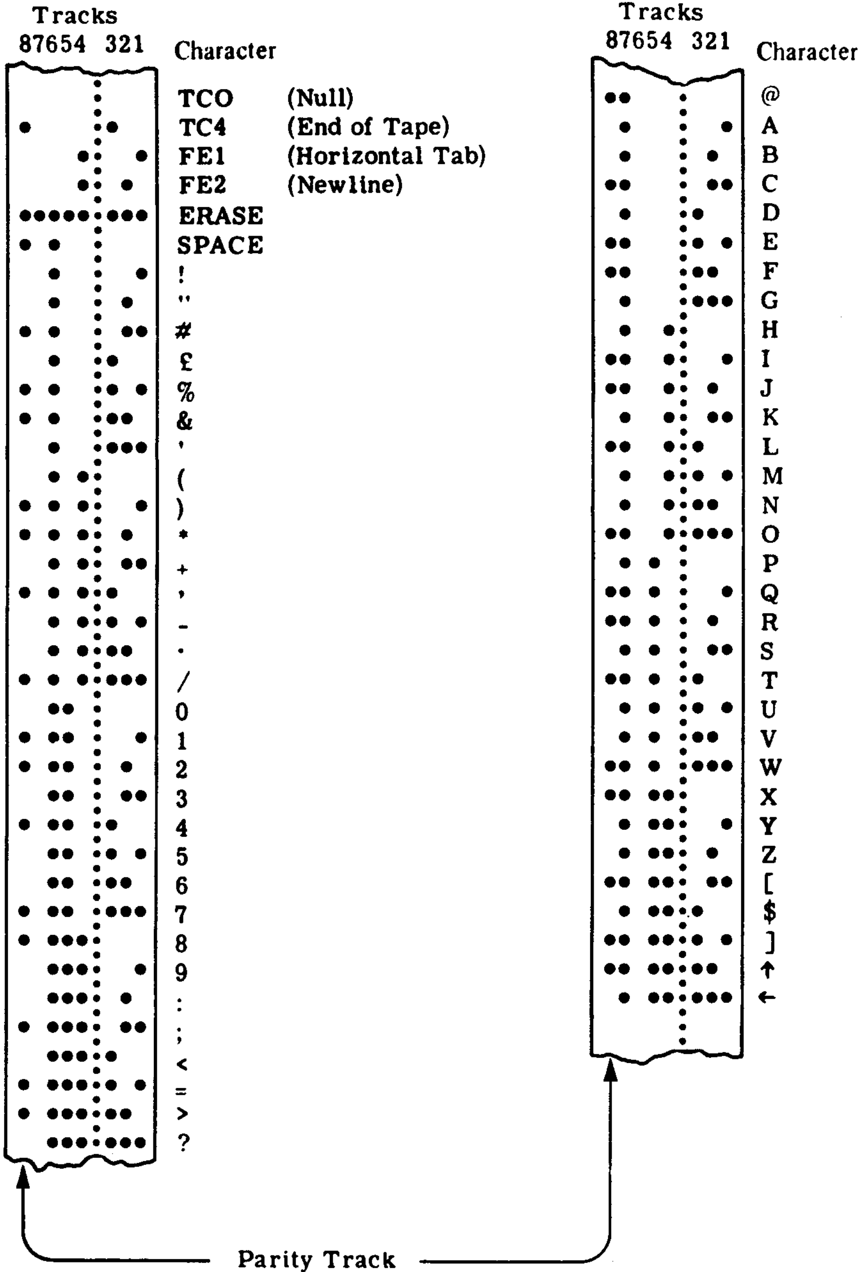
Error number	*Interpretation*
* 29	An attempt has been made to use FILE when the file is already open.
* 30	Illegal sequence of operations on serial file, e.g. read after write, read or write after ENDFILE.
* 31	No filename specified for input device.
* 32	Input file has incorrect format, e.g. no start-of-date sentinel found when opening input magnetic tape.
33	Reserved.
* 34	End of tape mark detected when writing to magnetic tape.
* 35	Failure of data block-count check (unformatted magnetic tape).
* 36	An attempt has been made to write a formatted record longer than the maximum permitted block or bucket size.
* 37	Block-size on input file exceeds buffer size specified in the program description.
* 38	Reserved.
* 39	An attempt has been made to read a formatted record longer than the available buffer or the bucket size in an unformatted file exceeds the buffer-size.
40	Reserved.
* 41	Failure of integrity-code check on opening EDS/FDS files.
* 42	Purge-date not exceeded on opening output EDS/FDS files.
* 43	On: opening a scratch-file, extending a file, renaming a file } system control-area full
* 44	On: opening a scratch-file, extending a file, renaming a file } no space available
* 46	On: opening a scratch-file, extending a file, renaming a file } Specified file not opened as a write file. (Should occur only when renaming)
* 47	Argument outside the permitted range in FERROR.
48 & 49	Reserved.
50	Overflow has been set doing simple arithmetic
* 51	FTRAP used with TRACE0.
* 52	FRESET used with TRACE0.
53	Result exceeds capacity in EXP.
54	Result exceeds capacity in DEXP.
55	Non positive argement in ALOG.
56	Non positive argument in DLOG.
57	Negative argument in SQRT.
58	Negative argument in DSQRT.
59	Attempt to form ATAN2 (0/0).
60	Attempt to form DATAN2 (0/0).
61	Attempt to form ISIGN (x,0).
62	Attempt to form SING (x,0).
63	Attempt to form DSIGN (x,0)
64	Attempt to find MOD (x,0)

Error number	*Interpretation*
65	Attempt to find AMOD (*x*,0).
66	Attempt to find DMOD (*x*,0).
67	Argument 1 in ACOSH
68	Argument 1 in ATANH.
69	Argument 1 in ACOTH.
70	Argument 1 in ASIN.
71	Argument 1 in ACOS.
72	Result exceeds capacity in TAN.
73	Result exceeds capacity in CABS.
74	Results exceed capacity in CSIN.
75	Results exceed capacity in CCOS.
76	Argument too large to be held as an integer in NINT.
77	Result too large to be held as in integer in IFIX.
78	Result too large to be held as in integer in INT.
79	Trace list too short.
* 80	Error in the information block used by GENAH.
81 to 97	Reserved.
* 98	Misuse of monitor channel (for example, BACKSPACE on channel 0).
99	Reserved
* 100	The object program has reached an END statement.
101 to 104	Reserved.
105	Attempt to read beyond end of data on card input channel.
* 106	Attempt to read beyond end of data on paper tape input channel.
* 107	End of file encountered on magnetic tape input.
* 108	End of file encountered on serial disc (E.D.S. or F. D. S.) input.
109 to 113	Reserved.
* 114	A CALL to an overlay segment with arguments has caused the calling segment to be overwritten.
115 to 120	Reserved.
* 121	Attempt to write more than one record or too long a record to a DEFBUF array with a single WRITE.
122 to 499	Reserved.

Appendix 3 ICL 1900 Series codes

This appendix lists all the characters used for FORTRAN programs and data.

8-TRACK PAPER TAPE CODE



1900 CARD CODE

Character	*Card punching*	*Character*	*Card punching*	*Character*	*Card punching*
0	0	F	10/6	–	11
1	1	G	10/7	"	11/0
2	2	H	10/8	/	0/1
3	3	I	10/9	+	10/2/8
4	4	J	11/1	.	10/3/8
5	5	K	11/2	;	10/4/8
6	6	L	11/3	:	10/5/8
7	7	M	11/4	'	10/6/8
8	8	N	11/5	!	10/7/8
9	9	O	11/6	[	11/2/8
space	none	P	11/7	$	11/3/8
&	10 or 10/0	Q	11/8	*	11/4/8
#	3/8	R	11/9	>	11/5/8
@	4/8	S	0/2	<	11/6/8
(	5/8	T	0/3	↑	11/7/8
)	6/8	U	0/4	£	0/2/8
]	7/8	V	0/5	,	0/3/8
A	10/1	W	0/6	%	0/4/8
B	10/2	X	0/7	?	0/5/8
C	10/3	Y	0/8	=	0/6/8
D	10/4	Z	0/9	←	0/7/8
E	10/5				

ATLAS CARD CODE

ATLAS character	*Card punching*	*1900 FORTRAN interpretation*	*ATLAS character*	*Card punching*	*1900 FORTRAN interpretation*
0 to 9	As 1900 code	0 to 9	.	10/3/8	.
A to Z	As 1900 code	A to Z	)	10/4/8	)
space	none	space	>	10/5/8	>
=	3/8	=	– (underline)	10/6/8	@
!	4/8	!	| (vertical bar)	10/7/8	!
&	5/8	&	π	11/3/8	$
:	6/8	:	*	11/4/8	*
document terminator	7/8	]	?	11/5/8	?
			]	11/6/8	]
+	10	+	[	11/7/8	[
–	11	–	,	0/3/8	,
/	0/1	/	(	0/4/8	(
			<	0/5/8	<

4159(1.71)

Appendix 4 Program and data formats on magnetic tape files

Details of program formats on magnetic tape files are given in the manual *Compiling Systems.*

The format of FORTRAN formatted data files on magnetic tape is the same as that used for EMA and Algol. Unformatted files are intended for use only in FORTRAN programs. Both types of files may be created and read by the 1900 Series Magnetic Tape Housekeeping Package.

FILE STRUCTURE

Any FORTRAN magnetic tape file, either formatted or unformatted, consists of the following items in the order shown:

Header Label
Start of data sentinel
One or more data blocks
Trailer label

Only single reel files are allowed. Start of data sentinels and trailer labels are in standard 1900 format.

The utility program #XRMF (see the manual *Magnetic Tape Utilities*) may be used to print formatted files.

FORMATTED DATA BLOCKS

A formatted block consists of one or more records, each record having a format as follows:

Word 0 Number of words in the record

Word 1 Carriage control character in the least significant position. This will normally have the value # 41 although the first record in a file may have the value #51 in this position.

Word 2 onwards Record information space filled to give an integral number of words.

Notes:
1 A record may not be split over two blocks.
2 A minimum block size of five words is always output, and zero words will be added to produce this minimum block size where necessary.
3 The carriage control character is used for off-line printing.

UNFORMATTED DATA BLOCKS

An unformatted data block may contain all or part of one record only, having the following format:

Word 0 Number of words in the block.

Word 1 Data block sequence number relative to the start of the tape, starting at 1.

Word 2 Data block sequence number relative to the start of the record, starting at record 1.

Word 3 Number of words of information following. The sign bit of this word is set in the last block of each record.

Word 4 onwards Record information.

If a record output to an unformatted file is large than the size of buffer specified, then the record will be split over two or more blocks. If the record is smaller, than a block of sufficient size to hold the record will be output. Note that each block of a FORTRAN unformatted record may be treated as a complete record if the file is processed by the Magnetic Tape Housekeeping Package.

Index

8-track paper tape code 101
*1900 54
1900 card code 102

Acceptance rules 26
*ALTER 53
Arrays 66
Assembling a batch 48
*ATLAS 54
ATLAS card code 102

BACKSPACE statement 9
Batch
  compilation 2, 81-84
  editing system 48
  input media 83
  system statements 81-83
Basic peripherals 8
  compiling from 21
Binary program 17
Blank common areas 70
Block
  data segments 70
  size 10

Calling
  FORTRAN segments from PLAN 72
  PLAN segments from FORTRAN 63
  sequence 63
Cards in ATLAS code 54
Channel description statements 7
  magnetic tape 9
Channel
  numbers 7
  0 12
Character transfers in core store 12
Common storage areas 68
Communication of data in mixed PLAN/FORTRAN programs 63
Compact programs 43
COMPACT statement 43
Compilation 79
  error diagnostics 27
  error numbers 93
  of overlay program 39
  with errors 19
Compiler input 21
  media 21
Compiler listing
Compiler output 17
  suppression of 20
Compiling and running on different size processors 43
Compiling from
  basic peripherals 21
  magnetic tape 21
Compiling mixed FORTRAN/PLAN programs 75
Compiling system statements 1
  effect of 84
  for a single program 3
  format of 2
  initial 79
  on magnetic tape 23
Composite files 5
COMPRESS statement 44
DOUBLE PRECISION 45
INTEGER AND LOGICAL 44
Configuration
Console typewriter 12
Consolidated semi-compiled program 17
Consolidation 79
*CPCOPY 56
CREATE 10
Creating a batch 48

*DATA 54
Data formats on magnetic tape files 103
Default listing mode 80
DEFBUF 12
*DELETE 53
DEPTH OF OVERLAY statement 38
Directive
  end of batch 52
  end of job 56
  start of job 53
Disc
  editor output to 49
  overlaying from 40
Double buffering 8
Dump and restart 7, 10
DUMP ON statement 17

Editing 47-62.1
  file 48
  of source programs on magnetic tape 24
ENDFILE statement 9
End of batch directive 52
End of file marker 11
End of job directive 56
END statement 1
Entry points 75
Error
  diagnostics 27
  messages 27
  trap facility 29
Error numbers

compilation 93
execution 97
Extended data programs 43
EXTENDED DATA statement 43

FEPILOG subroutine 65
File names 5
at run time 10
File renaming 11
FINISH statement 2
Formatted
magnetic tape files, printing of 11
records 10
*FORTRAN 53
FORTRAN compiler library 24
FOVER subroutine 73
FPROLOG subroutine 64
FRESET subroutine 30
FTRAP subroutine 29
FULL listing 15

GENAH subroutine 67
Generation number 5
GEORGE 1 and 2 77
GETAH subroutine 66

*IN 51
Incorporation of library subroutines into a program 24
Initial
input medium 21
statements 1,17
Input
compiler 21
from simple files 23
from subfiles 22
object program 7
sequence 2
INPUT statement 9
Intersegment statements 2

Label errors 27
*LAST
LEADER statement 76
Libraries
FORTRAN compiler 24
other system 24
held on other tapes 25
paper tape and card 25
recommended order of search 25
Line editor 57, 58, 61
*LIST 54
LIST 15
Listing
compiler 15
default 15, 80
description 15
full 15
semi-compiled 54
short 15
statements 15
subfile 50

Load and go 17
Log analysis message 80

Magnetic tape
channel description statements 9
files 5, 8
files, opening 9
files, programs held on 47
formats 103
Main editor 57, 58
Master segments 75
Missing segments 27
Mixed
error detection levels 33
language programs 43, 63-76
PLAN/FORTRAN overlaid programs 73
MIXED SEGMENTS statement 43

Named files 5
Nested subfile structure 5
*NOCOMPILATION 56
*NOLIST 54
LIST 15

Object program
error diagnostics 27
input/output 7
*OFFLINE 52
OMIT statement 44
Opening magnetic tape files 9
Operator intervention at program run time 33
Operator's guide to the FORTRAN magnetic tape compiler 85-90
Operator's instructions
for the magnetic tape compiler 85
for the Editor #XKYA 92
for the Editor #XMUM 90
*ORDER 52
*OUT 51
Output
compiler 17
object program 7
OUTPUT statement 9
Overlay 37-41
area 37
system 37
units 37
Overlaying from disc 40
OVERLAY PROGRAM statement 1, 38
OVERLAY SEGMENTS statement 1, 38
OVERLAY statement 37

Paper tape and card libraries 25
Peripheral usage 75
Permanent area 37
PRIORITY statement 44
Program
description statements 1
description statements for overlay programs 37
formats on magnetic tape files 103
held on magnetic tape files 47
structure 1

testing 27-35
PROGRAM statement 1

READ FROM statement 21
READ statement 9
Record format 10
RENAME 11
Restart
dump and 7, 10
editor facility 49
*RESTART 52
Restrictions in overlay programs -39
Retention period 6
REWIND statement 9
RUN statement 18
Run time efficiency 39

Scratch files 5
SEGMENTS statement 1
*SELECT 51
*SEMI 54
Semi-compiled segments 17, 48
SEND TO statement
for consolidated semi-compiled program 18
for semi-compiled segments 18
SHORT LIST 15
SHORT LIST (TP) 15
Short listing 15
Simple files 5
Source program statements for magnetic tape files 8
Standard functions in overlays 39
Start of job directive 53
Statements
batch system 81
channel description 1
compiling system 1
initial 1, 17, 82
intersegment 2
listing 15
program description 1
source program 8
terminator 2
STATEMENT TRACE output 30
*STEER 54
Subfile 5
listing 50
names 5
Subprogramming 75
Suppression of
trace output 32
transfers 13
*SWITCH 54
System descriptions 79-80

Tape usage and efficiency 79
Terminator 2
*TPCOPY 56
Trace
level 0 33
level 1 28-30
level 2 30-32
output channel 28
steering lists 31
steering lists, format of 32
Tracing 28
Transfer
of control 63
suppression of 13

Unformatted records 10
USE statement 10

Word 30, use of 75
of #XFAM 80
*WORKFILE 52
Work assembler 48, 81
Write permit ring 6
WRITE statements 9

#XKYA 57-62.1
#XMUM 47-56

Amendment record

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

List number	Item number
1	1 to 13
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to

ICL

1900 Series

FORTRAN: Magnetic Tape Compiler

First Edition July 1969
Amendment list 1
incorporating User Notices 2 to 5.

Each amendment list contains one or more numbered instructions to replace one or more existing pages or to add one or more new pages.

When a page is amended, significant technical changes on the re-issued page will be indicated by a vertical line in the margin against the changed passages. Any lines on the re-issued page which remain from a previous amendment, will be removed. New chapters or completely revised chapters will not be marked with amendment lines.

The date of issue appears at the foot of all new pages and re-issued pages in the form (month, year).

1	Preface and Contents	Remove and destroy pages iii to xiii. Insert new pages iii to xi.
2	Chapters 2 and 3	Remove and destroy pages 5 to 13. Insert new pages 5 to 13.
3	Chapter 6	Remove and destroy pages 25 and 26. Insert new pages 25 and 26.
4	Chapter 7	Remove and destroy pages 29 and 30. Insert new pages 29 and 30.
5	Chapter 7 to 9	Remove and destroy pages 33 to 44. Insert new pages 33 to 44.
6	Chapter 10	Remove and destroy pages 47 to 61. Insert new pages 47 to 62.1.
7	Chapter 11	Remove and destroy pages 65 and 66. Insert new pages 65 and 66.
8	Chapter 11	Remove and destroy pages 73 and 74. Insert new pages 73 and 74.
9		Insert new part divider page before page 79.
10	Chapter 13	Remove and destroy pages 79 and 80. Insert new pages 79 and 80.
11	Chapter 15 to Index	Remove and destroy pages 85 to 111. Insert new pages 85 to 107.
12		Remove and destroy User Notices 2 to 5.
13		Update the amendment record and file this list at the back of the manual.

4159(1.71)

© International Computers Limited
ICL House, Putney, London, S.W.15