Fortran

AMENDMENT RECORD

List Number	*Item Numbers*
	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
	to
	to
	to

Technical Publication TL 1167

Reprinted October 1966
(incorporating Amendment List No. 1)

Reprinted February 1967
(incorporating Amendment Lists 1 and 2)

Issued by Technical Publications
International Computers and Tabulators Limited
Head Office : I.C.T. House, Putney, London, S.W.15
and printed in Great Britain.

CONTENTS

PREFACE

FORTRAN is an automatic programming language that is particularly suitable for problems of a mathematical or scientific nature.

Developments of the original FORTRAN language have given rise to several versions, broadly classified into two groups, FORTRAN II and FORTRAN IV. In an attempt to standardize the language, the American Standards Association (A.S.A.) has published a draft standard* which has been submitted to the International Standards Organization. This standard defines two versions of the language, FORTRAN (similar to most FORTRAN IV implementations) and Basic FORTRAN (equivalent in power to FORTRAN II).

The first version of FORTRAN implemented on 1900 series computers, described in the 1900 Basic FORTRAN manual, was developed before the publication of the standard. It is equivalent in general scope to A.S.A. Basic FORTRAN although it also has some A.S.A. FORTRAN features.

The present manual describes a later development, 1900 FORTRAN, the 1900 series implementation of the FORTRAN described in the A.S.A. standard. However, there are some extensions, and also some slight restrictions are required to obtain what is basically a one-pass compiler. These differences have been indicated in the text.

This manual is intended for reference purposes. The reader is expected to be familiar with the principles of FORTRAN or with a similar mathematical language and to appreciate that a source program must be compiled. The manual is written in three parts: the first part deals with the source language statements (excluding input/output); the second part deals with FORTRAN input and output facilities; the third part deals with the 1900 FORTRAN compiling system.

* COMMUNICATIONS OF THE A.C.M., VOLUME 7, NUMBER 10, OCTOBER 1964

Part 1

Source Language

(Excluding Input/Output)

CONTENTS

Chapter 1

BASIC ELEMENTS

This chapter gives general information applicable to all aspects of the language, introduces some of the basic terminology, and shows the broad structure of a 1900 FORTRAN program. Some of the features introduced here will be dealt with in detail in later chapters.

CHARACTER SET 1.1

The following character set is used in 1900 FORTRAN:

A to Z

0 to 9

\+

\-

*

/

=

(

)

,

.

Space

No other character may be used in the program except in TEXT (see 2.1.6) and in an H specification in a FORMAT statement (see Part 2).

NAMES 1.2

A name is a string of letters and digits used to identify items such as variables, arrays and functions. The first character of a name must be a letter. Spaces normally have no significance in a FORTRAN program; so, to the compiler, the names A M1, AM1 and AM 1 are identical.

In general, names may be freely chosen but the first character may have special significance (see 3.1). A name may have only one meaning within a program segment. The same name used in different segments does not necessarily refer to the same quantity. Common block names are an exception to both these rules (see 3.5).

In standard FORTRAN, names may contain from one to six alphanumeric characters; in 1900 FORTRAN, they may contain up to 32 alphanumeric characters.

STATEMENTS 1.3

A FORTRAN program is written on a FORTRAN coding sheet. The program is then punched on cards or paper tape for input to the computer. An example of a program written on a FORTRAN coding sheet is given at the end of this chapter.

A program comprises a series of statements, which are classified as executable and non-executable. Executable statements specify the actual operations to be performed; non-executable statements provide the compiler with information regarding the characteristics and arrangement of data, the structure of the program, and so on.

A statement consists of an initial line followed where necessary by up to 19 continuation lines.

Lines 1.3.1

A line consists of a string of 72 character positions. The character positions are called columns and are numbered consecutively from 1 to 72. An initial line contains a space or the digit 0 in column 6. A continuation line contains any character except 0 or a space in column 6 but it is conventional to number continuation lines consecutively from 1. A continuation line may not be labelled.

Comment 1.3.2

The letter C in column 1 of a line designates that line as a comment line. A comment line does not affect the program in any way and permits the programmer to include explanatory text in the program. A comment may not have any continuation lines, so where there are consecutive lines of comment each one must be preceded by the letter C.

Statement Labels 1.3.3

Any executable statement may be labelled so that it can be referred to in other statements.

A statement label may be any number from 1 to 99999 written in columns 1 to 5 of an initial line. The numbers have no sequential significance; for example, a statement labelled 2 may follow a statement labelled 97384. As usual,

spaces have no significance and labels may be right or left justified or have spaces between the digits.

DATA VALUES AND TYPES 1.4

Various FORTRAN quantities have values. There are six types of values as follows:

INTEGER

An INTEGER quantity can take whole number values in the range

$$-8388607 \text{ to } 8388607$$

and it is recorded exactly in fixed-point form.

INTEGER quantities are used chiefly in the organization of calculations as counts or as subscripts. The calculations themselves are usually performed with REAL quantities.

REAL

A REAL quantity can take values in the approximate range

$$-5.6 \times 10^{76} \text{ to } 5.6 \times 10^{76}$$

and it is recorded in floating-point form to a precision of approximately 11 significant digits. Most data and results in a FORTRAN program are held in REAL form.

DOUBLE PRECISION

A DOUBLE PRECISION quantity is a floating-point value in the range

$$-5.6 \times 10^{76} \text{ to } 5.6 \times 10^{76}$$

as above, but precision is increased to about 20 significant digits.

COMPLEX

A COMPLEX quantity consists of two REAL values as defined above. These two values correspond to the real and imaginary parts of a complex number.

LOGICAL

A LOGICAL quantity can take one of the values

.TRUE.

.FALSE.

TEXT

A TEXT quantity is a string of characters.

An executable program consists of a Master segment and possibly one or more other segments of the three types: Function segment, Subroutine segment and Block Data segment.

The Master segment is the controlling segment of the program or may be the whole program. Execution of the program starts at the first executable statement of this segment and the segment may subsequently perform calculations, initiate input/output operations, and transfer control to other segments.

A Function segment can be used whenever a particular function is often repeated in the program. The function is named and the statements required to evaluate it are written using 'dummy' arguments. Control will be transferred to this segment whenever the name of the function is encountered with a list of arguments to replace the 'dummy' arguments. A function must have at least one argument.

Certain standard functions, such as SIN and LOG, are provided and these are listed in Appendix 1.

A Subroutine segment is similar to a Function segment but it must be called in by a CALL statement. A subroutine may have any number of arguments, including none.

A Block Data segment is used to provide initial values for selected variables (normally those which are referred to in more than one segment).

Function and Subroutine segments are termed Procedure segments and may contain both executable and non-executable statements but a Block Data segment may contain only non-executable statements. (The term Subprogram, often used in descriptions of FORTRAN, is not used in 1900 FORTRAN as it has another meaning for 1900 series machines.)

A detailed description of each type of segment is given in Chapter 7.

TL 1167

TITLE A.C. CURRENT CALCULATION **I.C.T. FORTRAN CODING SHEET** SHEET 1 / 1

PROGRAMMER J. HEFFERAN DATE 12 / 2 / 66

C	STATEMENT NUMBER	CONT.	FORTRAN STATEMENT	IDENTIFICATION AND SEQUENCE No.
			MASTER CURRENT	
			REAL L, I	
			READ (2,7) E,R,L,C	
	25		READ (2,7) F	
	7		FORMAT(4F10.0)	
			FACT = 2.0*3.14159265356*F	
C			TEST FOR END CONDITION	
			IF (FACT) 9,9,8	
	8		I= E/SQRT(R**2 + (FACT*L-1./(FACT*C))**2)	
			WRITE (3,10) F,I	
	10		FORMAT (2E20.8)	
			GO TO 25	
	9		STOP	
			END	
			FINISH	

FORM 1/542 (8.64)

DATA

Three forms of data may be processed by a FORTRAN program: constants, variables and arrays. Constants appear as fixed values in the program. Variables are referred to by names devised by the programmer and their values may change during the course of a program. An array is a set of similar items assigned a single name. Individual elements of an array are similar to variables and are referred to by the array name followed by one or more subscripts.

CONSTANTS 2.1

Constants may be of the six types:

INTEGER, REAL, COMPLEX, DOUBLE PRECISION, LOGICAL or TEXT.

The range of values for numerical constants is given in 1.4.

INTEGER Constants 2.1.1

An INTEGER constant is a whole number. It may be signed or unsigned and is written as a string of digits. No decimal point, comma or exponent may appear. If no sign appears, the constant is assumed to be positive.

EXAMPLES

29	-1760
999999	+ 144

REAL Constants 2.1.2

A REAL constant is a number that contains a decimal point, a decimal exponent or both. It can take any of the following forms:

±n.	±n.n	±.n
±n.E±s	±n.nE±s	±.nE±s
±nE±s		

where n is a string of decimal digits, E indicates exponentiation to a power of 10, and s is a whole number. Thus, N.n E-s means n.n x 10^{-s}.

Precision is to about 11 significant digits; more may appear but only the first 11 contribute to the value. Spaces are not significant. Plus signs may be omitted.

EXAMPLES

298. +37.6 -.33334
1234E7 +98.4E-16

NOTE: A REAL constant with an integer value and the corresponding INTEGER constant are not identical; the use of, say, 9589. instead of 9589 may well produce a different result.

DOUBLE PRECISION Constants 2.1.3

A DOUBLE PRECISION constant is a number with one of the following forms:

±n.D±s ±n.nD±s ±.nD±s
±nD±s

It can be seen that these are similar to some of the above REAL constant forms but E is replaced by D. Precision is to about 20 significant digits.
Thus, while n.nE-s means n.n x 10^{-s} to a precision of about 11 significant digits, n.nD-s means n.n x 10^{-s} to a precision of about 20 significant digits. Spaces are not significant. Plus signs may be omitted.

EXAMPLES

0. 5D-56 76D6 .31D+3
1.02030405060708D-15

COMPLEX Constants 2.1.4

The form of a COMPLEX constant is

(A , B)

where A and B are REAL constants as defined above, corresponding to the real and imaginary parts of a complex number.

EXAMPLES

(3.4 , 2.) (-4.0 , 2.34)

These are equivalent to (3.4 + 2i) and (-4 + 2.34i) in the usual notation.

LOGICAL Constants 2.1.5

There are only two LOGICAL constants:

.TRUE.

.FALSE.

They are used in logical expressions (see 4.2).

TEXT Constants 2.1.6

The form of a TEXT constant (sometimes known as a Hollerith constant) is

n H s

where s represents a string of characters and n indicates the number of characters. Space characters are significant and should be included when determining the value of n. The characters may be any in the FORTRAN character set or any of the additional characters in the 1900 set. TEXT constants can appear only as the actual arguments in subroutine references or in DATA statements.

EXAMPLES

4HRATE	8HICT-1900
7HHEADING	10HI C T 1900

Note that the spaces in the last example have all been included in the count of the characters. Further information on the use of TEXT is given in 3.8.

VARIABLES 2.2

A variable is a single item of data whose value can be changed in the course of a program. Each variable is referred to by a name assigned by the programmer according to the rules given in Section 1.2.

A particular variable is normally available in only one segment. A name used for a variable in one segment may be used for a different variable or even for a quite different kind of item in another segment.

There are five different types of variable: INTEGER, REAL, DOUBLE PRECISION, COMPLEX and LOGICAL. The possible ranges of values of these types of variable are as given in Section 1.4. A variable of any type may be used to hold TEXT data (see 3.8). The type of a variable is decided either by its name or by a Type statement (see 3.1).

EXAMPLES

MASS	COEFF1
RATIO	X6

ARRAYS 2.3

An array is a group of elements similar to variables. Each array is referred to by a name assigned by the programmer according to the rules of Section 1.2. A particular array is normally available in only one segment. A name used for an array in one segment may be used for a different array or even for a quite different kind of item in another segment.

There are five different types of array: INTEGER, REAL, DOUBLE PRECISION, COMPLEX and LOGICAL. The type of an array is decided either by its name or by a Type statement (see 3.1). Array elements are similar to variables and all elements of a particular array are of the same type as the array itself. An array of any type may be used to hold TEXT data (see 3.8).

An array has one or more dimensions. The number of dimensions and their sizes must be declared (see 3.3).

EXAMPLES

TABLE	LIST
MAT1	VECTOR

Subscripts 2.3.1

In some contexts, a whole array is referred to by name. More often, an individual element is referred to by appending a subscript list to the array name.

A subscript list is a number of subscripts separated by commas and enclosed in parentheses. In standard FORTRAN, a subscript must be an expression of one of the following forms:

c * V + k	
c * V - k	
c * V	where c and k are INTEGER constants and V is an INTEGER variable
V + k	
V - k	
V	
k	

In 1900 FORTRAN, a subscript can be any expression of Type INTEGER (see 4.1.2). In each case, the expression is evaluated, the result defines a particular subscript, and the resulting subscript list defines a particular element of the array. The number of subscripts in a subscript list must be the same as the number of dimensions of the array.

Each subscript must be greater than zero but less than the maximum size of that dimension as specified in the array declaration (see 3.4).

Slightly different rules apply to the array elements in EQUIVALENCE statements (see 3.6).

EXAMPLES

TABLE(1)	The first element of the one-dimensional array TABLE
CUBIC3 (1,1,1)	The first element of the three-dimensional array CUBIC3
A(8*J+2 , L-1) B(9*I , K+7)	If I,J,K,L have values 1,2,3 and 5 then these references are to the elements A(18,4) and B(9,10) of the two-dimensional arrays A and B.

Chapter 3

STORAGE

This chapter considers the non-executable statements concerned with storage allocation and the setting of initial values of variables.

TYPE STATEMENTS 3.1

All items of data must be identified as a particular type in order that the compiler can arrange to process them correctly and reserve any storage required.

The type of a constant is indicated by the way in which it is written (see 2.1), but the type of any other item of data is determined either by a Type statement or by the initial letter of the name. The five Type statements take the forms

```
INTEGER list
REAL list
DOUBLE PRECISION list
COMPLEX list
LOGICAL list
```

where list is a sequence of names, successive names being separated by a comma.

A Type statement indicates to the compiler the type of all the items in its list. Items may be variables, arrays or functions. Any such item not declared explicitly in a Type statement will be interpreted according to the following rule:

> If the name begins with any of the letters I,J,K,L,M,N, the compiler will assume the item to be type INTEGER.
>
> If the name begins with any other letter it will be assumed to be type REAL.

The name of any variable, array or Statement function whose type is not as indicated by its first letter must appear in a Type statement in the segment in which it is used. Similarly, the name of any function whose type is not as indicated by its first letter must appear in a Type statement in each segment in which it is referenced (and also in its FUNCTION statement - see 7.3.1.

However, it is necessary to specify the types of the standard functions listed in Appendix 1. No type is associated with the name of a subroutine, Master segment or common block.

Type statements apply only to the segment in which they appear. No quantity may appear more than once in Type statements of the same segment.

EXAMPLES

```
REAL ITEM, JOULES, ANSWER
DOUBLE PRECISION PI, VAR1, VAR2, MEAN
```

Note: In the first example above, the entry ANSWER is not strictly necessary. It would have been automatically treated as type REAL by virtue of its initial letter A.

DATA STORAGE REQUIREMENTS 3.2

Different amounts of storage are required for different types of variables or array elements.

Type	Number of Storage Units
INTEGER	1
REAL	1
DOUBLE PRECISION	2
COMPLEX	2
LOGICAL	1

The method of recording values in these storage units will normally be of interest only if parts of a program are to be written in another language, e.g. PLAN. In 1900 FORTRAN, a storage unit is two words.

DIMENSION STATEMENTS 3.3

DIMENSION statements declare names to be array names and provide the compiler with the information necessary to allocate storage for arrays by defining the maximum size and shape of each array.

A DIMENSION statement normally takes the form

```
DIMENSION A(i), B(j), C(k),...X(y)
```

where A,B,C.... are array names and i,j,k.... are lists indicating the number of dimensions and the maximum size of each dimension.

For example, the statement

```
DIMENSION ARR (5, 5, 10)
```

specifies a three-dimensional array, ARR, with 250 elements. The quantities in parentheses show the maximum size of each dimension. They are normally unsigned INTEGER constants but in special cases can be INTEGER variables (see 7.4.3).

DECLARING ARRAYS 3.4

The dimensions of an array may be declared in a DIMENSION statement as above, or in a COMMON statement (see 3.5), or in a Type statement (see 3.1). Thus, the 10 x 5 INTEGER array A could be dimensioned in one of three ways:

```
      INTEGER A
      DIMENSION A (10,5)
or
      INTEGER A (10,5)
or    (if the array is common)
      INTEGER A
      COMMON A (10,5)
```

which would all have the effect of reserving storage for the 50 INTEGER elements of array A.

An array may be dimensioned only once. Thus, if an array is dimensioned in a DIMENSION statement, then no dimension information for the array may appear in Type statements of the same segment.

In standard FORTRAN, an array may have 1, 2 or 3 dimensions; in 1900 FORTRAN, up to 32 dimensions may be specified.

Note: For details on the declaration of an array that is a dummy argument of a function or subroutine, see 7.4.2.

Array Storage 3.4.1

Elements of an array are stored in column order in consecutive storage locations so that the left-hand subscript varies most rapidly and successive subscripts vary less rapidly. For example, a 3 x 3 x 3 array A would be stored in the order:

A(1,1,1), A(2,1,1), A(3,1,1), A(1,2,1), A(2,2,1), A(3,2,1),
A(1,3,1), A(2,3,1), A(3,3,1), ... A(2,3,3), A(3,3,3).

The order in which elements of an array are stored may become important in COMMON, EQUIVALENCE, DATA, READ and WRITE statements. Apart from these cases, the programmer need not be concerned with the relative positions of array elements.

A COMMON statement enables the programmer to specify that certain areas of store are available to more than one segment of the program. In this way, the values obtained in one segment may be used in another segment. The COMMON statement take takes the form

$$\text{COMMON } /X_1/a_1/X_2/a_2/X_3/a_3...../X_n/A_n$$

where each X is a common block name and each 'a' is a list of variables and array names, possibly with dimension information (see 3.4). This list must not include dummy arguments of functions or subroutines.

A block name, which must be formulated according to the rules given in 1.2, bears no relationship to any variable, array or statement function having the same name and is therefore an exception to the rule that a name must have only one meaning within a segment. However, a common block may not have the same name as a segment or a standard function.

The quantities between one block name, say XYZ, and the text are said to be in common block XYZ. If two oblique strokes appear with no block name between them, the entities that follow are said to be in the blank common block. If the blank column block is the first one referred to in the common statement, the first oblique strokes may be omitted, e.g.

```
COMMON X, ITEM, NAME /JOHN/A,B,C.....
```

has the same effect as

```
COMMON / / X, ITEM, NAME /JOHN/A,B,C....
```

When the same block name appears in two or more segments of the program, the compiler will allocate corresponding items in the lists associated with the name to the same storage in the order in which they appear. When an array name is given, the order of elements is as given in 3.4.1. The correspondence is by storage unit, where each variable or array element occupies one or two storage units depending on its type (see 3.2). For example, if in one segment the programmer wrote

```
COMMON/EQN/PI,LONG,HIGH
```

and in another segment he wrote

```
COMMON/EQN/PI,LEN,HEIGHT
```

then, assuming all the variables occupy one storage unit, LONG and LEN would be assigned the same storage, as would HIGH and HEIGHT, and in each segment PI would refer to the same location. If PI had not appeared in a COMMON statement, then the compiler would have treated PI in the first segment and PI in the second segment as different items and would have reserved two separate areas of store.

Since common areas are intended for communication of values between segments, corresponding entries in lists will normally be of the same type. Although it is possible to assign common storage to items of different types, care must be taken when doing this. If a storage unit is assigned a value of one type in one segment, it is an error to attempt to refer to this value in a second segment which specifies the unit as another type. For example, if in one segment the programmer wrote

```
COMMON/ALPHA/INT
```

and in another segment he wrote

```
COMMON/ALPHA/RL
```

then the REAL variable RL and the INTEGER variable INT would share the same storage. However, if INT is assigned an INTEGER value in the first segment, it is incorrect to refer to RL in the second segment until it has been assigned a REAL value.

A given block name may appear more than once in the COMMON statements of a segment. The compiler will allocate all the entities associated with this block name to one common block in the order in which they appear. For example, the statement

```
COMMON/SAME/A,B,C/CONST/I,J/SAME/D,E
```

would have the same effect as

```
COMMON/SAME/A,B,C,D,E/CONST/I,J
```

In standard FORTRAN, a named common block must have the same declared size in all segments in which it is used; in 1900 FORTRAN this is not essential as the actual size of any block is taken to be the maximum of its declared sizes.

EXAMPLES

```
COMMON/SET1/EL1,EL2,EL3,ARRAY
COMMON/SET3/AR1,AR2,AR3(5,10)/SET4/LAST
```

EQUIVALENCE STATEMENTS 3.6

An EQUIVALENCE statement enables the programmer to save storage by arranging for two or more entities in a segment to share the same storage units. It takes the form

$$\text{EQUIVALENCE } (k_1),\ (k_2),\ (k_3),\ldots\ldots\ldots\ldots\ldots(k_n)$$

where each k is a list of variables and array elements to be assigned the same storage by the compiler.

The list must not include dummy arguments of functions or subroutines and no variable should appear in more than one list. However, it is possible though not recommended for an array element to appear in more than one list subject to certain restrictions given below. The subscripts of array elements must be unsigned INTEGER constants; array names without subscripts are not permitted.

EQUIVALENCE lists must be compatible. In particular, the programmer should ensure that his EQUIVALENCE lists do not result in an attempt to assign a variable or array element to more than one area of store. Thus, while it is possible for an array element to appear in two lists in statements such as

```
EQUIVALENCE (X(1),A).....(X(1),B)
```

a statement such as

```
EQUIVALENCE (X(1),ARR(1))......(X(1),ARR(3))
```

is obviously incorrect since it is an attempt to equivalence X(1) to both ARR(1) and ARR(3), which will necessarily occur in different areas of store.

To prevent any possibility of incompatible equivalence lists, it is recommended that no item, not even an array element, should appear in two equivalence lists.

Although it is possible to equivalence items of different types, care must be taken when doing this. For example, if two variables RL of type REAL and INT of type INTEGER were equivalenced in a segment and RL had been assigned a value of 38.76, then INT must be assigned an INTEGER value before it can be used in an expression. If a two-storage unit variable (see 3.2) is equivalenced to a one-storage unit variable, the latter will share space with the first storage unit of the former.

Since all the items in a particular EQUIVALENCE list share the same storage, a change in the value of one item may change the value of another. The programmer must allow for this but must not try to exploit it to give one item the same value as another item, particularly if they are of different types.

The EQUIVALENCE statement used in conjunction with a COMMON statement will not change in order of a common block. However, if any array element is declared equivalent to a variable in a common block, then the whole array will be in common, the common block being lengthened if necessary. The block can only be extended beyond the last entry for that block; it cannot be extended before the first entry. For example, if a five-element array ARR is declared equivalent to an entry in a COMMON statement as follows:

```
COMMON/JSG/A,B,C
EQUIVALENCE (ARR(1),A)
```

then the common block will consist of

ARR(1) equivalent to A
ARR(2) equivalent to B
ARR(3) equivalent to C
ARR(4)
ARR(5)

The same effect could have been obtained by equivalencing B and ARR(2), or C and ARR(3). It is also permissible to equivalence ARR(1) to B or C, and ARR(2) to C, the latter resulting in a common block consisting of

A
ARR(1) equivalent to B
ARR(2) equivalent to C
ARR(3)
ARR(4)
ARR(5)

However , a statement

```
EQUIVALENCE (ARR(4),C)
```

is invalid because this attempts to extend the common block into the storage before A.

In the example quoted above, the array elements and A,B,C are all assumed to have the same number of storage units (see 3.2). The effect when this is not the case can best be illustrated by an example. Suppose ARR and A above were COMPLEX and B and C were REAL, i.e. ARR and A have two storage units each, and B and C have one. The common block would then consist of

ARR(1) equivalent to A
ARR(2) first unit equivalent to B
second unit equivalent to C
ARR(3)
ARR(4)
ARR(5)

It is possible to use a single subscript when equivalencing an element of a multi-dimensional array. For example, the second element A(2,1,1) of a three-dimensional array A may be equivalenced to the variable B either by the statement

```
EQUIVALENCE (B,A(2,1,1))
```

or by

```
EQUIVALENCE (B,A(2))
```

Similarly, the 26th element could be equivalenced to Z in the statement:

```
EQUIVALENCE (Z,A(26))
```

The order in which elements are stored is given in 3.4.1. This is the only case when a subscript list need not correspond to the declared dimensionality.

DATA STATEMENTS 3.7

A DATA statement is a non-executable statement used to define initial values for variables and array elements. It takes the form

$$\text{DATA } R_1/d_1/,R_2/d_2/,\ldots.R_n/d_n/$$

where each R is a list of variables. array elements and array names, and each d is an associated list of initial values for these variables, array elements and array names. An array name is equivalent to a list of all elements of the array written in the order given in Part 1, Section 3.4.1.

An R list may not include dummy arguments of functions or subroutines. The subscripts of array elements must be unsigned INTEGER constants. Successive entries must be separated by commas.

A d list may include signed or unsigned numerical constants or TEXT constants. A COMPLEX constant may include a sign only within its parentheses. Any item in a d list may be preceded by 'j*' where j is an unsigned integer; such an entry is equivalent to repeating the item j times.

For every item in an R list, there must be an entry in the associated d list which specifies an initial value for the item. At the time a program is first entered, the item is assigned this value. For example, the statement

```
DATA A,B(1),C/2*1.0,3.5/
```

gives an initial value of 1.0 to both A and B(1), and an initial value of 3.5 to C.

Further examples are:

```
DATA ANDY/38.3/,TERRY,BOB/2*-3.9/

DATA ITEM(1),ITEM(2),/2*326/

DATA HEAD/8HANSWER =/
```

The effect of the final example, where a TEXT constant is assigned to a variable, is explained in 3.8.

Note: A variable or array element in a common block may be given an initial value only in a DATA statement of a BLOCK DATA segment (see 7.6).

There are no TEXT variables as such, but a variable of any type may be used to hold TEXT data. TEXT values may be assigned to variables by DATA statements (see 3.7) : by specifying a TEXT constant as the actual argument of a subroutine call (see 7.4) : or by an A format specification used in conjunction with a READ statement (see Part 2).

TEXT values are held as strings of six-bit characters in the 1900 character set. One storage unit may hold eight such characters; consequently, up to eight characters may be held in an INTEGER, REAL or LOGICAL variable and up to sixteen characters may be held in a COMPLEX or DOUBLE PRECISION variable.

It is recommended that all TEXT constants to be assigned to variables contain exactly eight characters (for REAL, INTEGER and LOGICAL variables) or sixteen characters (for DOUBLE PRECISION or COMPLEX variables). If this is not done, the effect is as follows:

> If a TEXT constant contains fewer characters than can be held in the variable, extra space characters are added to the right until the number of characters is a multiple of eight. Any remaining character positions of the variable are undefined, i.e. nothing should be assumed about their values. It is an error to assign more than eight characters to a one-storage unit variable or more than sixteen characters to a two-storage unit variable.

Array elements are treated in the same way as variables, except that any characters in excess of eight (or sixteen) will be held in succeeding elements of the array. For example,

```
DATA A(1)/14HEND OF RESULTS/
```

will result in 'END OF R' being held in A(1) and 'ESULTS', followed by two space characters, being held in A(2).

It is an error to assign more characters to an array than can be held in that array.

This chapter gives the rules for the formation and evaluation of arithmetic and logical expressions. Arithmetic expressions have numerical values. Logical expressions have logical values.

ARITHMETIC EXPRESSIONS 4.1

An arithmetic expression is a sequence of arithmetic operators and elements of type INTEGER, REAL, COMPLEX or DOUBLE PRECISION. Its value is always numerical. The simplest form consists of a single element, which may be a constant, a variable, an array element or a function reference. Function references are dealt with in 4.3.

EXAMPLES

123.0 MASS A(I,3) SIN(X)

More complicated expressions may be formed by combining elements with the arithmetic operators

+	addition or positive value
-	subtraction or negative value
*	multiplication
/	division
**	exponentiation, i.e. raising to a power

Two arithmetic operators must not be adjacent. The operators + and - must be followed by an element. The other operators must be both preceded and followed by an element. Multiplication must always be indicated by *; thus A*B cannot be re-written as AB or A.B. Constructs such as A**B**C are ambiguous and must be written as (A**B)**C for $(A^B)^C$ or A**(B**C) for A^{B^C}.

Further impermissible constructs are given in 4.1.2.

EXAMPLES

```
123.0*MASS
-A+B*COUNT-3.197
ARRAY(2,10)-COS(Z)/2*PI
CONJ-(3.0,2.0)+CONJB*I  (a COMPLEX arithmetic expression)
L**I-3.0/CHAIN          (meaning: L^I - 3.0/CHAIN
```

An arithmetic expression may also contain elements consisting of other arithmetic expressions enclosed in parentheses. These may be fullly general expressions, so they may themselves contain parenthesized expressions, and so on.

EXAMPLES

```
-123.0*MASS/(-A+B*COUNT-3.197)
ARRAY(1,10)-LENGTH**(I-3*(PARAM-1/(WIDTH*2))-1)
```

Order of Evaluation 4.1.1

Evaluation of an arithmetic expression starts at the innermost set of parentheses and works outwards. Within one set of parentheses, or in the expression as a whole, the theoretical order of evaluation is as follows:

Function references and parenthesized expressions

Exponentiations

Multiplications and divisions

Additions and subtractions

Within each of these classes the order is from left to right.

EXAMPLE

The expression

```
A+B*C/D*(P-1)-3.0**(Q+R)+2.0/X**2
```

is effectively evaluated as follows:

(P-1)	gives E_1
(Q+R)	gives E_2
3.0**E_2	gives E_3
X**2	gives E_4
B*C	gives E_5

E_5/D gives E_6

E_6*E_1 gives E_7

$2.0/E_4$ gives E_8

and $A+E_7-E_3+E_8$ gives the final result.

The actual order of evaluation may differ from the above but will be mathematically equivalent. For example, A/B*C may be evaluated as though it were either (A/B)*C or (A*C)/B. Parentheses can be used to specify a particular order of evaluation if required, and this order will be followed. A sequence of multiplication and division operations on INTEGER quantities will always proceed from left to right.

Type of Expressions 4.1.2

An arithmetic expression and its value may be of type INTEGER, REAL, COMPLEX or DOUBLE PRECISION, depending on the types of its elements. The expression is evaluated step by step as defined in 4.1.1. Each step yields a value to be used in the next step. The type of each value is determined as shown in Tables 4.1 and 4.2. The value generated in the final step is the value of the expression.

	OPERATOR	TYPE OF ELEMENT			
	+, -, *, /	INTEGER	REAL	DOUBLE PRECISION	COMPLEX
TYPE OF ELEMENT	INTEGER	INTEGER (see note 1)	N(REAL)	N(DOUBLE PRECISION)	N(COMPLEX)
	REAL	N(REAL)	REAL	DOUBLE PRECISION	COMPLEX
	DOUBLE PRECISION	N(DOUBLE PRECISION)	DOUBLE PRECISION	DOUBLE PRECISION	N
	COMPLEX	N(COMPLEX)	COMPLEX	N	COMPLEX

Table 4.1

OPERATOR	EXPONENT			
**	INTEGER	REAL (see note 3)	DOUBLE PRECISION (see note 3)	COMPLEX
INTEGER	INTEGER (see note 1)	N(REAL)	N(DOUBLE PRECISION)	N
REAL	REAL	REAL	DOUBLE PRECISION	N
DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	N
COMPLEX	COMPLEX	N	N	N

BASE (see note 2)

Table 4.2

N Prohibited combination

N( ) Prohibited combination in standard FORTRAN but a permitted combination in 1900 FORTRAN with a resulting value as shown in parentheses.

Notes: 1. A result of type INTEGER is the nearest integer whose absolute value does not exceed the absolute value of the mathematically correct result. Thus, 11/4 and 3**(-1) give the values 2 and 0 respectively.

2. A zero base must not be raised to a zero exponent.

3. A negative element must not be raised to a REAL or DOUBLE PRECISION exponent.

One effect of the rules given in this section is that the expression

X-Y + DP/Z + DP

(where X, Y and Z are REAL and DP is DOUBLE PRECISION) will have a DOUBLE PRECISION value but some of the individual operations may not take place in DOUBLE PRECISION fashion, depending on the order of evaluation (see 4.1.1). Parentheses may be used to control the order of evaluation. Similar considerations apply to the mixed INTEGER and REAL or DOUBLE PRECISION expressions allowed in 1900 FORTRAN; that is some of the operations may be carried out in INTEGER fashion unless parentheses indicate otherwise.

A logical expression is a sequence of logical operators and elements of type LOGICAL. Its value is always either .TRUE. or .FALSE. The simplest form consists of a single LOGICAL element which may be a constant, a variable, an array element or a function reference. Function references are dealt with in 4.3.

EXAMPLE

.TRUE. LVAR STATUS(I,3) OK(B)

Another simple kind of logical expression is the relational expression. This has the form

$$e_1 \quad r \quad e_2$$

where e_1 and e_2 are arithmetic expressions and r is one of the following relational operators:

Operator	Representing
.LT.	less than
.LE.	less than or equal to
.EQ.	equal to
.NE.	not equal to
.GT.	greater than
.GE.	greater than or equal to

The full stops are essential.

A relational expression has the value .TRUE. or .FALSE. as the relation is true or false, respectively. Allowable combinations of e_1 and e_2 are given in Table 4.3.

	INTEGER	REAL	DOUBLE PRECISION	COMPLEX
INTEGER	Y	Y*	Y*	N
REAL	Y*	Y	Y	N
DOUBLE PRECISION	Y*	Y	Y	N
COMPLEX	N	N	N	N

Table 4.3

N Prohibited combination

Y Allowable combination

Y* Prohibited combination in standard FORTRAN but a permitted combination in 1900 FORTRAN.

EXAMPLES

```
X .LT. 0.0
B**2  .GT.  4*A*C
SIN(X)+COS(Y)-1.0 .GE. 1.0/(2*PI)
```

More complicated logical expressions may be formed by combining logical elements and relational expressions with the logical operators

.NOT.

.AND.

.OR.

The effect of the logical operators are defined as follows (where a and b are logical elements).

.NOT. a — This has the value .TRUE. if a is .FALSE.; it has the value .FALSE. if a is .TRUE.

a .AND. b — This has the value .TRUE. if a and b are both .TRUE.; it has the value .FALSE. if either a or b is .FALSE.

a .OR. b — This has the value .TRUE. if either a or b is .TRUE.; it has the value .FALSE. if both a and b are .FALSE.

The full stops are essential.

Two logical operators must not be adjacent unless the first is .AND. or .OR. and the second is .NOT. The operator .NOT. must be followed by, but must not be preceded by, an element. The operators .AND. and .OR. must be preceded by an element and followed by an element or .NOT.

EXAMPLES

```
.NOT. OVER21
STATUS .AND. CLASS
JOB .EQ. 3 .AND. AGE .LT. 18
CHECK .OR. SIZE .LT. 31.5 .AND. .NOT. LOGAR(I,5)
```

A logical expression may also contain elements consisting of other logical expressions enclosed in parentheses. These may be fully general expressions, so they may themselves contain parenthesized expressions and so on.

EXAMPLES

```
A .LT. B+1.0 .AND. (X .LT. 0.01 .OR. COUNT .EQ. 0)
(AB .OR. AC) .AND. .NOT. (CHECK .OR. MISS)
```

Order of Evaluation 4.2.1

Evaluation of a logical expression starts at the innermost sets of parentheses and works outwards. Within one set of parentheses, or in the expression as a whole, the order of evaluation is as follows:

Relational expression, function references and parenthesized logical expressions

.NOT. operations

.AND. operations

.OR. operations

Within each of these classes the order is from left to right.

EXAMPLE

The expression

```
A .OR. B .OR. C .AND. (P .OR. Q) .AND. (I .LT. 1 .OR. J .EQ. 0)
```

is effectively evaluated as follows:

	P .OR. Q	gives E_1
	I .LT. 1	gives E_2
	J .EQ. 0	gives E_3
	E_2 .OR. E_3	gives E_4
	C .AND. E_1	gives E_5
	E_5 .AND. E_4	gives E_6
	A .OR. B	gives E_7
and	E_7 .OR. E_6	gives the final result

A function reference may appear in an expression as described in 4.1 and 4.2. It has the form

$$f(a_1,a_2,....a_n)$$

where f is the function name and the 'a's are actual arguments; f may be the name of a standard function, a FUNCTION segment or a Statement function.

Standard Functions 4.3.1

A variety of standard functions such as sine, cosine and logarithm are available in FORTRAN. A list of these functions, their names and meanings, is given in Appendix 1. The actual arguments in a reference to a standard function must agree in order, number and type with the arguments shown in the table.

They can be any expressions of the appropriate type and therefore may themselves include function references.

EXAMPLES

```
X=SIN(ANGLEA)
F=SQRT(B)
FORM=A**2+2.0*COS(BETA+PI/2.0)
AG32=A+SIN(LOG(A+B+C))**3+SQRT(SQRT(Z+SQRT(Y)))
```

FUNTION Segments 4.3.2

The programmer may define his own functions by writing FUNCTION segments (see 7.3). When a reference to such a function is encountered in the evaluation of an expression, there is a temporary transfer of control to the FUNCTION segment. The function is calculated and a RETURN statement (6.6) returns control to the evaluation of the expression, where the value of the function is made available.

Actual arguments must agree in order, number, kind and type with the dummy arguments used in the FUNCTION segment (see 7.4).

Statements Functions 4.3.3

If a function can be expressed as one statement, it may be written as a Statement function rather than a FUNCTION segment. A Statement function is defined by a statement of the form.

$$f(a_1,a_2,....a_n)=e$$

where f is the name chosen for the function, the 'a's are dummy arguments and e is an expression; f has the form described in 1.2 and must be distinct from any other name appearing in the segment (except for common block names).

Each 'a' must be a dummy variable name. Its type is decided as for an ordinary variable, either by its first letter or by a Type statement (see 3.1). A dummy variable of this sort is only for use within the Statement function. The same name may be used elsewhere in the segment for a variable of the same type, but there is no connection between the two.

e may be any expression conforming to the rules given in 4.2 or 4.3, except that array elements must not appear. The dummy arguments will normally appear in the expression. Any Statement functions referred to must have been defined earlier in the segment.

Statement function definitions must appear before any executable statements of a segment. (Statement order is covered more fully in 7.5.)

The form of a Statement function definition is very similar to that of an assignment statement, except for the list of arguments on the left-hand side. The relation between the type of the function and the type of the expression on the right-hand side must conform to the same rules as for assignment statements. These rules are given in Chapter 5.

A Statement function reference may occur only in expressions within the same segment as the Statement function definition. At each reference, the actual arguments in the reference are substituted for the dummy arguments, the function is evaluated, and its value is transmitted back to the referencing statement. Actual arguments must agree in order, number and type, and may be any expressions of the appropriate type. These expressions will be evaluated once and these values will be substituted each time the corresponding dummy arguments occur in the Statement function.

EXAMPLE

Suppose it is required to calculate the positive root of several quadratic equations. A Statement function could be defined as:

```
ROOT(A,B,C)=(SQRT(B*B-4.0*A*C)-B)/(2.0*A)
```

Then the root of an equation such as

$$99.7x^2 + (A-3*B)x + Z = 0$$

could be included in any expression, for example:

```
ANS=SIN(F)+ROOT(99.7,A-3*B,Z)
```

The dummy arguments A and B bear no relation to the A and B of the actual argument A-3*B, which will be evaluated and its value substituted wherever the dummy argument B occurs.

An assignment statement is an executable statement that gives a new value to a variable or array element. Any previous value is lost. There are two types of assignment statement : arithmetic and logical.

ARITHMETIC ASSIGNMENT 5.1

An arithmetic assignment statement assigns a numerical value to a variable or array element of appropriate type. It has the form

V=e

where V is a variable or array element of type INTEGER, REAL, COMPLEX or DOUBLE PRECISION and e is an arithmetic expression (see 4.1).

Execution of the statement causes e to be evaluated and its value assigned to V, replacing any previous value of V. In general, V will retain this new value until another value is assigned by an assignment statement or by a READ statement.

V and e need not be of the same type. If they are of different types, e is evaluated according to the normal rules for its type and the resulting value is transmitted to V according to Table 5.1. Certain prohibited combinations of V and e are also shown.

EXAMPLES

```
VARN=38.7654
COUNT=COUNTA          } A constant, variable or array
NEXT=ITEM(3,1)        } element is a trivial example
                      } of an expression
DIAG(LIST)=DIAG(LEN)+DIAG(LEN+10)
AREA=2.*PI*(3.*X*SIN(Y)-Z)
M=(2*M-1)/Q-M
```

TYPE OF V	TYPE OF e	RULE (see Notes below)
INTEGER	INTEGER	Assign
INTEGER	REAL	Fix and Assign
INTEGER	DOUBLE PRECISION	Fix and Assign
INTEGER	COMPLEX	Prohibited
REAL	INTEGER	Float and Assign
REAL	REAL	Assign
REAL	DOUBLE PRECISION	REAL Assign
REAL	COMPLEX	Prohibited
DOUBLE PRECISION	INTEGER	Float to DOUBLE PRECISION and Assign
DOUBLE PRECISION	REAL	Float to DOUBLE PRECISION and Assign
DOUBLE PRECISION	DOUBLE PRECISION	Assign
DOUBLE PRECISION	COMPLEX	Prohibited
COMPLEX	INTEGER	Prohibited
COMPLEX	REAL	Prohibited
COMPLEX	DOUBLE PRECISION	Prohibited
COMPLEX	COMPLEX	Assign

Table 5.1

Notes: 1. Assign means transmit the resulting value, without change, to V.

2. REAL Assign means transmit to V the most significant part of the resulting value.

3. Fix means truncate towards zero any fractional part of the result and convert the remaining value to INTEGER form.

4. Float means transform the value to REAL form.

5. Float to DOUBLE PRECISION means transform the value to DOUBLE PRECISION form.

LOGICAL ASSIGNMENT 5.2

A logical assignment statement assigns one of the values .TRUE. or .FALSE. to a logical variable or array element. It has the form

V=e

where V is a variable or array element of type LOGICAL and e is a logical expression (see 4.2).

Execution of the statement causes e to be evaluated and its value assigned to V, replacing any previous value of V. In general, V will retain this new value until another value is assigned by an assignment statement or by a READ statement.

EXAMPLES

```
COURSE= .TRUE.
QUAD=B**2 .GT. 4*A*C
STATUS=STUDNT .AND. AGE .LT. 21
```

MULTIPLE ASSIGNMENT 5.3

Multiple assignment statements are allowed in 1900 FORTRAN but not in standard FORTRAN. A multiple assignment statement gives the same value to more than one variable or array element. It has the form

$$V_1,V_2,....V_n=e$$

where each V is a variable or array element and e is an expression.

Execution of the statement causes e to be evaluated and its value to be assigned in turn to V_n, V_{n-1},....V_1. If e is an arithmetic expression, the assignment rules of Table 5.1 are applicable. If all the 'V's are not of the same type, the effect is unpredictable. If e is a logical expression, all the 'V's must be of type LOGICAL.

EXAMPLES

```
SUM,SUMSQ,ITEM(N)=0
PARAM,START=(N+Q-1)/L
VOL(1),VOL(2),VOL(3)=986.1/LIM
```

CONTROL STATEMENTS

A FORTRAN program is controlled so that execution begins with the first executable statement of the MASTER segment; thereafter, statements are executed in the order in which they are written until a transfer of control is specified. The methods of specifying such a change in the order of execution are described in this chapter. When control is said to be transferred from one statement to another, the latter statement and the statements following it are executed sequentially until another transfer of control is initiated.

GO TO STATEMENTS 6.1

A GO TO statement is an executable statement that transfers control to another statement in the same segment. There are three types of GO TO statement: Unconditional, Computed and Assigned.

Unconditional GO TO 6.1.1

Each time an unconditional GO TO statement is executed, control is transferred to the same specified statement. It has the form

```
GO TO k
```

where k is a label applied to a statement in the same segment as the GO TO. Execution of the GO TO statement transfers control to the statement labelled k.

EXAMPLES

```
GO TO 3
GO TO 99999
```

Computed GO TO 6.1.2

A Computed GO TO statement transfers control to one of a list of statements, depending upon a previously computed value of a variable. It has the form

$$\text{GO TO } (k_1,k_2,\ldots.k_n),i$$

where i is an integer variable and each k is a label applied to a executable statement in the same segment as the GO TO. A particular execution of the statement is equivalent to

$$GO\ TO\ k_j$$

where j is the value of the variable i at the time of execution. The effect of the statement is unpredictable if the value of i is outside the range 1 to n.

EXAMPLES

```
    INDEX=4
    .......
    GO TO (97,3,307,463,5),INDEX        GO TO the statement labelled 463,
    .......                             i.e. the fourth in the list.
    GO TO (30,101,30,101,30,101),JELL   GO TO the statement labelled 30 if
                                        the value of JELL is 1, 3 or 5, or
                                        to the statement labelled 101 if
                                        the value of JELL is 2, 4 or 6.
```

In 1900 FORTRAN, oneor more of the 'k's in the list of a computed GO TO may be 0 instead of a statement label. If 'i' specifies one of these zero values, then control is transferred to the statement following the GO TO statement.

EXAMPLE

The following sequences are equivalent in 1900 FORTRAN

```
    GO TO (31,71,99,99,99,197,99),MOOT    GO TO (31,71,0,0,0,197,0) MOOT
 99 .................(next statement)     .............(next statement)
```

Assigned GO TO 6.1.3

An Assigned GO TO statement transfers control to one of a list of statements, depending upon a value assigned to a variable by an ASSIGN statement (6.1.4).

It has the form

$$GO\ TO\ i,(k_1,k_2,....k_n)$$

where 'i' is an integer variable and each k is a label applied to an executable statement in the same segment as the GO TO.

A particular execution of the statement is equivalent to

GO TO k_j

where k_j is the label that was last assigned to i by an ASSIGN statement (see 6.1.4). If at the time of execution of the Assigned GO TO 'i' has a value that was not assigned by an ASSIGN statement in the same segment, then the effect is unpredictable.

EXAMPLE

```
    ASSIGN 57 TO MEANS
    ........
    GO TO MEANS, (92,3,9999,57,3)      GO TO the statement labelled 57.
```

In 1900 FORTRAN the list of statement labels may be omitted in an Assigned GO TO statement.

EXAMPLE

```
    ASSIGN 57 TO MEANS
    ........
    GO TO MEANS                        GO TO the statement labelled 57
```

ASSIGN Statements 6.1.4

An ASSIGN statement is a special type of assignment statement that assigns a statement label to an integer variable. It has the form

ASSIGN k TO i

where k is a label applied to an executable statement in the same segment as the ASSIGN statement and 'i' is an INTEGER variable.

The statement assigns the label k to the variable 'i'. It does <u>not</u> simply assign the integer value k to i; e.g. ASSIGN 3 TO INDEX is not equivalent to INDEX = 3. This form of assignment is intended for use in conjunction with an Assigned GO TO statement. An index given a value by an ASSIGN statement must not be used, except in an Assigned GO TO, until it has been given a conventional integer value; e.g. the following sequence is invalid and would leave MESS with no predictable value:

```
ASSIGN 31 TO LINK
MESS=3*LINK+1
```

An example of the correct use of ASSIGN is given in 6.1.3.

An IF statement is an executable statement that tests a particular condition and performs different actions depending on the result of the test. There are two types of IF statement: Logical and Arithmetic.

Logical IF 6.2.1

A Logical IF statement will test the truth or falsity of a logical expression and, if it is true, execute a particular statement written within the IF statement. It has the form

IF (l)s

where l is a logical expression (as defined in 4.2) and s is any executable statement except a DO statement (see 6.3) or another Logical IF statement.

The effect of the statement is determined by the value of l as follows:

(a) If l has the value .FALSE. the statement 's' is ignored and control is transferred to the next statement.

(b) If l has the value .TRUE. the statement 's' is executed. If s is an arithmetic IF (see 6.2.2) or a GO TO (see 6.1), control is transferred as specified for such statements. If s is a CALL (see 6.5), control reverts to the next statement upon return from the subroutine. If s is any other type of statement, control passes to the next statement when s has been executed.

EXAMPLES

```
IF (B*B .LT. 4.0*A*C) GO TO 100
```

GO TO the statement labelled 100 if $B^2<4AC$. Otherwise, continue with the next statement.

```
IF (SUM+TERM .GE. 9E7 .OR. TERM .LT. 1E-2) CALL CHECK
IF (A .LT. 0.0)A= -100.0*A
IF (READY) WRITE (1,32)A,B,MASS1,VOL1A
```

Arithmetic IF 6.2.2

An Arithmetic IF statement transfers control to one of three statements depending on the value of an arithmetic expression. It has the form

IF (e)k_1,k_2,k_3

where e is an arithmetic expression of type INTEGER, REAL or DOUBLE PRECISION (not COMPLEX) and k_1, k_2 and k_3 are labels applied to executable statements in the same segment as the IF.

The statement transfers control to the statement labelled k_1, k_2 or k_3 depending on whether e is less than, equal to, or greater than zero respectively.

EXAMPLES

```
IF (B*B-4.0*A*C)100,101,102
```

Transfer to:
100 if $B^2 < 4AC$
101 if $B^2 = 4AC$
102 if $B^2 > 4AC$

```
IF (SIN(X)+COS(PI-Z)-0.57)31,58,58
IF ((PREC-ERR)/5.3D-4)71,77,71
```

In 1900 FORTRAN, one or more of the 'k's in an Arithmetic IF statement may be 0 instead of a statement label. A zero will refer control to the next statement.

EXAMPLE

The following sequences are equivalent in 1900 FORTRAN:

```
    IF (MASS-CRITIC)99,31,99
 99 ........(next statement)
```

```
    IF (MASS-CRITIC)0,31,0
    ......(next statement)
```

DO STATEMENTS 6.3

A DO statement is executable and causes a group of statements to be executed repeatedly. It has one of the forms

DO n i=m_1,m_2,m_3

or

DO n i=m_1,m_2

where n is the label of the terminal statement associated with the DO statement; i is the control variable; and m_1, m_2 and m_3 are the initial parameter, the terminal parameter and the incrementation parameter respectively.

The terminal statement must be in the same segment as the DO statement and must physically follow it. The control variable, i, must be an INTEGER variable. In standard FORTRAN, the parameters m_1, m_2, m_3, must be INTEGER constants or INTEGER constants or INTEGER variables; in 1900 FORTRAN they may be any expressions of type INTEGER. At the time of execution of a DO statement, all three parameters must be greater than zero.

The second form of the DO statement implies a value of 1 for the incrementation parameter.

The range of a DO statement is the set of statements beginning with the one following the DO statement and continuing up to and including the terminal statement.

The actions caused by a DO statement are as follows:

1. The control variable is assigned the value of the initial parameter, which must be less than or equal to the value of the terminal parameter.

2. The statements in the range of the DO are executed. (Thus they are always executed at least once.)

3. If control reaches the terminal statement, the control variable of the DO is incremented by the value of the incrementation parameter.

4. If the new value of the control variable is less than or equal to the value of the terminal parameter, the actions above are repeated from step 2. Otherwise, the DO is said to be satisfied and the statement following the terminal statement is executed.

The situation is slightly more complex for nested DOs (see 6.3.1).

The values of i, m_1, m_2, and m_3 may be referred to during the execution of the range of a DO but they must not be changed. The effect of doing so is unpredictable. When a DO becomes satisfied, the value of the control variable is unpredictable; it should not be assumed to be the same as m_3. The terminal statement of a DO must not be a GO TO, RETURN, STOP, PAUSE, DO or Arithmetic IF statement, nor may it be a Logical IF statement containing any of these forms.

EXAMPLE

```
      SUMSQ=0.0
      SUM=0.0
      DO 27 J=1,MAX,2
      SUM=SUM+PART(J+1)
   27 SUMSQ=SUMSQ+PART(J)*PART(J)
```

There are two statements in the range of this DO. The effect of the sequence is to set SUM equal to the sum of PART(2), PART(4), ...PART(MAX+1) and SUMSQ equal to the sum of the squares PART(1),PART(3), ...PART(MAX) (MAX is assumed to be odd).

Nested DO Statements 6.3.1

When the range of a DO itself contains one or more DO statements, the DOs are said to be nested. The range of an inner DO must be completely contained within the range of an outer DO. However, they may share the same terminal statement. In the latter case, the control variable of a particular Do is incremented and tested only if any inner DOs are satisfied.

Transfer of Control 6.3.2

Control transfers within the range of a DO are allowed as usual, except for one minor restriction: if a statement is the terminal statement of more than one DO statement, control can be transferred to that statement only from the range of the innermost DO with that terminal statement.

Control transfers out of the range of a DO are allowed. After such a transfer the control variable retains its current value.

A control transfer to within the range of a DO is permitted in standard FORTRAN only if the following rules are applied. (The simpler rule for 1900 FORTRAN is given later.)

1. The transfer must be from a statement completely outside the range of any DO statements to within the range of a DO that contains no other DO statements.

2. There must previously have been a transfer from this innermost range to a statement outside the range of a DO, and the values of the control variable and the three parameters of the DO must have remained unchanged.

3. During a temporary transfer of control from the range of a DO as described by 1 and 2 above, it is not permitted to execute another DO or nest of DOs in the same segment and temporarily transfer control out of that nest (except to a Procedure).

The rule for 1900 FORTRAN is much simpler. A control transfer to within the range of a a DO is permitted if there has previously been a transfer out of the range and if the values of the control variable and the three parameters of the DO have meanwhile remained unchanged.

CONTINUE STATEMENTS 6.4

A CONTINUE statement is a dummy statement and causes no action. It has the form

```
CONTINUE
```

It is most frequently used as the terminal statement of a DO range when a control transfer from within the range is intended to increment the control variable and either repeat the range or satisfy the DO.

EXAMPLE

```
      DO 3 I=1,100
      IF (A(I))4,3,3
    4 A(I)=A(I)+10.0
    3 CONTINUE
```

CALL STATEMENTS 6.5

A CALL statement transfers control to a SUBROUTINE segment and specifies actual arguments for the Subroutine to operate on. It has the general forms

$$\text{CALL } s(a_1, a_2, \ldots . a_n)$$

or

$$\text{CALL } s$$

where 's' is the name of a SUBROUTINE segment and a_1, a_2,....a_n are actual arguments to be substituted for the dummy arguments of the subroutine (see 7.4).

The second form is used when the subroutine has no arguments.

The statement transfers control to the first executable statement of 's'. Statements of 's' are executed normally, including any temporary transfers to other segments, until a RETURN statement is executed within 's'. Then control reverts to the statement following the CALL.

A subroutine must not CALL itself, not even indirectly. Thus, it is an error to execute two CALLS for the same subroutine without the intervening execution of a RETURN statement within the subroutine.

EXAMPLES

```
CALL JASON(VAR,INT,ARRAY,CHECK,A(1)+SIN(X))
CALL ARGON
```

RETURN STATEMENTS 6.6

A RETURN statement returns control from a SUBROUTINE or FUNCTION segment to the calling segment. It has the form

RETURN

It may appear only in a SUBROUTINE or FUNCTION segment. There may be any number of RETURN statements in such a segment but only one of them is executed on any single execution of the segment.

From a SUBROUTINE segment, control reverts to the statement following the CALL statement. From a FUNCTION segment, control reverts to the evaluation of the expression containing the function reference.

STOP STATEMENTS 6.7

A STOP statement is used to stop the running of a program. It has one of the forms

```
STOP
```

or

```
STOP n
```

where n is a string of between one and five octal digits.

When the statement is executed, no further statements of the program are executed and the program is deleted from the computer. It cannot be re-started. Different values of n may be used to distinguish different STOPS of a program.

EXAMPLES

```
STOP 77
STOP 12473
```

PAUSE STATEMENTS 6.8

A PAUSE statement can be used to halt the program temporarily. It has one of the forms

```
PAUSE
```

or

```
PAUSE n
```

where n is a string of between one and five octal digits. Usually, the statement halts the program so that an operator can take some action such as the loading of a card reader. The operator may restart the program from the PAUSE statement. Different values of n may be used to distinguish different PAUSE statements in a program.

EXAMPLES

```
PAUSE 33
PAUSE 77777
```

PROGRAM STRUCTURE

A FORTRAN program consists of one or more segments. Each segment is relatively self-contained and may be written and compiled independently of other segments. This chapter describes four kinds of FORTRAN segment: MASTER, SUBROUTINE, FUNCTION and BLOCK DATA. SUBROUTINE and FUNCTION segments are similar and are known collectively as Procedure segments.

MASTER SEGMENT 7.1

Every program must contain one, and only one, MASTER segment. In 1900 FORTRAN it must start with a non-executable statement of the form

MASTER m

where m is a name chosen to identify the segment. The name must have the form described in 1.2 and must be distinct from the following names:

(a) the name of any other segment in the program,

(b) the name of any standard function referred to in the program,

(c) the name of any common block used in the program,

(d) the name of any variable, array or Statement function used in the MASTER segment.

No type is associated with a MASTER segment name. The last line of a segment must be

END

This is effectively a non-executable statement with no continuation lines.

Since it is not executable, it must be preceded by a STOP statement or a control statement, e.g. GO TO.

A MASTER segment is the controlling segment of a program. In small programs it may be the only segment. Execution of a program starts at the first executable statement of the MASTER segment. Control may subsequently transfer to SUBROUTINE segments by CALL statements or to FUNCTION segments by function references.

Control may revert to a MASTER segment only by means of a RETURN statement in another segment; a MASTER segment name cannot appear in a CALL statement.

EXAMPLE

```
      MASTER TEST
      COMMON/AAA/A,B,C,N
      READ(1,4)A,B,C,N
    4 FORMAT(3E10.5, I4)
      DO 5 K=1,N
    5 CALL TESTA(A,B,C,K)
      STOP
      END
```

SUBROUTINE SEGMENT 7.2

A SUBROUTINE segment consists of a SUBROUTINE statement followed by a series of statements and, finally, an END line (see 7.1).

Each time a subroutine is referred to in a CALL statement (6.5), the statements of the segment are executed.

SUBROUTINE Statement 7.2.1

The first statement of a SUBROUTINE segment is a SUBROUTINE statement of the form

SUBROUTINE $s(a_1,a_2,a_3,\ldots.a_n)$

or

SUBROUTINE s

where 's' is the name chosen by the programmer to identify the SUBROUTINE and each 'a' is a dummy argument. The Subroutine name 's' has the form described in 1.2 and must be distinct from the following names:

(a) the name of any other segment in the program,

(b) the name of any standard function referred to within the program,

(c) the name of any common block used in the program,

(d) the name of any variable, array or Statement function used in the SUBROUTINE segment.

No type is associated with a subroutine name.

Dummy arguments are dealt with in 7.4

Subroutine Body 7.2.2

The body of a subroutine, i.e. those statements that follow the SUBROUTINE statement and define the operations to be performed, may contain any executable or non-executable statements except BLOCK DATA, FUNCTION, SUBROUTINE or MASTER statements.

The segment must include at least one RETURN statement to transfer control back to the statement following the CALL (6.6).

The order of statements within a SUBROUTINE segment is given in 7.5.

Return of Results 7.2.3

Results may be returned by assigning a new value to one or more arguments (subject to 7.4) or to items in common blocks.

Example 7.2.4

A subroutine MMULT to multiply two 10 x 10 matrices could be defined as follows:

```
      SUBROUTINE MMULT(X,Y,Z)
      DIMENSION X(10,10),Y(10,10),Z(10,10)
      DO 20 I=1,10
      DO 20 J=1,10
      Z (I,J)=0
      DO 20 K=1,10
   20 Z(I,J)=Z(I,J)+X(I,K)*Y(K,J)
      RETURN
      END
```

The subroutine could be used to multiply two matrices A and B to produce a matrix C, as follows:

```
      DIMENSION A(10,10),B(10,10),C(10,10)
      . . . . . .
      CALL MMULT(A,B,C)
      . . . . . .
```

FUNCTION SEGMENTS 7.3

A FUNCTION segment consists of a FUNCTION statement followed by the series of statements required to evaluate the function and, finally, an END line (7.1). A FUNCTION segment defines a function in terms of dummy arguments. Each time the function is referred to in an expression, actual arguments are substituted for dummy arguments, the function is evaluated, and its value is transmitted back to the referencing segment.

FUNCTION Statement 7.3.1

The first statement of a FUNCTION segment is a FUNCTION statement, of the form

$$t \text{ FUNCTION } f(a_1,a_2,a_3,....a_n)$$

where t is either INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL or is omitted; f is the name chosen to identify the function; and the 'a's are dummy arguments of the segment.

t indicates the type of the function. If it is omitted, the type is determined by the first letter of the name in the same way as variable names (see 3.1). The type of the function must also be defined in a Type statement in any segment which refers to the function (unless the first letter indicates the type).

The function name f has the form described in 1.2 and must be distinct from the following names:

(a) the name of any other segment in the program,

(b) the name of any standard function referred to in the program,

(c) the name of any common block used in the program.

Dummy arguments are dealt with in 7.4.

Function Body 7.3.2

The body of a function, i.e. those statements that follow the FUNCTION statement and define the calculations to be performed may contain any executable and non-executable statements except BLOCK DATA, SUBROUTINE, FUNCTION or MASTER statements.

The segment must include at least one RETURN statement to transfer control back to the segment containing the function reference.

The order of statements within a FUNCTION reference is given in 7.5.

The name of the function must appear as a variable name within the segment. The value of this variable at the time of execution of a RETURN statement is the value of the function. The function name must not appear in any non-executable statement within the segment.

A FUNCTION segment may also return values by means of its arguments or by common blocks in the same way as a SUBROUTINE segment. Such methods should be used only with care. For example, in the statement

```
VAL=A**2/FUN(A,B)*B**2
```

the function FUN must not alter the values of its arguments or unpredictable results will be obtained.

7.3.4

Example

Suppose it is required to evaluate the areas ofseveral triangles. If the lengths of the sides are referred to as A, B and C, a function AREA could be defined as follows:

```
REAL FUNCTION AREA(A,B,C)
S=(A+B+C)/2.0
AREA=SQRT(S*(S-A)*(S-B)*(S-C))
RETURN
END
```

The function could then be referred to in statements such as

```
RES=AREA(D*3.0,F,97)
VOL=LONG*AREA(B,A,C)+AREA(SIN(B),COS(Z),AREA(F,G,H))
ANS=AREA(T,U,W)+AREA(V,Y,AR(3,1))
```

There is no connection between the variables B, A and C, in the second statement above, and the dummy arguments A, B and C in the FUNCTION segment.

DUMMY ARGUMENTS 7.4

Any dummy arguments of a procedure segment are defined in the normal way within the segment to be variables, arrays or procedures. When there is a control transfer to a procedure, any actual arguments, supplied in the function reference or CALL statement, are effectively substituted for the dummy arguments throughout the procedure before it is executed.

Actual arguments must correspond in order, number, kind and type with the dummy arguments they replace. Further details of the different kinds of dummy argument follow.

Dummy Variables 7.4.1

A dummy variable must not appear in a COMMON, DATA or EQUIVALENCE statement (nor, of course, in a DIMENSION statement). It may appear in a Type statement.

The corresponding actual argument must be an expression of the same type. The expression will be evaluated once and its value substituted for the dummy variable. SUBROUTINE segments allow the additional possibility that the actual argument may be a TEXT constant.

If the argument is used to return a result, i.e. if the dummy variable is assigned a new value before the RETURN statement is executed, the actual argument must be a variable or array element.

EXAMPLE

```
SUBROUTINE CHECK(X,Y,RESULT)
INTEGER Y,....
.....
RESULT=..
RETURN
END
```

A CALL statement referring to this subroutine must have a REAL expression substituted for X, an INTEGER expression substituted for Y, and a REAL variable or array element substituted for RESULT.

Dummy Arrays 7.4.2

A dummy array must be dimensioned in a DIMENSION or Type statement. It must not appear in a DATA or EQUIVALENCE statement.

The corresponding actual argument must be an array name (not an array element) of the same type. If the dummy array is specified to have n elements, the first n elements of the actual array are substituted for the dummy array. Thus, the actual array must be at least as big as the dummy array. It need not have the same structure, e.g.

a 10 x 10 two-dimensional actual array could replace
a 100 element one-dimensional dummy array.

EXAMPLE

```
SUBROUTINE COPY(A)
REAL A(500),MAP(500)
COMMON/MAPBLK/MAP
DO 3 N=1,500
```

```
      3 MAP(N)=A(N)
        RETURN
        END
```

This subroutine copies the first 500 elements of any actual array substituted for A to the common block MAPBLK.

Dynamic Dummy Arrays 7.4.3

When a dummy array is dimensioned, the maximum size of each dimension may be specified by an integer as described in 3.4 (and assumed in 7.4.2), or it may be specified by an INTEGER variable, which must itself may be a dummy argument. The dummy array may thus be a different size each time the procedure is executed. Only dummy arrays may be dimensioned in this way.

Dummy variables used to provide dimension information must not be assigned new values within the procedure.

EXAMPLE

```
      SUBROUTINE COPY(A,I,J)
      DIMENSION A(I,J),MAP(500)
      - - -
      - - -
      END
```

This subroutine processes the first I x J elements of any actual array substituted for A.

The actual argument corresponding to a dummy argument used as a Procedure name must be the name of a SUBROUTINE segment if the dummy argument appears in a CALL statement; or the name of a FUNCTION segment or standard function if the dummy argument is used as a function reference. In the latter case, the standard function must not be an intrinsic function (see Appendix 1).

Any Procedure name used as an actual argument must appear in an EXTERNAL statement in the segment in which it is so used. The EXTERNAL statement has the form

EXTERNAL $V_1,V_2,....V_n$

where the 'V's are names of procedure segments or standard functions excluding intrinsic functions) used as actual arguments. If the statement did not appear, these arguments would be taken to be variables.

EXAMPLE

```
SUBROUTINE GREEN(FUN,X,Y,Z)
X=FUN(Y/Z)
RETURN
END
```

A segment that called the above subroutine could include the following statements:

```
.....
EXTERNAL SIN,COS
.....
CALL GREEN(COS,ALPHA,PHI,SIGMA/C)
....
CALL GREEN(SIN,A1,F1,EXP(C))
....
```

STATEMENT ORDERING 7.5

Within a MASTER, SUBROUTINE or FUNCTION segment, statements must appear in the order:

Type and DIMENSION statements

COMMON statements

DATA statements

EQUIVALENCE statements

Statement function definitions

Executable statements

EXTERNAL statements may appear anywhere before the Statement function definitions.

FORMAT statements may appear anywhere within the segment.

BLOCK DATA SEGMENT 7.6

A BLOCK DATA segment is used to give initial values to items in common blocks by means of DATA statements. Each such segment starts with a non-executable statement of the form

BLOCK DATA

and terminates with an END line (see 7.1). Between the two may appear only Type, EQUIVALENCE, DIMENSION, COMMON and DATA statements.

No executable statements are permitted in a BLOCK DATA segment, which is never executed.

All variables and arrays in a particular common block named in a BLOCK DATA segment must be completely specified (for instance, by DIMENSION statements) even if they do not themselves appear in DATA statements.

A program may contain several BLOCK DATA segments but any particular common block may be given initial values in only one of them. Initial values should not be specified for the blank common block.

Within a BLOCK DATA segment, statements must appear in the following order:

Type and DIMENSION statements

COMMON statements

EQUIVALENCE statements

DATA statements

EXAMPLE

```
BLOCK DATA
INTEGER BETA,GAMMA
LOGICAL OMEGA
DIMENSION KAPPA(10,50)
COMMON/B1/BETA,ALPHA,PI/B2/KAPPA,OMEGA,IOTA
DATA PI/3.14159/,OMEGA/.TRUE./
END
```

Part 2

Input and Output Operations

CONTENTS

Chapter 1

INTRODUCTION

This chapter gives a brief summary of the input and output facilities available in FORTRAN.

CHARACTER SET 1.1

The characters that may be input and output by a 1900 FORTRAN program are those in the 1900 printing set (see Appendix 3).

RECORDS 1.2

In FORTRAN, all data and results are handled in the form of records. These records are held on media such as punched cards, punched paper tape or magnetic tape. Two kinds of record are recognised: Formatted and Unformatted. A Formatted record might be a string of characters on one card or one line of print. The record is considered to be split into fields, where each field normally represents the value of one variable or array element or is a string of TEXT information. An Unformatted record is information in essentially internal machine form. When output it is normally intended only for subsequent re-input, not for examination by the programmer. Unformatted records are most appropriately used on backing-store media such as magnetic tape.

READ AND WRITE STATEMENTS 1.3

Formatted records are input and output by means of Formatted READ and Formatted WRITE statements. Unformatted records are input and output by means of Unformatted READ and Unformatted WRITE statements.

Normally each of these statements contains a list of variables, array elements and array names. The value of each item in this input/output list will be transferred when the statement is executed.

FORMAT INFORMATION 1.4

Each Formatted READ or WRITE statement will have associated with it information about the external format of the data it is to input or output. This information is normally provided in a FORMAT statement, which specifies for each of the items in the input/output list such things as the number of characters in the external field and the positions of any decimal points.

Items in the input/output list and items in the Format specifications are paired dynamically. Any action performed will depend on information jointly provided by the next item in the input/output list and the next item in the Format list.

Each input and output statement will refer, by a number chosen by the programmer. to a specific input or output peripheral. For example. if a programmer wished to use one card reader. one paper tape reader and a line printer in a program, he might chose to refer to these by the numbers 1,2, and 3 respectively. He would then associate these numbers with actual types of peripherals by special statements within the 1900 Program Description.

CONTROL OF PERIPHERALS 1.6

FORTRAN also provides 3 auxiliary input/output statements:- REWIND, BACKSPACE and ENDFILE. As their names suggest, these statements are more appropriately used to control some devices, magnetic tape for example, then others.

Some input and output subroutines are provided in 1900 FORTRAN to perform operations not allowed for in standard FORTRAN. These subroutines - ALLOT, RLEASE, RUNOUT and DISENG - are used, for example, to dynamically control which peripherals, are currently available to the program, an advantageous feature on multiprogramming machines.

This chapter has merely introduced the input/output system used in FORTRAN. All the topics introduced here will be dealt with in detail in later chapters.

Chapter 2

READ AND WRITE STATEMENTS

This chapter describes in detail the statements used in FORTRAN to input and output records.

INPUT/OUTPUT LISTS 2.1

READ and WRITE statements normally contain a list of the items whose values are to be transmitted. The list has the form

$$e_1, e_2, e_3, e_4, \ldots\ldots e_n$$

where each e is a list element and may be a variable, an array element, an array name, or a DO-implied list (described below).

Each variable or array element that appears in a list specifies that the value of that item is to be transmitted. An array name specifies that the values of all the elements of that array are to be transmitted, in the order given in Part 1, Section 3.4.1.

EXAMPLES

```
A,I(J,K), Y(2*L-3), ARRAY
MATRIX, X,B, AR(3,7,2), AR(3,7,3), AR(3,7,I+2)
REL, INTG, COMP, DUBL, LOGCL.
```

The subscripts of array elements may have any of the general forms given in Part 1, Section 2.3.1.

An element of a list may be a DO-implied list. The form and effect of a DO-implied list closely resembles that of a DO statement (see Part 1, Section 6.3).

A simple DO-implied list has the form

$$(e_1, e_2, \ldots\ldots e_n, i = m_1, m_2, m_3)$$

Where each e is a list element as defined above and i, m_1, m_2 and m_3 have the same form and meaning as in the DO statement; the omission of m_3 has the same effect as in a DO statement.

A simple example in the DO-implied list

$$(A(I), I = 1, 10)$$

which would have the same effect as the input/output list:

A(1), A(2), A(3), A(4), ... A(10)

As any e may itself be a DO-implied list, the DO's may be nested to any depth. The names in these lists are not restricted to array names. If a variable occurs then either the value of this variable will be output several times, or, on input, different values will be assigned successively to the same variable, each value overwriting the previous value.

An example of a more complex list is

((AR(I,J), J=1,10),BR(I+1,I), CR(I,I+1), I=1,10)

which might be thought of as the 'pseudo-program' :

```
      DO  5  I=1, 10
      DO  3  J=1, 10
    3 AR  (I,J)
      BR  (I+1,I)
    5 CR  (I,I+1)
```

The values being read or written would be equivalent to the input/output list:

```
AR(1,1), AR(1,2), .....AR(1,10), BR(2,1), CR(1,2),
AR(2,1), AR(2,2), .....AR(2,10), BR(3,2), CR(2,3).
    ..........................................
AR(10,1),AR(10,2), ......AR(10,10), BR(11,10), CR(10,11)
```

Further examples of input/output lists are given below

EXAMPLES

```
(AR1(I), AR2(I), I=1,5)
((( A(J,K), K=1,21,2), B,C(L,J), L=1,10),J=1,25,2)
(((( X(J,K), INT(K,L,M), J=1,5),K=1,5),L=1,5),M=1,5)
     K,L,(VAR(J, J+1, J=K,L,3), NEXT
```

If m_1, m_2 or m_3 are INTEGER variables they may themselves appear as elements in an input/output list, subject to the same restrictions as for DO statements - the values of the parameters must not be altered in the 'body' of the DO.

The way in which the input/output list interacts with the Format specification is described in 3.3.

FORMATTED WRITE 2.2

A formatted WRITE statement has one of the forms

```
WRITE (u,f) k
```

or

```
WRITE (u,f)
```

where u is an INTEGER constant or INTEGER variable that identifies a particular peripheral unit; f is either the lable of a FORMAT statement or an array name; and k is an input/output list.

When the statement is executed, the values of the items in the list will be output. in the order in which they appear in the list, to create one or more new records on the unit u according to the format specified by f. The correspondence between list items and the format specification is discussed in 3.3.

If f is a FORMAT statement label then that statement must be in the same segment as the WRITE statement. If f is an array name then the array must contain format information as described in 3.4

The second form of the statement may be used to write blank records or records containing a fixed character string.

EXAMPLE

```
19 FORMAT  (/16HEXAMPLE OF TITLE/)
   WRITE (3, 19)
```

The above statements cause a record consisting of

```
EXAMPLE OF TITLE
```

to be output, preceded and followed by a blank record. It is assumed that the peripheral unit 3 is not identified with a line printer.

FORMATTED READ 2.3

A Formatted READ statement has one of the forms

```
READ  (u, f) k
```

or

```
READ  (u, f)
```

where u is an INTEGER constant or INTEGER variable that identifies a particular peripheral unit; f is either the label of a FORMAT statement or an array name; and k is an input/output list.

Each field in the external record should be presented in the format specified by f, and in the order required by the list k. The correspondence between list items and the format specification is discussed in 3.3.

When the statement is executed the reading of a new record is initiated. Further records may be read as required by the format specification.Each field read will be converted to the internal form of the appropriate type and assigned, in order, to the elements of the list k.

If f is the label of a FORMAT statement then that statement must be in the same segment as the READ statement. If f is an array name then the array must contain format information as described in 3.4.

A READ statement always initiates the reading of a new record. Any fields at the end of a previous record which have not been read will be lost. For example, if one card was a record containing 10 fields and this was read by a READ statement which required only the first 5 fields, then the next READ statement would read the next card and the 5 remaining fields on the first card would be ignored.

The second form of the READ statement might be used in the next record or records on a particular input device are not required. Although as many fields or records as are specified by the format information are read, no assignment takes place (except, possibly to an H descriptor within the format information) and the effect is to make the following record available to the next READ statement.

UNFORMATTED WRITE 2.4

An Unformatted WRITE statement has the form

```
WRITE (u) k
```

where u is an INTEGER constant or INTEGER variable that identifies a particular peripheral unit; and k is an input/output list.

When the statement is executed, the values of the items in the list k will be output, in the order in which they appear in the list, to create one new record on the unit u. The record will be in internal machine form, normally later re-input to the processor.

Each Unformatted WRITE statement will create one new record, irrespective of the number of items in the list k.

UNFORMATTED READ 2.5

An Unformatted READ statement has one of the forms

```
READ (u) k
```

or

```
READ (u)
```

where u is an INTEGER constant or INTEGER variable that identifies a particular peripheral unit; and k is an input/output list.

The external record should be in internal machine form and is normally created by a previous Unformatted WRITE statement.

When the statement is executed the whole of the next record from the unit u will be read and values will be assigned, in order, to the elements specified in the list k. The number of elements in the list k may be less than or equal to the number of values in the record but must not exceed this number.

The second form of the statement might be used if the next record is not required. Although the record is read, no assignment takes place, and the effect is to make the following record available to the next Unformatted READ statement.

This chapter describes the format information which is used in conjunction with Formatted READ and WRITE statements and provide information on the external character representation of data. This information is normally contained in a FORMAT statement but may be held in an array.

FORMAT STATEMENTS 3.1

A FORMAT statement is a non-executable statement with the form

FORMAT (s)

where (s) is the format specification. Every FORMAT statement must be labelled and may appear anywhere within a MASTER, SUBROUTINE or FUNCTION segment.

The statement may be referred to in one or more READ and WRITE statements in the same segment as itself.

FORMAT SPECIFICATION 3.2

Field Descriptors 3.2.1

A format specification is a series of slashes, commas and field descriptors enclosed in parentheses.

The field descriptors are of the general forms:

s r F w d
s r E w d
s r G w d
s r D w d
r I w
r L w
r A w
w H $h_1h_2h_3h_4.....h_w$
w X

where the capital letters indicate the manner of conversion and are called conversion codes; w represents the width of the field in the external character string; d represents the number of character spaces in the fractional part of a number (except in the G code on output); s is a scale factor designator and r is a repeat count. Each field descriptor and its effect is described in detail later in this chapter.

The transfer of INTEGER values is effected by the I conversion code, REAL values by the E,F, and G codes, DOUBLE PRECISION values by the D code, and LOGICAL values by the L code. Because a COMPLEX value can be considered as two REAL values, transfer of COMPLEX quantities is effected by two E,F or G conversion codes. Information in character form is transferred by means of the A or H codes.

The X code indicates fields to be ignored.

Scale Factors 3.2.2

A scale factor is of the form.

```
n P
```

where n is an integer, possibly preceded by a minus sign. It may appear with an F, E, G or D code as shown in 3.2.1. A scale factor is normally used when numbers are too large or too small for convenient processing by the program. The positive or negative value of n specifies a power of 10 by which the number is to be divided on input or multiplied on output.

A scale factor affects only F, E, G or D descriptors. When a format specification is first referenced, a scale factor of zero is assumed. If a scale factor is encountered during the scanning of the format specification it operates on all subsequent E, F G or D descriptors until a new scale factor is encountered. The precise effect of a scale factor on the various descriptors is described in 3.5

Field Separators 3.2.3

Field descriptors and groups of field descriptors (see below) must be separated by a comma, a slash or a series of slashes. Other slashes may appear before or after a series of field descriptors.

A comma or slash marks the end of a field. A slash also marks the end of a record. On input, a series of n slashes will cause n - 1 records to be skipped; on output, n slashes will output n - 1 blank records. For example, the statement.

```
FORMAT (I4, I7/E7.2 /// I5)
```

associated with a WRITE statement will cause a record with two integer numbers; a record with one floating point number; two blank records; and a record with one integer number, to be output.

The parentheses at the beginning and end of the specification may be considered to initiate a new record and terminate a record respectively.
For example, on output, the statement

```
FORMAT (/// I7 ///)
```

will create three blank records; one record with one integer number; and three more blank records.

Repetition of Descriptors 3.2.4

The repetition of an item in the format specification list may be achieved by writing an integer (r in the list in 3.2.1) in front of the item. This will have the same effect as specifying the item r times. Here, an item means a single field descriptor, or a group of field descriptors enclosed in parentheses. The descriptors wH and wX cannot be repeated, unless enclosed in parentheses.

The repeat count, or group repeat count, must be a positive integer and not zero. If a group enclosed in parentheses has no group repeat count in front of it, then a count of one is assumed.

A group of descriptors may contain further groups enclosed in parentheses, each with a repeat count. In standard FORTRAN these inner groups may not contain further groups. In 1900 FORTRAN, groups may be nested to any depth.

EXAMPLES

```
3 FORMAT (13, 3I5, 6(/E7.2, E9.3), 2F9.4)
7 FORMAT (14, L6, 10(10X, 3L12,I4)/2I5)
```

ACTION OF FORMAT SPECIFICATION 3.3

When a Formatted READ or WRITE statement is executed, each action performed will depend on information jointly provided by the next element in the input/output list (if one exists) and the next field descriptor obtained from the format specificiation.

Except for the effects of repeated groups, the format specification is interpreted from left to right. A new record is commenced at the start of execution of the READ or WRITE statement and additional records are commenced as the format specification requires.

If there is an input/output list, at least one field descriptor other than wH or wX must appear.

When an item from the input/output list is selected the next available field descriptor is examined. If it is wH or wX, appropriate action is taken and the next field descriptor is examined. The process is repeated until a descriptor other than wH or wX is found. This must of of the appropriate type for the list item, as indicated in 3.2.1. The value is transmitted and the new item selected from the input/output list. The whole process is then repeated.

When all the elements in the input/output list have been operated upon, the next field descriptor is examined to see if it is wH or wX. If it is not, then execution will cease. If it is, then this descriptor will be acted upon and the next descriptor tested, and so on until a descriptor other than wH or wX is reached, when execution will cease. A special case of the above is when the format specification contains only wH and wX descriptors and there is no input/output list.

If the format specification has been completely scanned and there are still items left in the input/output list, then a new record will be commenced and the scanning will be repeated as follows.

1. If there are no internal parentheses then scanning is repeated from the beginning of the specification.
2. If there are internal parentheses then scanning is repeated from the left parenthesis corresponding to the right-most internal right parenthesis.

EXAMPLES

```
FORMAT  ↑(..............)
FORMAT (... ↑(............))
FORMAT  (...  ↑(.... (....)..).........)
FORMAT  (....(....).....↑(.(.).())..)
```

The arrows show where repetition would re-commence.

The last example should clarify rule 2, above.

FORMAT SPECIFICATION IN ARRAYS 3.4

Any of the formatted READ or WRITE statements may contain an array name in place of a FORMAT statement label. An array so referenced must contain a valid format specification. That is, a format specification, complete with enclosing parentheses, must be present in character form. It may have been previously read into that array by means of an A format specification or inserted by means of a TEXT constant. The specification need not entirely fill the array, and it is immaterial what information is held in the part of the array following the final right parentheses. A format specification contained in an array must not contain any wH field descriptors.

EXAMPLES

```
(1) SUBROUTINE INPUT (UNIT,INFORM)
    INTEGER UNIT
    DIMENSION INFORM (15), A(100), M(50)
    COMMON /IOLIST/I,X,A,J,M
    READ (UNIT,INFORM) I,X,A,J,M
    RETURN
    END
```

The above subroutine might be called several times, with the actual array that replaces the dummy array INFORM containing a different format specification each time.

```
(2) ..........
    DIMENSION LAYOUT (10)
    .........
    READ (2,10) LAYOUT
 10 FORMAT (10 A 8)
C  THE ABOVE READ INPUTS A FORMAT TO THE ARRAY LAYOUT
    .........
```

```
   READ (2,LAYOUT) X,Y,I, LOGIC, NO
C THE ABOVE READ USES THE FORMAT PREVIOUSLY
C READ TO LAYOUT TO INPUT THE DATA
```

FIELD DESCRIPTORS 3.5

This section describes the field descriptors available in FORTRAN. The character b is used to indicate a space on the external record.

I 3.5.1

The conversion code I is used for the transfer of the values of INTEGER data. the descriptor has the form

Iw

where w is the field width, an integer indicating the number of characters in the external representation of the number.

OUTPUT

The descriptor will cause the value of the corresponding list element to be output as an integer number, right justified and occupying w character positions in the external record. A negative sign will precede the first digit if the number is negative. Positive numbers will not be signed. Leading blanks will appear if necessary to make up the w character positions.

The number of characters to be output should not exceed the field width. If this error occurs, an asterisk, followed by the whole number, will be output. This means that any values output afterwards will be displaced to the right, destroying the expected layout.

EXAMPLES

DESCRIPTOR	INTERNAL NUMBER	EXTERNAL NUMBER
I5	+3659	b3659
	-987	b-987
	+3	bbbb3

INPUT

The descriptor will cause the next w characters in the external record to be read and converted to INTEGER form. This value will be assigned to the corresponding list element.

The characters of the external field may be a signed or unsigned integer number. Numbers without signs will be assumed positive. Blanks before the first digit are ignored but must be included in the character count. Other blanks in the field are treated as zero. A field of all blanks is considered zero. The field must not contain any decimal point or exponent.

EXAMPLES

DESCRIPTOR	EXTERNAL NUMBER	INTERNAL NUMBER
I5	b+393	+393
	bbb23	+23
	-5858	-5858
	b23bb	+2300
	bbbbb	0

E 3.5.2

The conversion code E is used for the transfer of a REAL value or one of the components of a COMPLEX value. The descriptor has the form

E w.d

where w is the field width, an integer indicating the number of characters in the external representation of the number, and d is the number of digits in the fractional part of this number. In all cases w must be greater than or equal to d.

OUTPUT

The specification will cause the value of the corresponding list element to be output as a decimal fraction with an exponent. The fraction f will be within the limits

$$0.1 \leq f < 1$$

and will be rounded to d digits, preceded by a zero and a decimal point. The exponent has one of the forms

$$Ebd_1d_2 \quad \text{or} \quad E\text{-}d_1d_2$$

where d_1 and d_2 are digits. The number will be right justified. Both fraction and exponent will be signed only if negative.

Leading blanks will appear if necessary, to make up the w character positions.

The number of characters to be output should not exceed the field width. If this error occurs an asterisk, followed by the whole number, will be output, thus destroying the expected layout.

If a scale factor of n is in operation (see 3.2.2) the value is unchanged but the form is modified. The fraction will be multiplied by 10^n and the exponent will be reduced by n.

EXAMPLES

DESCRIPTOR	INTERNAL NUMBER	EXTERNAL NUMBER
E14.5	+12345678	bbb0.12346Eb08
	-1.23	bb-0.12300Eb01
	+.000123	bbb0.12300E-03
	-.003	bb-0.30000E-02
3PE14.5	+1234.5678	bbb123.457Eb01
	-1.23	bb-123.000E-02
	-.003	bb-300.000E-05

INPUT

The descriptor will cause the next w characters in the external record to be read and converted to REAL form. This value will be assigned to the corresponding list element. If the element is of Type COMPLEX, then two descriptors are required.

The w characters of the external field consist of an optional sign, followed by a string of digits optionally containing a decimal point, and optionally followed by an exponent part.

The exponent part may be one of the forms:

1) a signed integer constant
2) E or D followed by a signed integer constant
3) E or D followed by an unsigned integer constant

The exponent field has a maximum width of four characters.

Numbers and exponents may be signed. Unsigned numbers and exponents will be assumed positive. Blanks before the first digit are ignored but must be included in the character count. Other blanks in the field are treated as zero, except within an exponent. Blanks within an exponent are ignored. A field of all blanks is considered zero.

If the field contains a decimal point, then this will over-ride the implied point specified by .d in the descriptor.

If a scale factor of n is in operation (see 3.2.2), the number will be divided by 10^n unless there is an exponent in the external field. If there is an exponent, the scale factor has no effect.

EXAMPLES

DESCRIPTOR	EXTERNAL NUMBER	INTERNAL NUMBER
E7.3	7654321	+7654.321
	b+137-3	+.000137
	1.234E2	+123.4
	-123E02	-12.3
3PE7.3	7654321	+7.654321
	b+137-3	+.000137

The conversion code F is used for the transfer of a REAL value or one of the components of a COMPLEX value. The descriptor has the form

F w.d

where w is the field width, an integer indicating the number of characters in the external representation of the number, and d is the number of digits in the fractional part of this number.

In all cases, w must be greater than or equal to d.

OUTPUT

The descriptor will cause the value of the corresponding list element to be output as a decimal fraction, rounded to d decimal places. The number is right justified and a minus sign will precede the first digit if the number is negative. Leading blanks will appear if necessary, to make up the w character positions. Trailing zeros will appear if necessary, to make up the decimal places. If the number has no integral part, then a zero will precede the decimal point if the field is wide enough.

The number of characters to be output should not exceed the field width. If this error occurs, then an asterisk, followed by the whole number, will be output, thus destroying the expected layout. Allowance should be made for the decimal point and a minus sign.

If a scale factor of n is in operation (see 3.2.2) the external value will be 10^n times larger than the internal number.

EXAMPLES

DESCRIPTOR	INTERNAL NUMBER	EXTERNAL NUMBER
F10.4	+5227.3278	b5227.3278
	-345.6789	b-345.6789
	+12.3	bbb12.3000
	-3.21989623	bbb-3.2199
3PF14.4	+5227.3278	bb5227327.8000
	+12.3	bbbb12300.0000

INPUT

The form and effect of the external field on input is the same as it is for the E conversion code.

The conversion code G is used for the transfer of a REAL value or one of the components of a COMPLEX value. The descriptor has the form

G w.d

where w is the field width, an integer indicating the number of characters in the external representation of the number. On output, d represents the number of significant digits in the field. On input, d represents the number of digits in the fractional part of the external character string.

OUTPUT

The specification will cause the value of the corresponding list of elements to be output in a similar format to the output from either the E or F specifications, depending on the magnitude of the number. This is illustrated by the following example.

Suppose it is required to output a list of values, each one accurate to six significant digits. The specification G 13.6 might cause the following to be output:

bb123.456bbbb

b-3.45678bbbb

bb835954.bbbb

b0.293761bbbb

-0.340000E-07

b0.828895Eb09

Each number is printed as a decimal fraction, accurate to six significant digits, so long as the value, N, lies within the range:

$.1 \leq N < 10^{d}$

If N lies outside the range then the value is output with an exponent, as in the E specification.

Note that four blanks are left after the value unless it has an exponent. Allowance should be made, when specifying the field width, for these blanks and also for the sign and the decimal point. The following table indicates the correspondence between the value, N, of the list element, and the equivalent method of conversion.

Magnitude of Value	Equivalent Conversion
$0.1 \leq N < 1$	F (w-4).d, 4X
$1 \leq N < 10$	F (w-4).(d.1), 4X
.....	.....
.....	.....
$10^{d-2} \leq N < 10^{d-1}$	F (w-4).1, 4X
$10^{d-1} \leq N < 10^{d}$	F (w-4).0, 4X
Otherwise	E w.d

If a scale factor is in operation it will have no effect unless the value lies outside the range

$$0.1 \leq N < 10^{d}$$

when it will have the same effect as in the E conversion code.

EXAMPLES

INPUT

DESCRIPTOR	INTERNAL NUMBER	EXTERNAL NUMBER
G11.4	+10.34567	bb10.35bbbb
	-.000367	-0.3670E-03
	+4958672	b0.4959Eb07
	+39.7	bb39.70bbbb
2PG11.4	+.766789	b0.7668bbbb
	+10.3456	bb10.35bbbb
	-.00036	-36. 00E-05

The form and effect of the external field on input is the same as for the E conversion code.

The conversion code D is used for the transfer of a DOUBLE PRECISION value. The descriptor has the form

D w.d

where w is the field width, an integer indicating the number of characters in the external representation of the number, and d is the number of digits in the fractional part of this number. In all cases, w must be greater than or equal to d.

OUTPUT

The form of the external field on output is the same as for the E conversion code.

INPUT

The form of the external field on input is the same as for the E conversion code. The characters input will be converted to DOUBLE PRECISION form and stored as the value of the corresponding list element.

The conversion code L is used for the transfer of LOGICAL values. The descriptor has the form

L w

where w is the field width, an integer i.e. the number of characters in the external representation of the value.

OUTPUT

The descriptor will cause w-1 blank characters to be output, followed by a T or F as the value is.TRUE. or.FALSE.

INPUT

The descriptor causes a field of w characters to be read and converted to the internal representation of either .TRUE. or .FALSE. The external field consists of optional blanks followed by one of the characters T or F, representing true or false, followed by any other characters.

The conversion code A is used for the transfer of information in character form. The descriptor has the form

A w

where w is the field width, an integer indicating the number of characters in the external field.

OUTPUT

The specification will cause w characters to be output. The list element corresponding to this descriptor is assumed to contain eight characters. If w is less than or equal to eight when the w left-most characters are output. If w is greater than eight then the output will be w-8 blanks followed by the eight characters of the list element.

INPUT

The specification will cause 8 characters to be stored in the corresponding list element, irrespective of the type of this element. If w is less than 8 then the characters will be stored with 8-w blank characters inserted on the right. If w is greater than or equal to 8, then the 8 right-most characters will be stored.

The conversion code H is used to transfer character information held in the format specification. The descriptor has the form

$$wHh_1h_2h_3....h_w$$

where w is the field width and each h is a character in the 1900 printing character set (see Appendix 3). The field width is the same as the number of characters in the descriptor. No list item is associated with an H descriptor.

EXAMPLE

```
11HTHE TIME IS
```

The 2 spaces as well as the 9 letters must be included in the field width.

OUTPUT

The descriptor will cause the w characters following the H code to be written as part of the output record. For example, the following statements could be used to print a heading followed by a value of the variable IDATE.

```
      WRITE (3,20) IDATE
   20 FORMAT (21HMATRIX MULTIPLICATION, I6)
```

The variable IDATE will be output according to the descriptor I6.

INPUT

If the H conversion code is used in conjenction with a READ statement, then w characters will be read from the external record and stored in the format specification itself. The characters already present in the descriptor will be over-written. A subsequent WRITE operation will cause these new characters to be output.

The conversion code X is used to skip characters on the input record or to output blank characters. The descriptor has the form

w X

where w is the field width, an integer constant indicating the number of character positions to be skipped or written as blank characters.

No list item is associated with an X descriptor.

OUTPUT

The specification will cause w blank characters to be output on the external record.

INPUT

The specification on input will cause the next w characters of the record to be skipped.

In 1900 FORTRAN, special forms of the I,D,E,F and G descriptors are provided to allow input records to have a free format. If w and d are zero in any of these descriptors, e.g. F0.0, then the current record is scanned from the field previously read, until the number field is encountered. During this scanning, spaces (and Horizontal Tabs on paper tape) are ignored. The field is terminated by the first space within it or by the end of the record. Obviously, spaces are not allowed within numbers to be input by these descriptors (except after an E or D of an exponent). If the end of a record is reached before a number field is found, a new record is read and scanning continues. Any number of blank records may be scanned and ignored in this way.

EXAMPLE

The following sequence reads 100 REAL number in free format

```
      READ (1,5) (A(I), I = 1,100)
      ....
5 FORMAT (100 F0.0)
```

The following sequence does not necessarily have the same effect since a new record will be commenced before each number is read:

```
      READ (1,5) (A(I), I = 1,00)

5 FORMAT (F0.0)
```

Chapter 4

OTHER INPUT/OUTPUT OPERATIONS

This chapter describes the auxiliary input/output statements used chiefly to control backing store peripherals and the input/output subroutines provided in 1900 FORTRAN.

AUXILIARY INPUT/OUTPUT STATEMENTS 4.1

The Auxiliary input/output statements described below, like the unformatted statements, are primarily intended for use with backing-store peripherals such as magnetic tape units. Further information on the effect of these statements on various media is given in chapter 5.

REWIND 4.1.1

A REWIND statement has the form

```
REWIND u
```

where u is an INTEGER constant or INTEGER variable that identifies a particular peripheral unit.

Execution of this statement causes the unit identified by u to be positioned ready for the first record to be read. If the unit is already at its initial point, the statement has no effect.

BACKSPACE 4.1.2

A BACKSPACE statement has the form

```
BACKSPACE u
```

where u is an INTEGER constant or INTEGER variable that identifies a particular peripheral unit.

Execution of this statement causes the unit identified by u to be positioned ahead of the previous record. If the unit is at its initial point the statement has no effect.

ENDFILE 4.1.3

An ENDFILE statement has the form

ENDFILE u

where u is an INTEGER constant or INTEGER variable that identifies a particular peripheral unit.

Execution of this statement causes the recording of a special end-of-file record on the unit identified by u.

INPUT/OUTPUT SUBROUTINES 4.2

Some operations connected with input and output that are useful on 1900 series computers are not provided for in standard FORTRAN. Accordingly, subroutines are provided in 1900 FORTRAN for these operations.

Allocations and Release of Peripherals 4.2.1

At the start of the program run no peripherals are allocated to the program. As each peripheral is referred to by READ, WRITE or other statements it is automatically allocated to the program by the 1900 system. If no such peripheral is available (e.g. a card reader is required by the program and it is allocated to another program, or a required magnetic tape tile is not loaded on a tape deck) the program is suspended awaiting operator action.

The efficiency of a multi-programming processor is increased if any peripherals no longer required program are released for use by another program. In 1900 FORTRAN this may be achieved by a call of the subroutine RLEASE. The general form is:

CALL RLEASE (u)

where u is an expression of type INTEGER that specifies the peripheral unit. If the peripheral has already been released, the call has no effect. Any future reference to the peripheral will re-allocate it if possible. Programs should release a peripheral that will be needed again only if it is not required for a considerable length of time.

When a peripheral is not allocated to a program, either because it has not yet been referred to or because it has been released, it may be so allocated by a call of the subroutine ALLOT. The general form is

CALL ALLOT (u, L)

where u is an expression of type INTEGER that specifies the peripheral unit and L is any variable or array element of type LOGICAL. If unit u is already allocated to the program, L is set to.TRUE. and there is no further effect. It unit u is not already allocated to the program and is available, it is allocated and L is set to .TRUE. If it is not available, L is set to .FALSE. and there is no further effect.

The advantage of using ALLOT rather than relying on automatic allocation is that the program may be able to continue if the unit is not available, possibly by using a unit of a different type, e.g. paper-tape punch instead of a line printer. This advantage is gained only on line printers, card reader and punches, and paper-tape readers and punches. For magnetic tape the program is always suspended until the required tape is loaded. It follows that for magnetic tape the L is always set to .TRUE. or the program is suspended.

Disengagement of Peripherals 4.2.2

The subroutine DISENG is provided to allow a program is disengage a peripheral. A call has the form

```
CALL DISENG (u)
```

where u is an expression of type INTEGER that specifies a particular peripheral unit to be disengaged. Any further reference to it will suspend the program until the unit has been re-engaged by the operator. The operator's attention may thus be drawn to some action needed on the unit, e.g. special pre-printed paper may be needed on a line printer.

RUNOUT 4.2.3

A call of the subroutine RUNOUT has the form

```
CALL RUNOUT (u)
```

where u is an expression of type INTEGER that specifies a peripheral unit. Normally, u will be a paper tape punch and four inches of runout will be output. However, it could be any output device; the actions taken are described in Chapter 5.

Chapter 5

INPUT/OUTPUT MEDIA ON THE 1900 SERIES

This chapter describes the effect on the input and output media of 1900 series computers of the FORTRAN operations introduced in chapters 1 to 4.

THE INPUT, OUTPUT, USE AND CREATE STATEMENTS 5.1

Within a FORTRAN program, peripheral units are referred to by a distinct integers (in the range 0 to 4095 for 1900 FORTRAN). When the program is to be run on a 1900 series computer these integers must be related to the 1900 system names of actual peripheral units. It is permissible for several different numbers to be used within the program and associated with the same physical peripheral unit.

The above association must be made within the 1900 FORTRAN Program Description by INPUT, OUTPUT, USE or CREATE statements. The general forms are given by

$$t \ \ m_1, m_2 \ldots m_q = P$$

where t is one of INPUT, OUTPUT, USE or CREATE, the m's are the unit numbers used within the program to refer to the peripheral specified by P.

The simplest form of P is pn, where p is a two-letter code for the type of peripheral and n ($0 \leq n \leq 15$) designates an individual unit of that type. Thus, TR0 and TR2 are two different paper-tape readers. It is conventional, but not essential, to refer to unit of one type numbered upwards from zero. Thus, if a program used two paper-tape readers they would normally be called TR0 and TR1. (These names do not refer to fixed geographical units, since the 1900 system can make any available paper tape reader into TR0 of the program).

For some types of peripheral, P is extended to give information about such things as file names. Details appear in the appropriate sections below.

If several different numbers are used within a program for the same peripheral unit they should normally all be listed in one statement in the Program Description. This rule need not be followed if the unit is released in one guise (by the use of the RLEASE subroutine - see 4.2.1.) before it is referred to in another.

The different statements above indicate the mode of use of the unit as follows:

INPUT	Input only
OUTPUT	Output only
USE	Input and output
CREATE	Input and output

Details of which statements may be used for particular types of peripheral appear in later sections. CREATE is used rather than USE if a new name is to be assigned to a file.

These statements obey the normal rules for 1900 Program Description statements i.e. they must start later than column 6 and cannot have continuation lines.

EXAMPLES

```
INPUT 1 = TRO
OUTPUT 2 = LP2
OUTPUT 3, 9, 5 = CP1
USE 6 = MTO (SURVEY DATA)
```

80 column cards may be used for input and output. The standard 1900 card code is used (see Appendix 3). The two-letter codes for card readers and punches are CR and CP respectively. Examples of appropriate Program Description statement are:

```
INPUT    1 = CR1
INPUT 5, 9 = CR0
OUTPUT   2 = CP0
OUTPUT 3,101 = CP2
```

The USE and CREATE statements must not be used for cards.

A Formatted record is one punched card. On input, the first non-blank card is taken to be the first genuine record.

An ENDFILE statement outputs a card containing only

```
****
```

in the first four columns.

The subroutine RUNOUT outputs two blank cards.

Paper Tape 5.3

8-track paper tape may be used for input or output. The standard 1900 paper tape code is used (see Appendix 3). The two-letter codes for paper tape readers and punches are TR and TP respectively. Examples of appropriate Program Description statements are:

```
INPUT  2 = TR0
INPUT 3,7,99 = TR2
OUTPUT 12 = TP1
OUTPUT 5,32 = TP0
```

The USE and CREATE statements must not be used for paper tape.

A Formatted record is one 'line' or more precisely, a string of characters in the 1900 printing set, terminated by a newline character and three Null characters; the Null characters are not essential for an input record. FORTRAN imposes no limit on the length of a line. The only limitation on line length is therefore that imposed by the tape-editing equipment used to punch or print the tape.

On input, the first non-empty line is taken to be the first genuine record, i.e. superfluous newline characters before the first record are ignored. Once the first record has been read, all records are taken to be at least 80 characters long. Shorter records (including completely empty records) are filled up to 80 characters by the addition of space characters to the end of the record. No spaces are added if the record is 80 or more characters long.

The characters Null and Erase are completely ignored on input. Other paper tape characters not in the 1900 printing set are either translated into some reasonably equivalent character (e.g. Horizontal Tab becomes space) or translated into!.

An ENDFILE statement outputs a line of four asterisks, followed by the paper tape stop characters TC_4 and DC_4, a line of Erase characters and twelve inches of blank tape.

The subroutine RUNOUT output four inches of blank tape.

Line Printer 5.4

A line-printer may be used for output. The standard 1900 printing character set is given in Appendix 3. The two-letter code for a line printer is LP. Examples of appropriate Program Description statements are:

```
OUTPUT  6    = LP0
OUTPUT 3,17 = LP1
```

The INPUT, USE and CREATE statements must not be used for a line printer.

A formatted record on a line printer may be a line of 97, 121 or 161 characters depending on the type of printer. The first character in a line to be printed is taken as a paper control character and is not printed. These paper control characters are translated as follows:

space	one line feed
0	two line feed
1	throw to top of new page
+	no advance

In 1900 FORTRAN 6 further characters are permitted:

2 - 7	throw to channel 2 - 7 (see below)

The line printer contains a loop of 7 track paper tape which is the same length as one page. This loop revolves as the paper is fed through the printer. It is conventional to punch a hole in channel 1 of this tape in the position corresponding to the top of a page and the other channels may be punched in any position. A throw to channel 5, say, will cause the paper to be fed through the printer until a hole is encountered in channel 5 of the paper tape loop. The format of the paper tape loop may be varied to suit the program and installation.

Any other character which occurs as the first character of a line to be output will be treated as space.

The subroutines ENDFILE and RUNOUT will cause a throw to the top of a page.

Magnetic tape may be used for input and output or as a backing store. Standard 1900 half-inch magnetic tapes are used. 1900 Magnetic Tape Housekeeping conventions are observed. Each reel of tape is a simple file with its own file name and can be used only by programs that specify it by name, or it is a 'scratch tape' that is available for use by program. The two-letter code for a magnetic-tape unit is MT. The Program Description statements that refer to magnetic tapes have the following general forms:

t m_1, m_2, ... m_q = MTn (f)

t m_1, m_2, ... m_q = MTn/FORMATTED (f)

where t is one of INPUT, OUTPUT, USE or CREATE

m_1, m_2 ... m_q and n are as in 5.1

f is a file name consisting of up to twelve characters chosen from the set:

A to Z
0 to 9
space
minus sign (hypen)

The name must start with a letter. Note that spaces within a file name are significant.

The two forms of the statements are used for files ccntaining unformatted and formatted information respectively. The different values of t have the following meanings:

INPUT	Obtain the named magnetic tape and allow it to be used for input only.
OUTPUT	Obtain the named magnetic tape and allow it to be used for output only.
USE	Obtain the named magnetic tape and allow it to be used for input and/or output.
CREATE	Obtain a scratch tape, rename it, and allow it to be used for output followed by input if required.

The file name and enclosing parentheses may be omitted in the USE statement. Then a scratch tape is obtained and remains scratch at the end of the program. This will be the normal system if the tape is to be used purely as a work tape. (usually with Unformatted READ and WRITE statements) and the information on the tape is not to be retained.

A formatted record may contain up to 161 characters.

An unformatted record may be any length. If necessary it will be split into several physical blocks.

The ENDFILE statement outputs an end-of-simple-file trailer label to indicate the end of a file.

The REWIND statement rewinds the tape and positions it ready to READ or WRITE the first record. If the previous statement referring to that tape was a WRITE statement, an ENDFILE operation is automatically carried out before the REWIND.

The BACKSPACE statement moves the tape backwards one record. If the previous statement referring to that tape was a WRITE statement, an ENDFILE operation is automatically carried out before the BACKSPACE. Since BACKSPACE moves the tape backwards over a record rather than a physical block it is a fairly complex operation compared with reading or writing a record.

An output operation effectively destroys any information later on the tape. It is therefore an error to attempt to READ a record after a WRITE operation and without the intervention of a REWIND or BACKSPACE.

Part 3

FORTRAN on 1900 series Computers

CONTENTS

Chapter 1

INTRODUCTION

The information in the earlier parts of this manual is largely concerned with the FORTRAN language as such. This part is solely concerned with the 1900 FORTRAN Compiling System.

1900 SERIES COMPUTERS 1.1

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 multi-programming, 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 to run the computer effectively. It controls the running of other programs and the multiprogramming activities of the larger machines: it also controls the peripherals and executes requests for a transfer of information between peripherals and the central processor. It communicates with, and executes orders given by, the operator. This communication is achieved through the console typewriter, which gives a permanent typed record of all communications in the sequence in which they appear. Not only can the operator type instructions to Executive, to load, activate and delete programs for example, but the Executive program can also use the typewriter to inform the operator of unusual situations within the machine.

For convenience above and later in this Part, a human operator has been assumed but the principles remain the same when an operating system is used.

1900 SERIES COMPILATIONS 1.2

FORTRAN and other 1900 series compilers do not translate directly into equivalent machine-code programs. Instead, each compiler produces from a source program one or more semi-compiled segments. These segments can then be combined by a special routine known as the consolidator to form a consolidated, semi-compiled program. Such a program is output (for example, to magnetic tape) along with a General Purpose Loader, the function of which is to load the program at run time and complete the translation into machine code.

This system is designed to permit segments of program written in different languages or compiled at different times to be easily consolidated into a

single program. In many cases, however, the programmer need not concern himself with the system for he may write a complete program in one language, it will be translated and consolidated in a single computer run, and it will then be available for execution, either immediately or at a later date.

A separate consolidation run is essential only if the program includes segments written in different languages, which will be produced in semi-compiled form by the compiler for that language. However, if the programmer has chosen to divide his program into different sections to be compiled at different times, he can either arrange a separate consolidation run or consolidate during the last compiling run.

1900 FORTRAN COMPILERS 1.3

Several FORTRAN compilers are available for the 1900 series. They differ chiefly in the type of configuration for which they are intended. The compiling system for each compiler is basically the same but some of the statements described later in this Part might not be available on particular compilers. Details of the compiling system will be given with the specification for each compiler.

COMPILING SYSTEM STATEMENTS 1.4

The compilation and consolidation of a 1900 FORTRAN program is controlled by a series of Compiling System Statements. Each statement is written in columns 7 to 72 of a line. Spaces are ignored (except in file names). The statements are therefore very similar to FORTRAN statements, except that these statements must not have continuation lines.

Some of the Compiling System statements cause the program to be compiled in some specific way, that is, they are effective during compilation. These are written between segments.

Other statements are chiefly concerned with the object program and appear in a Program Description segment supplied ahead of the program segments.

However, there is no rigid distinction between the two types of statement, and some statements can appear either in the Program Description or between segments.

1900 Algol and 1900 EMA have a similar series of statements.

Chapter 2

COMPILER INPUT

OVERALL STRUCTURE 2.1

The input to the 1900 FORTRAN compiler on one compiling run consists of the following items, in the order shown:

Listing statement (optional)

SEND TO statement (optional for some compilers)

Program Description Segment

Source and semi-compiled segments

FINISH statement

Listing statements and the SEND TO statement are described in Chapters 4 and 3 respectively. The other items are described below.

PROGRAM DESCRIPTION SEGMENT 2.2

Program Description for a Complete Program 2.2.1

The Program Description segment is a collection of Compiling System statements whose chief purpose is to give information about the object program, such as what peripherals it uses, its name, etc. When a complete program, which may comprise source and/or semi-compiled segments, is to be compiled and consolidated into a loadable program, the first statement of the Program Description is a PROGRAM statement. It has the form

PROGRAM (n c)

where n is the name by which the object program is to be known to Executive and c is an accounting code. n obeys the rules for names given in 1.2 of Part 1, except that it must contain exactly four characters; additional rules for program names may be imposed at particular 1900 installations.

c consists of up to eight letters and digits. It may be omitted except in programs to be compiled and/or run at certain installations, where it ensures that the correct customer or department is charged for the use of the computer. The Program Description terminates with an END statement in the same way as other segments.

Between the PROGRAM and END statements may appear a number of Compiling System statements in any order. The most common such statements are the INPUT, OUTPUT,

USE and CREATE statements described in 5.1 of Part 2. Other statements are described in appropriate chapters of Part 3.

EXAMPLES

```
PROGRAM (SURV)
INPUT 1=TRO
OUTPUT 2=LPO
END

PROGRAM (TRY2 SCI123A3)
PRIORITY 70
INPUT 1,100,(MONITOR)=CRO
INPUT 2=CR1
OUTPUT 4,(MONITOR)=LPO
OUTPUT 3=CPO
TRACE
END
```

The SEGMENTS Statement 2.2.2

When one or more segments are to be translated into semi-compiled form for future consolidation with other segments into a complete loadable program, a different form of Program Description is used. Instead of a PROGRAM statement, the first statement is a SEGMENTS statement of the form

```
SEGMENTS
```

or

```
SEGMENTS (n c)
```

where n and c have the same form as for PROGRAM. As before, c is an accounting code and is needed only at certain installations. The name, n, is not now the name of a program but merely serves to distinguish this compilation from any others in the same series.

An END statement terminates the Program Description.

Usually, the only statements within this type of Program Description are those that affect the mode of compilation, e.g. TRACE. INPUT, OUTPUT, USE and CREATE statements are not usually given at this stage, since they are ignored in favour of those under the PROGRAM statement if the segments are later consolidated with others using the FORTRAN compiler. The statements are necessary (assuming FORTRAN input/output operations are used) if the segments are to be consolidated in any other way.

EXAMPLES

```
SEGMENTS
TRACE
END

SEGMENTS
END
```

ACCEPTABLE SEGMENTS 2.3

The compiler accepts as input segments written in FORTRAN and also semi-compiled segments. The latter may have been produced from FORTRAN segments during a previous compilation or may have been produced by other 1900 compilers, e.g. PLAN.

Source and semi-compiled segments may be interspersed with each other. If the program is to be consolidated there must be one and only one MASTER segment (in either source or semi-compiled form).

If two or more segments have the same name all but the first one are ignored. In this connection, a BLOCK DATA segment is thought of as having the special type of name

n%

where n is the name of the first common block in the first COMMON statement of the segment.

A consolidated (loadable) program is simply a series of semi-compiled segments with other information attached (3.1.1) and may be treated as such. That is, it may be re-input to the compiler and re-consolidated with other new segments. Along with the rule of the previous paragraph, this gives a method for amending a program by the replacing of segments by others of the same name.

Also, a program may be re-consolidated as above with a new Program Description defining a different set of peripherals to be used.

THE FINISH STATEMENT 2.4

The FINISH Statement indicates the end of the program. It has the form

FINISH

and appears after the last segment supplied by the programmer.

INPUT MEDIA 2.5

Programs are initially prepared on paper tape or punched cards. They may be input to the compiler either in this form or indirectly via a medium such as magnetic tape.

Paper Tape 2.5.1

Programs may be punched on eight-track paper tape using the standard 1900 codes (see Appendix 3).

Any 'line' that is shorter than the standard FORTRAN length of 72 characters can be regarded as being lengthened by the compiler up to this limit by the addition of space characters on the right. As spaces are not normally significant, this has no apparent effect except when a TEXT constant or H field descriptor overlaps onto a continuation line.

The Horizontal Tab character is interpreted as one or more spaces according to its position on the line.

A tab in columns 1 to 6 will cause advance to column 7; a tab in columns 7 to 12 causes advance to column 13; and so on. The complete series of tab stops is columns

7, 13, 16, 36, 70, 73, 76.

Most advantage is gained from this system if statement labels are written starting in column 1, since it is then possible to 'tab' after the label to the start of the statement in column 7.

Punched Cards 2.5.2

Programs may be punched on standard 80 column cards. The standard 1900 series card code is used (see Appendix 3).

The first 72 columns of a card are a line as defined in 1.3.1. The last 8 columns are ignored by the compiler except for listing purposes and may be used, for example, for card sequencing.

Magnetic Tape 2.5.3

Source program is originally prepared on paper tape or cards but may be copied to a magnetic tape file before being input to the compiler. Semi-compiled segments also may be input from magnetic tape, having normally been output to magnetic tape during a previous compilation.

Two chief types of magnetic tape file are standard on 1900 series computers: composite files, which contain subfiles, and simple files, which do not. This section considers only those aspects of file structure that affect a FORTRAN programmer. For FORTRAN purposes each type of file occupies one reel of magnetic tape only.

A composite file is split into subfiles. Each subfile contains either FORTRAN source program or semi-compiled program or further subfiles. These subfiles may also contain subfiles. Nesting of subfiles to any depth is permitted.

A source program subfile contains one or more segments and Compiling System statements. A semi-compiled subfile contains one or more segments.

Each file and each subfile is identified by a name consisting of up to 12 characters chosen from the set

A to Z
0 to 9
Space
Minus sign (hyphen)

The first character must be a letter. All subfiles in a file should have distinct names.

Different versions of the same named file are distinguished by the addition of an integer generation number in parentheses after the file name, e.g. MYSOURCEFILE(10).

The method of selecting named subfiles for compilation is described in 2.6.

The only simple files allowed as input to the FORTRAN compiler are those containing n consolidated programs to be re-consolidated as described in 2.3. These will normally have been produced by the FORTRAN compiler (Chapter 3) and will have names of the form

PROGRAM name

where 'name' is the name of the program and is preceded by one space.

THE READ FROM STATEMENT 2.6

A program need not be prepared solely on one type of input medium. The compiler may be made to switch from one input medium to another by Compiling System statements of the form

READ FROM (p)

or

READ FROM (p, a)

where p is a two-letter code for the type of peripheral from which input is to continue and a is identifying information needed for some media (e.g. a file name).

Input normally starts from paper tape or punched cards and continues as specified by READ FROM statements. Such statements may occur at any point outside a segment. They are ignored if encountered anywhere except on paper tape or cards.

Switching between Paper Tape and Cards 2.6.1

If input is taking place from paper tape and the statement

READ FROM (CR)

is encountered outside a segment, the paper-tape reader is released from the compiler and a card reader assigned instead. Input then continues from that reader. The converse operation is initiated by a statement of the form

```
READ FROM (TR)
```

EXAMPLE

```
LIST (LP)                          }
SEND TO (MT)                       }
PROGRAM (TEST)                     }
INPUT 1=TR0                        }   Prepared on paper tape
OUTPUT 2=LP0                       }
END                                }
READ FROM (CR)                     }

MASTER TRY3                        }
....                               }
....                               }   Prepared on punched cards
END                                }
READ FROM (TR)                     }

SUBROUTINE MAYBE(K , A)            }
....                               }
END                                }   Prepared on paper tape
FINISH                             }
```

Compiling from Magnetic Tape 2.6.2

A named subfile may be selected for compilation from magnetic tape by a READ FROM statement on paper tape or cards with one of the forms

```
    READ FROM (MT , f.s)
or
    READ FROM (MT , .s)
```

where f and s are file and subfile names with the form described in 2.5.3. f may or may not contain a generation number.

Before a magnetic tape is used it must be opened. When it is no longer needed it is closed.

When f.s is found in a READ FROM statement, any input magnetic tape file that is currently opened is rewound and closed. Then the file with name f is opened. If the generation number is absent or zero, any file with that name is accepted, otherwise the file must have the specified generation number. The file is scanned for the subfile s. When found, its contents are compiled. If s contains nested subfiles the contents of all the deepest subfiles in the nest are compiled. The file is left positioned after s. The card reader or paper tape

reader containing the READ FROM statement will have been retained throughout the whole of the above operation and input now continues from this reader as usual, until another READ FROM is encountered.

If f is omitted from the READ FROM statement, the action is an above except that the search starts from the current subfile of whatever file is already opened.

Searching takes place only in the forward direction. If a subfile earlier than the current one is required, the user must write a READ FROM statement containing the name of the current file. The file then rewound, closed and immediately re-opened.

A complete simple file may be input by a statement of the form

```
READ FROM (MT , f)
```

where f is the file name.

EXAMPLES

1.
```
LIST (LP)
SEND TO (MT)
READ FROM (MT , PROGFILE.PROG3)
```

This example reads a complete program, including Program Description and FINISH statement, from a magnetic tape file and compiles it to another magnetic tape.

2.
```
SHORT LIST (TP)
SEND TO (MT , SEGSFILE.PART1)
SEGMENTS
TRACE
END
READ FROM (MT , SOURCEFILE.PART1)
FINISH
```

This example compiles some segments in TRACE mode form from a subfile of one file onto a subfile of another file.

3.
```
SHORT LIST (LP)
SEND TO (MT)
PROGRAM (SURV 1211784)
PRIORITY 70
INPUT 1,(MONITOR)=TRO
OUTPUT 2,(MONITOR)=LPO
END
```

```
TRACE
SUBROUTINE SOON (X)
....
END
NO TRACE
READ FROM (MT , ICTAB1(9).SF10)
READ FROM (MT , .SF20)
TRACE
READ FROM (MT , ICTAB1(9).SF2)
READ FROM (MT , ICTZZ3.SSF5)
FINISH
```

This example compiles and consolidates a complete program onto a magnetic tape, taking one segment, called SOON, from either paper tape or cards and other segments from files ICTAB(10) and ICTZZ3.

4.

```
SHORT LIST (LP)
SEND TO (MT)
PROGRAM (TRY3)
INPUT 1,(MONITOR)=CR0
OUTPUT 3,(MONITOR)=LP0
END
READ FROM (MT , PROGRAM TRY2)
FINISH
```

This example re-consolidates a program called TRY2, giving it a new Program Description and re-naming it TRY3. New segments could have been inserted between the END and READ FROM statements.

COMPILER OUTPUT

FORM OF OUTPUT 3.1

Loadable Program 3.1.1

If the Program Description starts with a PROGRAM statement, and if there are no errors in the program, the compiler produces a loadable program, that is, a program that can be loaded into the computer and executed.

The program consists of the following:

1. Request Block

   This block indicates to Executive the name and priority of the program and the amount of store it needs.

2. General Purpose Loader (GPL)

   The GPL is input by Executive and then inputs the remainder of the program.

3. Consolidated Leader

   The consolidated leader contains information for the GPL about the following semi-compiled segments.

4. Semi-Compiled Segments

   The order in which quantities 1 to 4 are output depends upon the version of the compiler.

Unconsolidated Semi-Compiled Segments 3.1.2

If the Program Description starts with a SEGMENTS statement, the compiler produces semi-compiled segments but does not consolidate them into a loadable program. These segments may be re-input to the compiler along with other semi - compiled and source segments, and consolidated into a complete program on a later run. Alternatively, they may be consolidated by a separate Consolidator program.

The SEND TO Statement 3.2.1

The SEND TO statement indicates what type of output medium is to be used for the semi-compiled program. It has the forms

```
SEND TO (p , a)
SEND TO (p)
```

where p is a two-letter code indicating the output medium and a is information needed for some media to allow the output to be completely identified (such as a file name on magnetic tape).

Versions of the compiler that always output to the same fixed media may ignore the statement.

Magnetic Tape 3.2.2

The two-letter code for magnetic tape is MT. The effect of the SEND TO statement must be considered in conjunction with the PROGRAM or SEGMENTS statement.

If the Program Description starts with a PROGRAM statement, the simple form SEND TO statement is used, i.e.

```
SEND TO (MT)
```

Then the program is output as a simple file to a scratch tape (a tape with no useful information already on it) which is given the new file name

```
PROGRAM name
```

where 'name' is the four-character name in the PROGRAM statement, preceded by a space. The program may then be loaded from this tape.

EXAMPLE

```
SEND TO (MT)
PROGRAM (TEST)
```

A magnetic tape called PROGRAM TEST is created.

If the Program Description starts with a SEGMENTS statement the SEND TO statement takes the form

```
    SEND TO (MT , f.s)
or
    SEND TO (MT , f(g).s)
```

where f and s are a file and subfile name as described in 2.5.3. f may or may not include a generation number.

The effect of this statement is to output a new subfile, s, of semi-compiled program on the end of an already existing file identified by f. If the generation number of f is absent or zero, any tape with the right name is used, regardless of its actual generation number.

EXAMPLES

```
SEND TO (MT , MYFILE(3).TRY3)
SEGMENTS

SEND TO (MT , CUSTOMERFILE.ICTSECTION3)
SEGMENTS (PAR3 9173AB12)
```

Paper Tape 3.2.3

The two-letter code for paper tape output is TP. The SEND TO statement therefore has the form

```
SEND TO (TP)
```

Punched Cards 3.2.4

The two-letter code for punched card output is CP. The SEND TO statement therefore has the form

```
SEND TO (CP)
```

THE LISTING STATEMENTS 4.1

During compilation of a source program the compiler outputs a partial or complete list of source program statements, together with other information about the program and details of any errors found. The mode of listing required is specified by one of the statements

```
LIST (p)
SHORT LIST (p)
```

where p is a two-letter code indicating the type of peripheral on which the list is to be output.

The LIST statement gives a full listing of source program statements, comments and compiling system statements. Semi-compiled segments are listed by name only.

The SHORT LIST statement gives a listing of compiling system statements, segment headings (i.e. MASTER, SUBROUTINE, FUNCTION and BLOCK DATA statements) and semi-compiled segment names.

The same diagnostic information is output with both statements (see Chapter 5).

A listing statement is normally the first statement read by the compiler. If it is omitted a standard listing mode and peripheral is assigned by default, depending on the compiler.

The mode of listing may be changed by the insertion of a listing statement between segments. The type of listing peripheral should not be changed.

EXAMPLE

```
LIST (LP)
SEND TO (MT)
PROGRAM (TEST)
OUTPUT 1=LPO
END
MASTER TESTA
....
....
END
SHORT LIST (LP)
SUBROUTINE MATMPY
....
....
END
FINISH
```

Line Printer 4.2.1

The two-letter code for a line printer is LP. Most versions of the compiler assume SHORT LIST (LP) in the absence of a listing statement.

Paper Tape 4.2.2

The two-letter code for a paper tape punch is TP. Only SHORT LIST may be specified.

Chapter 5

DIAGNOSTICS

This chapter describes the various aids provided to diagnose errors in a 1900 FORTRAN program.

ERRORS FOUND DURING COMPILATION 5.1

During the testing phase of a new program errors are found most easily if all statements are listed on each run, i.e. LIST rather than SHORT LIST should be specified (see 4.1). When the compiler finds an error in a statement it ignores the remainder of the statement, outputs an error message on the listing peripheral and continues searching for further errors until the FINISH statement is found. The format of the error messages is given in the appropriate compiler specification. The occurrence of one error may lead the compiler to list non-existent errors later in the program. Consider the following sequence

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

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 reasons for some errors are obvious they should be corrected and the program re-compiled; often, other apparent errors will then disappear.

The compiler translates segments independently of each other. Therefore. if two segments are not consistent with each other, the error is not normally found until the program is run.

If the program is to be consolidated, any segments that have been called but are not present are listed on the occurrence of a FINISH statement in the form

```
SEGMENTS NEEDED
      n1
      n2
      .
      .
      .
```

where each n is the name of a missing segment. Certain names starting with the character % may be listed if, because of an operator error, the correct FORTRAN compiler library has not been read. Consolidation is not completed unless the programmer expected this situation to occur and has requested the operator to

force consolidation. This action is sensible only if the programmer knows that the called segments are not, in fact, to be entered during program execution.

ERRORS FOUND WHEN RUNNING THE PROGRAM 5.2

Error Detection 5.2.1

The recommended method of finding the causes of the errors described in 5.2 is to compile the program in TRACE mode (see 5.3).

Overflow 5.2.2

Each segment has associated with it an overflow indicator which can be either clear or set. Overflow is made clear on initial entry to a segment. It is set whenever an arithmetic error occurs. Possible causes are an operation whose result is outside the permitted range for 1900 numbers, division by zero or evaluation of a standard function for undefined ranges of the argument (e.g. square root of a negative number). In all such cases the result of the operation is meaningless.

Overflow is also set after a RETURN from a SUBROUTINE or FUNCTION segment if it was set within that segment - or, of course, if it was set before entry to that segment.

The state of overflow at various points of the program may be signalled if the program is complied in TRACE mode. Alternatively it may be checked by a call of the subroutine OVERFL. This has the general form

```
CALL OVERFL (J)
```

where J is an INTEGER variable or array element. J is set to 1 if overflow is set in the current segment or to 2 otherwise. The subroutine also clears overflow if it was set.

Input/Output Errors 5.2.3

If an error is found during an input or output operation the program is halted and a message typed on the console typewriter. The possible messages are listed in the appropriate compiler specification.

Other Errors 5.2.4

Certain other errors may be found only with difficulty if the program is not compiled in TRACE mode. The most common such errors are:

1. An array subscript is outside the defined range for that array.
2. Actual and dummy arguments do not agree in number, order, kind and type.
3. The variable in a Computed GO TO statement is outside the appropriate range.
4. The variable in an Assigned GO TO statement has not been assigned a value by an ASSIGN statement.

5. A value of one type has been referred to as a value of a different type (by a misuse of EQUIVALENCE or COMMON statements).

These faults may cause incorrect results to be obtained or may cause the program to go out of control. In the latter case, the program often attempts to carry out some illegal operation and is stopped by the 1900 Executive, which then types a message including the word ILLEGAL on the console typewriter. Normally, the message is not of much assistance to a FORTRAN programmer because it contains information in machine-code rather than FORTRAN terms. Also, the real error probably occurred much earlier.

If the program was compiled in TRACE mode, error information is available to the programmer in FORTRAN terms and errors are much easier to find. It is recommended that all new programs be compiled initially in TRACE mode.

TRACE 5.3

Introduction to TRACE 5.3.1

The 1900 FORTRAN compilers have a TRACE mode of compilation which should be used for all new programs. Programs compiled in this mode may be tested in effectively source language terms. The increased speed of testing more than makes up for the slight increase in store and execution time for programs compiled in TRACE mode.

The most important feature of TRACE is that when a program stops because of an error it is possible to obtain a list of the source statements that were executed immediately prior to the error. If required, the programmer may also request that certain statements and their results be listed during the running of the program. He must then specify which statements are to be listed in a steering list supplied with the program.

If the whole program is to be compiled in TRACE mode the statement

```
TRACE
```

should appear in the Program Description. If only parts of the program are to be compiled in TRACE mode the TRACE and NO TRACE statements may be used between segments (see 5.3.5).

The unit on which trace information is to be output must be specified in the Program Description by writing (MONITOR) in place of a unit number in an OUTPUT statement.

EXAMPLES

```
OUTPUT (MONITOR) =LPO
OUTPUT 2, (MONITOR) = CP1
```

In the second example, trace output is to a card punch, which is also referred to within the program as unit 2.

If a steering list is to be read by a traced program (see 5.3.3) then (MONITOR) must also appear in an INPUT statement in the Program Description.

EXAMPLES

```
INPUT (MONITOR) =CR2
INPUT 1, (MONITOR),5=TRO
```

The simplest use of TRACE occurs when the whole program is compiled in TRACE mode and no steering list is supplied. When the program halts, either because of a PAUSE statement or because of an error stop, or because the program has gone out of control and been stopped by Executive, the operator may cause details of the last 100 source language statements that we have been executed to be listed on the MONITOR output unit. Each statement occupies a line and gives the following information:

A line number, running from -100 to -1

The statement label, if any

An abbreviation of the statement type

V if the overflow indicator is set

The result of the statement, if applicable.

If a change of segment occurs between statement lines, the name of the new segment is given on a line by itself.

The end of a DO loop, although not written as a statement in the source program, is treated as a statement (type LOOP) by TRACE and is "executed" each time around the loop.

The statements, their abbreviations and their results are as follows:

Statement	Abbreviation	Result
Arithmetic assignment	ARTH	Value of right hand side
Logical assignment	LOGC	Value of right hand side: TRUE or FALS
Arithmetic IF	IF	Value of expression in parenthesis
Logical IF	LIF	Value of logical expression: TRUE or FALS
GO TO	GO	
DO (	DO	m_1
(	LOOP	m_2-i
Computed GO TO	CGO	i
Assigned GO TO	AGO	
READ	READ	
WRITE	WRITE	
PAUSE	PAUS	
STOP	STOP	
CALL	CALL	
RETURN	RETN	
ENDFILE	ENDF	
CONTINUE	CONT	
BACKSPACE	BACK	
REWIND	REW	
ASSIGN	ASGN	

Information similar to that described in 5.3.2. can be obtained during the running of the program. The programmer must specify which parts of the program are to be traced in this way by preparing a steering list. This list is normally read from the MONITOR input unit by the object program before execution commences.

A steering list defines certain labelled statements to be "triggering" statements, and indicates the number of statements before or after each triggering statement that the programmer wishes to be traced. The list of triggering statements for each segment is prepared as follows:

n	segname		
			Heading line
l_1	d_1	c_1	Steering lines
l_2	d_2	c_2	
. . . .			
l_n	d_n	c_n	

After the list for the last segment appears a terminating line of the form:

0 END

n, the number of steering lines for the segment, is an integer and must be right justified in columns 1 to 6. 'segname' is the name of the segment; it must start in column 7 and must not contain spaces. l, d and c are integers (their use is described below) and must be separated by one or more spaces. 'END' in the terminating line must start in column 7 and contain no spaces.

No blank lines are allowed in the list. If the list is prepared on paper tape the horizontal tab character must not be used.

Each l in the list is the label of a triggering statement. It must not be listed more than once.

d, where $-99 \leq d \leq 2047$, indicates at which statement relative to the triggering statement tracing is to commence, e.g. if d is 50, trace information is output when 50 statements have been executed after the triggering statement. 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 $0 < c \leq 4095$ is the number of statements to be traced for that triggering statement.

The trace information for each triggering statement is of the form

1 segname

followed by c lines in the format given in 5.3.2. except that the line numbers may range from -99 to 4095, with the triggering statement itself as line 0. If there is an overlap between the ranges of statements 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.

Array Subscripts 5.3.4.

Array subscripts are checked if a program is compiled in TRACE mode. If an element is not within the defined bounds of its array the program is stopped and the line

ARRAY SUBSCRIPT ERROR

output on the MONITOR unit. The operator should then ensure that the last 100 statements are listed (see 5.3.2).

Partial Tracing 5.3.5.

If some segments of the program are to be compiled in TRACE mode and others not, the statements

TRACE

and

NO TRACE

should be used. They may appear either within the Program Description or between segments. TRACE indicates that the TRACE mode of compilation is to continue until the next NO TRACE statement or the FINISH statement. NO TRACE indicates that no following segments are to be compiled in TRACE mode until a TRACE statement is encountered. Neither statement has any effect on any semi-compiled segments that are input to the compiler.

If a segment is not compiled in TRACE mode no triggering statement may be specified within it, and no information about statements within the segment is ever output.

Chapter 6

OVERLAYS

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

INTRODUCTION 6.1

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 such as magnetic tape. That is, only part of the program is in the core store at a 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 will 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. In contains the MASTER segment and any other segments 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 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.

Unlike other versions of FORTRAN, 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. Thus, any segment may refer to any other segment, whether they are in the same area or not.

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.

THE OVERLAID PROGRAM 6.2

An overlay program consists of a unit of program in a permanent area and a number of overlay units in one or more overlay areas.

Each overlay unit contains one or more function or subroutine segments and is assigned to one overlay area. Each area is identified by a number. Each unit is identified by the number of its area and a unit number. Unit numbers are distinct for each area and the same unit cannot be assigned to more than one overlay area.

Non-common variables and arrays are overlaid if they are listed in DATA statements. They will therefore be reset to their original values each time the unit is copied from the backing store. Other variables and arrays are not overlaid.

A subroutine or function which is called from outside its own unit must not have more than 20 arguments. No subroutine or function whose name is listed in an EXTERNAL statement may be in an overlay unit.

THE OVERLAY STATEMENT 6.3

The structure of an overlaid program is defined by a series of OVERLAY statements in the program description. The form of the statement is

OVERLAY (a,u) seg1, seg2, seg3 ..... segn

where a is an overlay area number; u is an overlay unit number; seg1, seg2, ... segn are the names of segments to be included in unit u of area a. a is an integer in the range 1 to 255 and u is an integer in the range 1 to 1023. The numbers chosen need not be consecutive.

OVERLAY is a compiling system statement so no continuation lines are permitted. It is not necessary for all the segments of an overlay unit to be named in the same OVERLAY statement. If the same area and unit numbers appear in more than one statement, the unit comprises 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
```

Any segments not named in an OVERLAY statement will be assumed to be in the permanent area.

The DEPTH OF OVERLAY Statement 6.4

Calls from one unit to another may be nested to any depth the programmer requires. To control this nesting the compiler reserves words of storage to form a list. Each time a call is made from one unit to another, the list will be extended by one entry. Each time a RETURN from one unit to another is executed,

one entry will be deleted from the list. 'Unit' here means permanent or overlay unit. Thus, a call from the permanent unit to an overlay unit or a call from an overlay unit to the permanent unit (for instance to a standard function or subroutine) each adds one entry to the list.

The space reserved by the compiler to contain this list must be sufficient to contain the maximum number of entries required. Normally the compiler will reserve enough storage for one entry for each overlay unit. However, if this number is insufficient, the programmer must indicate the maximum depth of overlay he requires with a DEPTH OF OVERLAY statement in the Program Description. The statement may also be used to save storage if the maximum depth required is less than the number of overlay units. The statement takes the form

DEPTH OF OVERLAY n

where n is an integer indicating the greatest depth (starting at depth 1) required. The statement will reserve storage for n entries in the list.

EXAMPLES

1 Suppose a program contained only two overlay units but these were nested to depth 20 (for example, a segment in permanent calls segment 1 of unit A, which calls segment 1 of unit B, which in turn calls segment 2 of unit A, etc.).

The programmer must then write

DEPTH OF OVERLAY 20

in order to reserve storage for the list.

2 If the program contained 40 overlay units, but the greatest depth to which a unit is called if 2 (i.e. 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 permanent) then the programmer may save the space of 38 entries by writing:

DEPTH OF OVERLAY 2

COMPILATION 6.5

The structure of an overlaid 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 usually most convenient to compile the whole program from source. However, this is not strictly necessary for all segments. The following segments must be re-compiled from source:

Any segment that is to be in a different area and/or unit from before.

Any segment calling another segment that will now be in a different area and/or unit from before.

Any other segments may be presented in semi-compiled form.

Standard functions and subroutines will normally be in the permanent area but may be included in an overlay unit if the programmer chooses, subject to certain rules given below. However, consolidation time may be significantly increased.

Certain of the library functions are nested or grouped, and if the calling function is in an overlay unit then the functions it calls must be either in the same overlay unit or in permanent store. These functions are listed below.

The functions ALOG and EXP must not be in an overlay unit.

If exponentiation to a DOUBLE PRECISION power occurs in the program then the functions DEXP, DLOG and ANINT must not be in an overlay unit.

If any COMPLEX working occurs in the program, then the functions CABS, ATAN2, ATAN, SQRT, COS and SIN must not be in an overlay unit.

The following table lists those functions which call other functions.

STANDARD FUNCTION	STANDARD FUNCTIONS CALLED (must be in the same overlay unit or in permanent store)
IDINT	IFIX
AMOD	AINT
MAXO	AMAXO
MAX1	AMAX1
MINO	AMINO
MIN1	AMIN1
DEXP	ANINT
CEXP	EXP, SIN, COS
EXP10	EXP
CLOG	ALOG, ATAN2
ALOG10	ALOG
DLOG10	DLOG
ALOG2	ALOG
CSIN	CEXP, CABS, ATAN2, COS, SIN
COS	SIN
DCOS	DSIN
CCOS	CSIN
COT	TAN
SINH	EXP
COSH	EXP

STANDARD FUNCTION	STANDARD FUNCTIONS CALLED (must be in the same overlay unit or in permanent store)
TANH	EXP
COTH	EXP
CSQRT	SQRT
ACOS	ASIN
DATAN	DATAN2
ATAN2	ATAN
ACOT	ATAN2
ACOSH	ASINH
ATANH	ALOG
ACOTH	ALOG
DMOD	DABS, AINT
CABS	SQRT

The standard subroutine DVCHK calls the subroutine OVERFL.

MAGNETIC TAPE OVERLAYS 6.7

This section gives details of the overlay system when magnetic tape is the backing store chosen. A program to be overlaid from magnetic tape must not refer to the unit MT0.

Efficiency 6.7.1

At run time. all the overlay unite associated with a particular area will be grouped on the magnetic tape in ascending order of unit number. The grcups will be held in ascending order of area number. After an overlay unit has been obtained from the tape, the tape will remain in position at the end of that unit. The system works most efficiently if the next overlay unit required follows immediately.

For maximum efficiency at consolidation time, the section of permanent program should be presented first, followed by the overlay segments in the order in which they will be held on magnetic tape, i.e. in ascending order of unit number within ascending order of area number.

TABLE 1 INTRINSIC FUNCTIONS

Intrinsic Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Absolute Value	$\|a\|$	1	ABS	REAL	REAL
		1	IABS	INTEGER	INTEGER
		1	DABS	DOUBLE	DOUBLE
Truncation	Sign of a times largest integer $\leq \|a\|$	1	AINT	REAL	REAL
		1	INT	REAL	INTEGER
		1	IDINT	DOUBLE	INTEGER
Nearest Integer	Sign of a times nearest integer (0.5 = 1)	1	NINT*	REAL	INTEGER
		1	ANINT*	REAL	REAL
Remaindering** (see note below)	$a_1 \pmod{a_2}$	2	AMOD	REAL	REAL
		2	MOD	INTEGER	INTEGER
Choosing Largest Value	$\text{Max}(a_1,a_2,\ldots)$	≥ 2	AMAX0	INTEGER	REAL
		≥ 2	AMAX1	REAL	REAL
		≥ 2	MAX0	INTEGER	INTEGER
		≥ 2	MAX1	REAL	INTEGER
		≥ 2	DMAX1	DOUBLE	DOUBLE
Choosing Smallest Value	$\text{Min}(a_1,a_2,\ldots)$	≥ 2	AMIN0	INTEGER	REAL
		≥ 2	AMIN1	REAL	REAL
		≥ 2	MIN0	INTEGER	INTEGER
		≥ 2	MIN1	REAL	INTEGER
		≥ 2	DMIN1	DOUBLE	DOUBLE
Float	Conversion from integer to real	1	FLOAT	INTEGER	REAL
Fix	Conversion from real to integer	1	IFIX	REAL	INTEGER
Transfer of Sign	Sign of a_2 times $\|a_1\|$	2	SIGN	REAL	REAL
		2	ISIGN	INTEGER	INTEGER
		2	DSIGN	DOUBLE	DOUBLE

** The function MOD or AMOD (a_1,a_2) is defined as $a_1 - [a_1/a_2]a_2$, where $[x]$ is the largest integer whose magnitude does not exceed the magnitude of x and whose sign is the same as x.

* Indicates available in 1900 FORTRAN but not necessarily available in standard FORTRAN

Intrinsic Function	Definition	Number of Arguments	Symbolic Name	Type of	
				Argument	Function
Positive Difference	$a_1 - \text{Min}(a_1, a_2)$	2 2	DIM IDIM	REAL INTEGER	REAL INTEGER
Obtain Most Significant Part of Double Precision Argument		1	SNGL	DOUBLE	REAL
Obtain Real Part of Complex Argument		1	REAL	COMPLEX	REAL
Obtain Imaginary Part of Complex Argument		1	AIMAG	COMPLEX	REAL
Express Single Precision Argument in Double Precision Form		1	DBLE	REAL	DOUBLE
Express Two Real Arguments in Complex Form	$a_1 + a_2\sqrt{-1}$	2	CMPLX	REAL	COMPLEX
Obtain Conjugate of a Complex Argument		1	CONJG	COMPLEX	COMPLEX

STANDARD SUBROUTINES

The following abbreviations are used in the descriptions:

u	An INTEGER expression identifying a particular peripheral device
L	A LOGICAL variable
I	An INTEGER expression
J	An INTEGER variable

All the subroutines must be called by a CALL statement.

INPUT/OUTPUT SUBROUTINES

The following subrcutines are described in detail in Chapter 4 of Part 2.

RLEASE	(u)	Release the peripheral u for use by other programs
ALLOT	(u,L)	Allocate the peripheral u if it is available and set L to .TRUE. . If n is not available set L to .FALSE. .
RUNOUT	(u)	If, as is normal, u is a paper-tape punch, four inches of runout are output by a CALL of this subroutine. Action on other peripherals is described in Chapter 5 of Part 2.
DISENG	(u)	Disengage the peripheral u, and suspend the program if a further reference is made to that peripheral before the operator re-engages it.

HARDWARE SUBROUTINES

Earlier versions of the FORTRAN language had facilities to test switches and lights by means of Hardware IF statements. The following subroutines are provided to replace these statements.

Sense switches 1 to 8 are simulated by bits 1 to 8 of the 1900 series word 30. They may be switched on or off and tested either by the operator or by the following subroutines

SWON (I)	Switch on sense switch I
SWOFF (i)	Switch off sense switch I
SSWTCH (I , J)	Set J to 1 if I is on and to 2 if I is off. I is left unchanged.

Sense lights 1 to 8 are simulated by bits 1 to 8 of the 1900 series word 31. They may be switched on or off and tested either by the operator or by the following subroutines

SLITE (I)	Switch on sense light I if I is 1,2,3.....8. If I=0 then switch off all sense lights.
SLITET (I , J)	Set J to 1 if I is on and to 2 if it is off. I is always left off.

A further subroutine is provided to test the overflow indicator. Conditions that may set overflow are described in 5.2.2.

OVERFL (J)	Set J to 1 if overflow is set in the current segment or to 2 if it is not
DVCHK (J)	This has the same specification as OVERFL.

TABLE 2 BASIC EXTERNAL FUNCTIONS

Basic External Function	Definition	Number of Arguments	Symbolic Name	Type of Argument	Type of Function
Exponential	e^a	1	EXP	REAL	REAL
		1	DEXP	DOUBLE	DOUBLE
		1	CEXP	COMPLEX	COMPLEX
Exponential to base 10	10^a	1	EXP10*	REAL	REAL
Natural Logarithm	$\log_e(a)$	1	ALOG	REAL	REAL
		1	DLOG	DOUBLE	DOUBLE
		1	CLOG	COMPLEX	COMPLEX
Common Logarithm	$\log_{10}(a)$	1	ALOG10	REAL	REAL
		1	DLOG10	DOUBLE	DOUBLE
Base 2 Logarithm	$\log_2(a)$	1	ALOG2*	REAL	REAL
Trignometric Sine (radians)	sin(a)	1	SIN	REAL	REAL
		1	DSIN	DOUBLE	DOUBLE
		1	CSIN	COMPLEX	COMPLEX
Trigonometric Cosine (radians)	cos(a)	1	COS	REAL	REAL
		1	DCOS	DOUBLE	DOUBLE
		1	CCOS	COMPLEX	COMPLEX
Trigonometric Tangent (radians)	tan(a)	1	TAN*	REAL	REAL
Trigonometric Cotangent (radians)	cot(a)	1	COT*	REAL	REAL
Hyperbolic Sine	sinh(a)	1	SINH*	REAL	REAL
Hyperbolic Cosine	cosh(a)	1	COSH*	REAL	REAL
Hyperbolic Tangent	tanh(a)	1	TANH	REAL	REAL
Hyperbolic Cotangent	coth(a)	1	COTH*	REAL	REAL
Square Root	$(a)^{1/2}$	1	SQRT	REAL	REAL
		1	DSQRT	DOUBLE	DOUBLE
		1	CSQRT	COMPLEX	COMPLEX
Arcsine	$\sin^{-1}(a)$	1	ASIN*	REAL	REAL
Arc-cosine	$\cos^{-1}(a)$	1	ACOS*	REAL	REAL

* Indicates available in 1900 FORTRAN but not necessarily available in standard FORTRAN.

Basic External Function	Definition	Number of Arguments	Symbolic Name	Type of	
				Argument	Function
Arctangent	$\tan^{-1}(a)$	1	ATAN	REAL	REAL
		1	DATAN	DOUBLE	DOUBLE
(angles in radians)	$\tan^{-1}(a_1/a_2)$	2	ATAN2	REAL	REAL
		2	DATAN2	DOUBLE	DOUBLE
Arc-cotangent	$\cot^{-1}(a)$	1	ACOT*	REAL	REAL
Inverse Hyperbolic Sine	$\sinh^{-1}(a)$	1	ASINH*	REAL	REAL
Inverse Hyperbolic Cosine	$\cosh^{-1}(a)$	1	ACOSH*	REAL	REAL
Inverse Hyperbolic Tangent	$\tanh^{-1}(a)$	1	ATANH*	REAL	REAL
Inverse Hyperbolic Contangent	$\coth^{-1}(a)$	1	ACOTH*	REAL	REAL
Remaindering**	$a_1 \pmod{a_2}$	2	DMOD	DOUBLE	DOUBLE
Modulus		1	CABS	COMPLEX	REAL

** The function DMOD (a_1, a_2) is defined as $a_1 - [a_1/a_2]a_2$, where $[x]$ is the largest integer whose magnitude does not exceed the magnitude of x and whose sign is the same as the sign of x.

* Indicates available in 1900 FORTRAN but not necessarily available in standard FORTRAN.

The only significant difference between the two sets of standard functions is that the Intrinsic functions cannot appear in EXTERNAL statements and cannot be actual arguments of subroutines or functions.