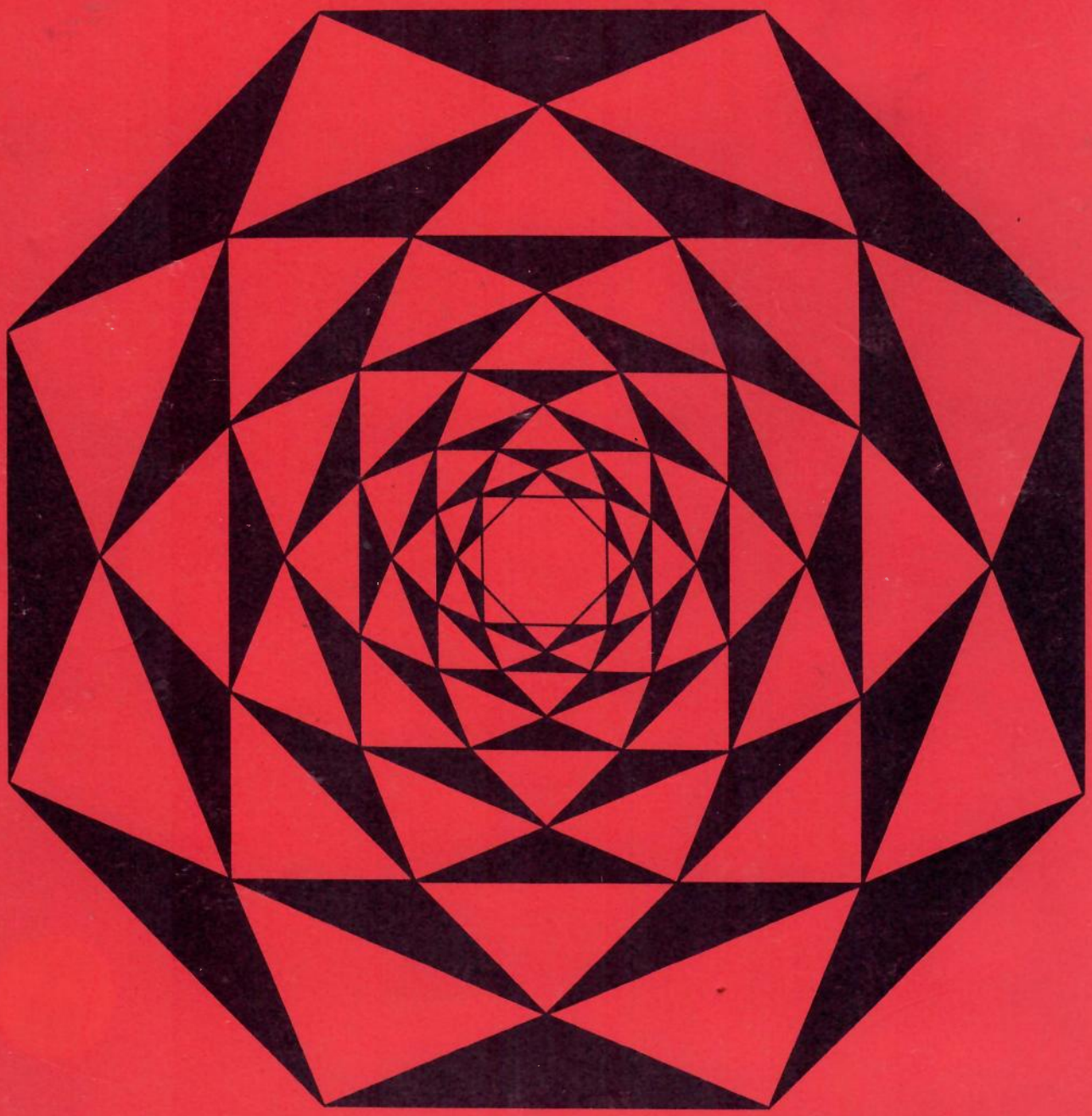
University of London
Atlas Computing Service

Fortran V



A manual of the **Atlas Fortran V Language**

by C. F. Schofield

University of London
Atlas Computing Service

44 Gordon Square, London WC1. Euston 3421 (01-387 3421)

May 1967

CONTENTS

CHAPTERS

*Sections marked with an asterisk contain information which is not normally necessary, but may be useful for specialised applications.

APPENDICES

CHAPTER 1 INTRODUCTION

Fortran is the most extensively used programming language in the world. Compilers exist for most machines - usually with variations in the language. Fortran is often considered to be an evolving language, with many dialects. Some of these dialects contain useful facilities which later become common to the language. The most common dialects are Fortran II, and, more recently Fortran IV.

Recently, the American Standards Association, (A.S.A) have proposed a standard subset of Fortran (see Ref.1). ASA Fortran is very similar to Fortran IV as implemented on many computers. Extensions to the language are explicitly allowed by the ASA report, so long as the ASA Fortran is included as a compatible subset.

The Fortran V compiler was developed by Atlas Computing Service in order to provide:-

(i) A subset compatible with ASA Fortran; and good compatibility with System/360 Fortran IV, and with Atlas Fortran (Hartran), whilst retaining as much compatibility as possible with Fortran II.

(ii) Efficient compilations using a reasonable amount of storage.

(iii) Good testing facilities for development work.

(iv) Extend the useful facilities of the Fortran Language. The useful extensions to the language include:-

Tracing (run-time testing) statements.
Block structure (nested subprograms).
Fully dynamic arrays.
The CLEAR statement for zeroing variables and arrays.
The OUTPUT statement for simple format free output.
"Format free" input of numbers.
Negative step, and real index for DO loops and I/O lists.
Improved control in FORMAT specifications (S, Y and Z controls etc).
The use of expressions, where only simple variables were normally allowed e.g. expressions may be used in I/O lists, as parameters of DO statements, and as subscripts, etc.
The use of in-line machine instructions.
The use of PUBLIC (name dependent) global variables.
Many other extensions will be found in the text.

Some of the facilities described above are also available in Atlas Fortran (Hartran). We are indebted to the Science Research Council, and in particular to Mr. E. B. Fossey, and Miss B. Stokoe of the Atlas Computer Laboratory at Harwell, for permission to use the Hartran system library including routines for dealing with input and output.

This publication is a definitive reference manual of the Fortran V language, and is not intended to be a Fortran primer; however, the presentation of material is such that no previous knowledge of Fortran is required.

CHAPTER 2 PUNCHING THE PROGRAM

2.1 CARD LAYOUT

Fortran V programs may be presented on 80 column cards, or 5 or 7-track paper tape. The rules for punching paper tape are given in Appendix 1. When punching cards, the following rules must be observed.

(i) Columns 1 - 5 of the first line of a statement may contain a label (statement number) to identify the statement.

(ii) Column 6 of the first line of a statement must be left blank or punched with a zero.

(iii) Columns 7 - 72 contain the actual Fortran statement. Blanks are ignored, except in TEXT constants, or in an S field in a FORMAT statement.

(iv) Columns 73 - 80 are not processed by Fortran V, and may be used for identification.

(v) Lines punched with the character C (or π) in column 1 are not processed by Fortran V, and may be used for comments.

(vi) If it is required to extend a statement over more than one line, then all lines except the first must have a standard Fortran character other than blank or zero in column 6. The statement is then continued in columns 7 - 72. Such lines are called continuation lines. Any one statement may not extend over more than 34 continuation lines (i.e. 35 cards in all).

(vii) Blank lines appearing between two statements will be ignored by Fortran V. Blank lines should not appear between a statement and its continuation.

2.2 CHARACTER SET

A program is written using the following standard characters:

Alphabetic: A, B, C, ..., Y, Z
Numeric: 0, 1, 2, ..., 8, 9

Special:

Character	Character Name
	Blank (space)
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left parenthesis
)	Right parenthesis

,	Comma
.	Point
π (or $)	Pi (or Dollar)
'	Apostrophe (or Prime)

Other special characters may be used, and these are described in Appendix 2.

An alphanumeric character is any alphabetic or numeric character.

2.3 CONDITIONAL COMPILATION (X CARDS)

Fortran V provides means for ignoring certain statements when compiling. All statements which contain an X in column one are treated as comments unless the TEST option is specified on the *FORTRAN directive of the routine (see section 12.2 (9)). If the TEST option is specified, then lines containing an X in column one are treated as normal statements, the X itself being treated as blank, and not as a label or part of a label.

This feature enables the user to include additional statements for testing purposes in his routines, and when testing is complete, the statements do not have to be removed from the program, since they can be turned into comments simply by removing the TEST option.

A normal statement may have continuation lines, which are X lines (see below). If an X line has continuation cards, then these cards should also be X cards.

Example:
```
X          PRINT  10
X  10      FORMAT ('TEST PRINT')

           PRINT 11, A, B
X      1             , C, D
   11      FORMAT (5X, F10.6)
```

CHAPTER 3 CONSTANTS

Seven types of constants are permitted in a Fortran V source program; integer, real, double precision, complex, text, logical and Boolean.

3.1 INTEGER CONSTANTS

Definition	±n
an integer constant is a string of one to eleven significant decimal digits written without a decimal point, and optionally preceded by a + or - sign.	

An integer constant occupies one word of storage

An integer constant must lie within the range:

$$2^{36} > n \geq -2^{36}$$

A string of more than 11 significant digits is set up as a double precision constant.

Examples
0
11
+75
032
-7746

3.2 REAL CONSTANTS

Definition	±n.n or ±n. or ± .n or ± n.nE ± n or ± n.E± n or ± .nE ± n or ± nE ± n
a Real constant may be: - (1) An optional + or - sign followed by one to eleven significant decimal digits, written with a decimal point, but without a decimal exponent. (2) An optional + or - sign followed by one to eleven significant decimal digits, written with or without a decimal point, and followed by a decimal exponent which is written as the letter E followed by an integer constant.	

A real constant occupies one word of storage.

The absolute value of a real constant must lie within the range

$$10^{-110} < x < 10^{105}, \text{ or be zero}$$

A real constant has precision to about eleven decimal digits.

A string containing more than eleven significant digits is set up as a double precision constant.

Examples: 1.0
1.
0.1
.1
-.73610003
3.E1 (means 3.0×10^{1} i.e., 30.0)
.2E-3 (means 0.2×10^{-3} i.e., 0.0002)
2 E+04 (means 2.0×10^{4} i.e., 20000.)

3.3 DOUBLE PRECISION CONSTANTS

Definition
a double precision constant is an optional + or - sign followed by a sequence of up to 22 decimal digits written with, or without, a decimal point, and followed by a decimal exponent, which is written as the letter D followed by an integer constant. In addition, real or integer constants containing more than 11 significant digits are set up as double precision constants.

A double precision constant occupies two words of storage.

The absolute value of a double precision constant must lie within the range

$$10^{-110} < d < 10^{105}, \text{ or be zero}$$

A double precision constant has precision to about 22 decimal digits.

Examples: 3D0 (means 3.0×10^{0} i.e., 3.0)
-.4D+2 (means -0.4×10^{2} i.e., -40.0)
3.141592653736

3.4 COMPLEX CONSTANTS

Definition	(real constant, real constant)
a complex constant is an ordered pair of real constants, separated by a comma, and enclosed in parentheses.	

A complex constant occupies two words of storage.

The first real constant represents the real part of the complex number; the second real constant represents the imaginary part of the complex number.

The parentheses are required regardless of the context in which the complex constant appears.

The magnitude and precision of each part of the complex constant obey the same rules as for real constants.

Examples: Fortran V representation	Complex number
(4.7,0.2)	4.7 + 0.2 i
(-8.3, -.01)	-8.3 - 0.01 i
(7.6E3,0.0)	7600. + 0.0 i

3.5 TEXT CONSTANTS

Definition	or 'character string' or nH character string

a Text constant is

(1) an unsigned integer constant (n) followed by the letter H followed by a string of n characters. This is a Hollerith constant.

or

(2) An apostrophe followed by a string of characters followed by a second apostrophe. This is a primed text constant.

A text constant occupies one or more words of storage, eight characters being stored in each word.

Spaces (Blanks) are significant characters in text constants.

Both types of constant are left-adjusted and filled out with blanks to an integral number of words, so that

2HXY is the same as 8HXYbbbbbb
and 'ABC' is the same as 'ABCbbbbb'

In the case of a primed text constant all characters (including spaces) between the apostrophes are taken as the text constant.
The apostrophe character itself may be used in a primed text constant by punching two apostrophes, e.g. 'DON''T' will be stored in the machine as DON'T.

When a text constant is stored, the surrounding primes, or the nH, are not stored with it.

Examples: 5HDON'T
'THISbISbAbPRIMEDbTEXTbCONSTANT'
11H(HOLLERITH)

Note: The primed text constant is preferable to the Hollerith constant, which should only be used in FORMAT statements.

3.6 LOGICAL CONSTANTS

Definition	.TRUE. or.FALSE.

A logical constant may take either of the following forms:

.TRUE.
.FALSE.

A logical constant occupies one word of storage.

A false constant is stored internally as a word whose least significant 24 bits are all zero.

A true constant is stored as a word whose least significant 24 bits are all ones.

*3.7 BOOLEAN CONSTANTS

Definition	n B octal digits or n O octal digits
A Boolean, or Octal constant is an unsigned integer constant, (n, which must not exceed 16) followed by a string of octal digits of length not exceeding n.	

A Boolean constant occupies one word of storage.

An octal digit is

0, 1, 2, 3, 4, 5, 6, or 7, but not 8 or 9.

Each octal digit is converted to three bits.

If the number (m) of octal digits is less than n, then n-m leading zeros are assumed.

If n is less than 16, then 16 - n following zeros are assumed.

Examples: Boolean Constant	Internal (Octal) word
1B0	0000000000000000
3B123	1230000000000000
8B123	0000012300000000
16B123	0000000000000123
16O123	0000000000000123

CHAPTER 4 VARIABLES AND ARRAYS

A variable is defined by its name and its type. There are seven types of variables: integer, real, double precision, complex, text, logical and Boolean.

4.1 VARIABLE NAMES

Definition
A variable name is a string of alphanumeric characters, the first of which is alphabetic. Only the first eight characters are significant. Any blanks embedded in the name are ignored.

Examples: I
L2557X
INNERSUM

Note that the following are equivalent to INNERSUM:

INNERbSUM
INNERbSUMMATION
INNERSUMP

4.2 ARRAYS

An array is a block of successive storage locations which can be referenced by a subscripted variable name (see 5.1.1). Arrays may have any number of dimensions. An array name has the same form as a variable name.

The array name and its dimensions must be declared before the name is referenced (see Appendix 3).

An array is composed of one or more elements; each element may be referenced by the array name followed by the appropriate subscript notation.

For Multi-dimensional arrays, Fortran V uses a special technique to access the element. Incorrect or incompatible results are likely if any subscript used is outside the range given by the dimensions. One-dimensional arrays (vectors) may be accessed out of range provided that the element accessed has had its position relative to the array defined by a COMMON or EQUIVALENCE statement (see sections 8.14 and 4.5).

For both single and multi-dimensional arrays, the (possible) error of exceeding array bounds (i.e. product of subscripts greater than product of dimensions declared) is automatically tested for in execution if the routine concerned is compiled in TEST mode (see section 12.2 (9)).

4.2.1. The DIMENSION statement

Note: see section 8.8 for details of adjustable dimensions

Definition	DIMENSION $dname_1$, $dname_2$, $dname_3$ ...
where $dname_1$, $dname_2$ ... are subscripted variable names, and the subscripts are unsigned integer constants, which represent the dimensions of the array(s) dname.	

The DIMENSION statement is not executable, and must precede the first reference to the array(s) being declared (see Appendix 3).

The DIMENSION statement may be used to declare the dimensions of any number of arrays.

An array may be declared as having any number of dimensions.

The number of storage words reserved for an array is equal to the product of its dimensions multiplied by a constant which is 2 for double precision or complex arrays and 1 for all other arrays.

The DIMENSION statement has no effect on the types of the arrays being declared.

Examples: DIMENSION ARRAY (20, 3, 4), B(2), I(1000, 3)
DIMENSION COSTS (1, 2, 3, 4, 5, 6, 7, 8, 9)

*4.2.2 Array storage

Arrays are stored by columns in ascending storage locations, so that the first subscript varies most rapidly, and the last least rapidly. e.g. the two-dimensional array A(m,n) is stored as follows:

$$A_{1,1}, A_{2,1}, A_{3,1} \cdots A_{m,1}, A_{1,2}, A_{2,2} \cdots A_{1,n}, A_{2,n} \cdots A_{m,n}$$

The second element of A is referred to as A(2,1), the third as A(3,1) and so on. Thus for the four-dimensional array X: X(I,J,K,L) refers to a storage location which is greater than X(1,1,1,1) by

$$C((I-1) + (J-1)d_1 + (K-1)d_1d_2 + (L-1)d_1d_2d_3),$$

where d_1, d_2, d_3 and d_4 are the dimensions of X. C is a constant which is 2 for double precision and complex arrays and 1 for all other arrays.

4.3 IMPLICIT TYPE ASSIGNMENT

The type of a variable name may be specified in two ways:

(i) Explicitly by a type statement (see section 4.4)
(ii) Implicitly by name

In the latter case, any variable names beginning with one of the characters I, J, K, L, M or N are assumed to be of type integer, and any other names are of type real.

Examples: I and J17 are integer names
P6 and ALPHA are real names

4.3.1 The IMPLICIT statement

The implicit type assignment described above may be modified by using the IMPLICIT statement.

Definition
IMPLICIT $type_1$ *n_1(ch_1, ch_2, ch_3-ch_4...)$type_2$(ch_5-ch_6)$type_3$*$n_2$$ch_7$...
Where ch_1, ch_2...ch_7...are any single alphabetic characters and $type_1$, $type_2$...are one of: INTEGER REAL DOUBLE PRECISION or DOUBLE LENGTH COMPLEX LOGICAL TEXT BOOLEAN and n_1, n_2... are 1, 2, 4, 8 or 16 The *n may be omitted, if present they have no meaning except that REAL *8 is taken as DOUBLE PRECISION. The parentheses may be omitted from the list of letters following the last type specified. An optional comma may follow each right parenthesis. The dash in ch_3-ch_4 etc. is a minus sign.

The IMPLICIT statement specifies that variable names beginning with certain designated letters, or ranges of letters, are of a certain type.

DOUBLE PRECISION, DOUBLE LENGTH and REAL *8 are synonymous.

The IMPLICIT statement does not affect the types of any names which have been declared before the appearance of the IMPLICIT statement. In particular dummy argument names are not affected by IMPLICIT.

If more than one IMPLICIT statement is given, then the later will override the earlier statements for any letters where the IMPLICIT statements conflict.

Examples:

a) IMPLICIT REAL I, J

Following this statement all new variable names beginning with I or J will be assumed to be of type real (unless overridden by a later type statement).
Names beginning with K, L, M or N will still be integer, and names beginning with A-H or O-Z will be real.

b) IMPLICIT TEXT M-Q

Following this statement all new variable names beginning with M, N, O, P or Q will be of type text. Names beginning with I, J, K or L will still be integer, and all other new names will be real.

c) IMPLICIT REAL (I, J) INTEGER (A-H, Z) BOOLEAN C, D

This statement has the effect:

Initial letter of new name	Implicit Type
A or B	Integer
C or D	Boolean
E - H	Integer
I or J	Real
K - N	Integer
O - Y	Real
Z	Integer

Note that the later appearance of BOOLEAN C, D overrides the previous declaration INTEGER A-H.

4.4 THE TYPE STATEMENTS

Note see section 8.8 for details of adjustable dimensions.

Definition
type *n $name_1$ (i_1, i_2..)/$data_1$/, $name_2$, $name_3$/$data_2$/ ....

Where

type is one of:

INTEGER
REAL
DOUBLE PRECISION or DOUBLE LENGTH
COMPLEX
LOGICAL
TEXT
BOOLEAN

and

$name_1$, $name_2$...are variable names, which may be subscripted. The subscripts must be unsigned integer constants.

and

n is 1, 2, 4, 8, or 16

The *n may be omitted. If present they have no meaning, except that REAL*8 is taken as DOUBLE PRECISION.

The /$data_1$/, $data_2$/...may be omitted. If present, then $data_1$, $data_2$....have the form:

i) A constant of the same type as the declaration
or ii) m* constant, where m is an unsigned integer
or iii) A series of the above two forms separated by commas.

4.4 cont

(1) The type statements are used to declare the types of variable names. If a variable name is declared in a type statement, then this overrides the type implicit in the first letter of the name.

(2) The type statements may be used to declare the dimensions of arrays. When subscripts appear in the list, the associated variable name is the name of an array and the subscripts are the dimensions of the array.

(3) The type statements are not executable, and must precede the first reference to the variable name(s) being declared. (see Appendix 3).

(4) The type statements may be used to assign initial data values to variables or arrays.
A list of constants of the form /m*c/ is equivalent to the list
/c,c,c.../ where c is written m times.

The constants in the list are loaded into the storage location(s) given by the preceding variable or array name. In the case of an array, the constants are loaded from left to right starting at the first location in the array. See also sections 4.6(3), and 4.6(9).

(5) When initial data values are to be assigned by means of a type statement, the type of constant stored is determined by the structure of the constant, rather than by the variable type (i.e. by the type of the type statement).

e.g. REAL X /3/

An integer 3 is stored in X, and not a real 3.0 as may be expected from the type of X. Care should thus be taken to ensure that the type of the constant(s) is the same as the type of the type statement.

(6) DOUBLE PRECISION, DOUBLE LENGTH, and REAL*8 are synonymous.

(7) If a variable name is declared in two or more type statements, then the first type holds until the second is read, the second holds until the third is read, and so on. Note, however, that the type of a variable cannot be changed once the variable has been referenced. (see Appendix 3).

Dimension information should be given once only, but the type of an array name may be declared in a type statement and its dimensions declared in a DIMENSION or COMMON or PUBLIC (see sections 8.13 and 8.14) statement. If this is done, and an array is to have values assigned in its type statement then its DIMENSION statement must come first, otherwise the order is immaterial.

(8) Variables which are given in an EQUIVALENCE statement and variables which are PUBLIC or in COMMON cannot be assigned initial data values by means of the type statements, except in a BLOCK DATA subprogram. (see section 8.15).

(9) Function names may be declared in type statements, but they must not have initial data values assigned to them.

Examples:

a) REAL I, JARRAY (2,5)

The name I now represents a real variable, and JARRAY is a real array of length 10 words. Note that JARRAY could also be declared as follows;

```
DIMENSION JARRAY (2,5)
REAL JARRAY
or
REAL JARRAY
DIMENSION JARRAY (2,5)
```

Since the DIMENSION statement is redundant in such cases, the single declaration

REAL JARRAY (2,5)

is preferred. Note that:

REAL JARRAY (2,5)
DIMENSION JARRAY (2,5)

would be in error.

b) INTEGER X/0/,Y (10), Z(2, 3)/3*0, 2*1, 3/

The name X now represents an integer whose initial value is zero. The name Y represents an integer array of length 10 words, whose initial values are undefined. The name Z represents an integer array of length 6 words whose initial values are:

Z(1, 1)=0, Z(2, 1)=0, Z(1,2)=0, Z(2,2)=1, Z(1,3)=1, Z(2,3)=3

4.5 THE EQUIVALENCE STATEMENT

Definition
EQUIVALENCE ($name_1$, $name_2$, ...), ($name_3$, $name_4$, $name_5$...)..
Where $name_1$, $name_2$... are variable names which may be subscripted. Any subscripts must be unsigned integer constants.

(1) The EQUIVALENCE statement specifies that variables with different names are to share the same storage location(s). Each pair of parentheses in the list encloses the names of two or more different variables (or array elements) which are to be stored in the same location. Any number of equivalences (sets of parentheses) may be specified.

(2) The EQUIVALENCE statement is not used to declare the dimensions of arrays. When subscripts appear in the list these indicate that particular element of an array which is to be made equivalent to the other items within the parentheses surrounding those names.

(3) The dimensions of any arrays specified must be declared before the EQUIVALENCE statement appears.

If the number of subscripts appended to an array name is less than the number of dimensions which have been declared for it, then the missing subscripts are taken to be 1. The number of subscripts must not be greater than the number of dimensions declared (see section 5.1.1).

(4) The EQUIVALENCE statement is not executable.

(5) No two variables which have been referenced may be made equivalent to each other (see Appendix 3).

(6) Dummy arguments, and COMMON or PUBLIC variables may be equivalenced to variables which have not been referenced.

(7) For each equivalence (set of parentheses) there is always one variable name to which the others are made equivalent. If one of the variable names has been referenced, then that name is the name to which the others are made equivalent. If none of the variables have been referenced, then the variable in the list which occupies the most storage is the one to which the others are made equivalent. (e.g. the longest array).

(8) The EQUIVALENCE statement must not contradict itself, or any previously established equivalence. No two elements of any one array may be made equivalent to each other. Adjustable arrays may not be equivalenced.

(9) In Fortran V, the EQUIVALENCE statement does not re-order COMMON. See section 8.13.2.

(10) There is no restriction on the types of variables which may be made equivalent; however, errors may occur in execution if apparently real variables in fact contain integer values:

```
Example:  EQUIVALENCE (I, X)
          I = 3
          Y = 3.0
          Z = Y/X
```

would cause an execution error, because X contains an integer value (I = 3).

This also applies to other mixtures of types which are made equivalent by means of EQUIVALENCE or COMMON statements. (see section 8.13.2).

(11) Variables which have been assigned initial data values by means of a type statement, should not appear in any EQUIVALENCE statement.

Examples: Assume REAL A(5), B(10), C(10, 2)

a) EQUIVALENCE (A, B, C(5, 1), X)

A(1), B(1), C(5, 1) and X now refer to the same storage location. Note that: A(2), B(2) and C(6, 1) will also be equivalent, and so on.

b) EQUIVALENCE (A(2), C(4)), (B(4), C(5, 1))

A(2) and C(4, 1) now refer to the same storage location similarly with B(4) and C(5, 1). This means that A(1), C(3, 1) and B(2) all share the same location.

c) EQUIVALENCE (C(5, 1), B, C)

This is illegal, because C(1, 1) and C(5, 1) cannot be made equivalent.

d) EQUIVALENCE (A, B(2), C(4, 2)), (A(3), B(4))

This is illegal, because the first equivalence establishes that A(1) is equivalent to B(2) and hence A(3) is equivalent to B(5); and the second equivalence is not permitted to contradict this.

e) Given COMPLEX CP(10)
EQUIVALENCE (A, CP)

A(1) and CP(1) are now equivalent. Since each element of CP occupies two words, A(2) and the imaginary part of CP(1) are equivalent. Similarly, A(3) and the real part of CP(2) are equivalent, and so on.

4.6 THE DATA STATEMENT

Initial values of variables (data) may be compiled into the object program by means of the type statements. An alternative and more flexible way of doing this is provided by the DATA statement.

4.6
cont

Definition	DATA $list_1/data_1/, list_2/data_2/, \ldots$

Where
$list_1$, $list_2$... are input lists with the restrictions given in (1), below.
and
$data_1$, $data_2$...are of the form:

i) a constant (of any type)
or ii) m * constant where m is an unsigned integer
or iii) a series of the above two forms separated by commas. The commas following the slashes may be omitted.

(1) The input list is defined in section 7.4.
The DATA statement variable list is of the same form as the input list with the following restrictions:-

(i) Implied DO parameters must be unsigned integers.
(ii) If a subscripted variable appears in the list, then the subscripts must be integers or variable names.
(iii) If any subscript is a variable name, then this name must be under the control of (i.e. used as the index of) a current implied DO.

(2) The DATA statement is used to assign initial data values to the variables or arrays which appear in its list.

A list of constants of the form /m*c/ is equivalent to the list /C, C, C, .../, where C is written m times.

The constants are loaded, from left to right, into the list of variable names, any implied DO loops being taken into account. If a non-subscripted array name appears in the list, then the whole of the array will be loaded with the constants. This method i.e. the Short List, is preferred when data is to be compiled into an entire array.

(3) There should be a one to one correspondence between the variables in the list and the constants. Each constant corresponds to one undimensioned variable or subscripted array reference. Note If it is desired to define a text constant of (say) 20 characters starting at B(1), then B must be dimensioned to cover at least three locations. If the constant is 'ISbTWENTYbCHARACTERS', then this is one constant which corresponds to three variables; B(1), B(2) and B(3). To this extent the one to one correspondence rule is modified.

(4) All variables referred to in the list must have their properties (i.e. types etc.) declared before the DATA statement appears.

(5) The DATA statement is not executable and does not define the value of any implied DO index present in the variable list.

(6) The type of the constant stored is determined by the structure of the constant itself rather than by the variable type in the statement,

e.g. DATA X/3/ (X is real)

An integer 3 is stored in X, and not a real 3.0 as may be expected from the type of X. Care should thus be taken to ensure that the type of the constant is the same as the type of the variable into which it is to be loaded.

(7) The DATA statement only initialises the values of the variables. DATA defined variables that are redefined during execution of the program will assume their new values regardless of the DATA statement.

4.6 cont

(8) The DATA statement cannot be used to enter data into blank COMMON. The BLOCK DATA subprogram may be used to compile data into labelled COMMON or PUBLIC (see section 8.15).

(9) If a variable is present in the list of more than one DATA statement, or is given more than once in the list of one DATA statement, then only the last value assigned to the variable will be effective, and all of the previous values are lost. Note also that the DATA statement may override any values assigned by a previous type statement (or vice versa).

Examples:

a)
```
DATA X,I,Q/1.0, 1, 1.0/
```

b)
```
DIMENSION A(10,10)
DATA A/100*1.0/
```

every element of A is set to 1.0. This form is preferable to:
```
DATA((A(I,J),I=1,10), J=1,10)/100*1.0/
```

c)
```
DIMENSION A(10), B(20)
DATA A, (B(I), I=1,20,2)/3.6,10*4.8,8*6.0,.0/
```

A (1) is set to 3.6, A(2) to A(10) are set to 4.8, B(1) is set to 4.8, B(3), B(5)...B(17) are set to 6.0 and B(19) is set to 0.0. The even numbered elements of B are not defined.

d)
```
COMPLEX C
TEXT A, B(2), A1
DATA C, A, A1,B/(1.0,0.0), 'XYZbbbbbABC', 'P', 'Q'/
```

C is set to 1.0 + 0.0i, A is set to 'XYZbbbbb', A1 is set to 'ABC', B(1) is set to 'P', B(2) is set to 'Q'.

CHAPTER 5 EXPRESSIONS AND REPLACEMENT STATEMENTS

Fortran V accepts two types of expressions: arithmetic and logical. These expressions form integral parts of Fortran V statements.

5.1 ARITHMETIC EXPRESSIONS

An arithmetic expression is a sequence of operators, operands and parentheses, assembled according to the rules given below.

An operator is: -

+	meaning	addition
-	"	subtraction
*	"	multiplication
/	"	division
**	"	exponentiation

An operand is: -

a constant
a simple variable
a subscripted variable
a function reference (see section 8.4)

Rules: -

(i) Any operand is an expression.

(ii) If X is an expression, then (X) is also an expression.

(iii) If X and Y are both expressions then the following are also expressions:

X + Y X - Y
X * Y X / Y
X**Y and -X (but see below)

Anything which results from repeated applications of these rules is also an expression. For example, rule (ii) implies that ((X)) and (((X))) etc. are expressions.

(iv) The sequence "operator operator" is not permitted.

e.g. A* - B must be written as A* (-B).

Examples of arithmetic expressions:

```
A
I - 76
(A)
4.3
-B -7.3 + X - (A+4.0)*6.3
```

5.1.1 Subscripted variables

In Fortran, subscripts may be written following the name of an array in order to access a particular element of the array. The subscripts must be enclosed in parentheses.

In Fortran V a subscript is any arithmetic expression.

If a subscript is not of type integer, then it will be truncated according to the kind of truncation in force when the statement is compiled. (see section 5.1.3).

An array must not be referred to as having more subscripts than the number of dimensions in its array declaration, so that if A is declared as DIMENSION A(12, 10), then any reference to A(I, J, K) would be illegal.

If an array is referred to as having less subscripts than dimensions, then the missing subscripts are assumed to be 1: So that after DIMENSION ARRAY (2, 3, 10) a reference to:

```
        ARRAY (M, N) would mean ARRAY (M, N, 1)
    and ARRAY (M)       "     "  ARRAY (M, 1, 1)
    and ARRAY           "     "  ARRAY (1, 1, 1)
in addition ARRAY (M, , N)  "     "  ARRAY (M, 1, N)
and so on.
```

Examples of arithmetic expressions containing subscripts are:

```
A(12)
I(7+J*4-6*K)
-A(JROW)+4.2
ARRAY2 (IARRAY(J))
```

In this case the value of the Jth element of the array IARRAY is the subscript of ARRAY2.

5.1.2 Evaluation of arithmetic expressions

The expression

A + B/C

might be evaluated as:

(A + B)/C

or as:

A + (B/C)

Actually, the latter form is correct. However, it is necessary to formulate rules for evaluation so that such ambiguities do not occur. In general these rules correspond to the ordinary rules of algebra.

(i) A subexpression is an expression which is enclosed in parentheses.

e.g. (I + J/K) is a subexpression.

Subexpressions by be nested to any level.

e.g. I*(A+B*(C+D/(H-J)+K)+L)

In this case (H-J) is referred to as the innermost subexpression; (C+D/(H-J)+K) is the next innermost subexpression, and so on.

(ii) The order of evaluation is: first the innermost subexpression followed by the next innermost subexpression until all subexpressions have been evaluated.

(iii) Within a subexpression, (or if no subexpressions are present), the operations are done in the following order:

** (first)	:class 1
* and /	:class 2
+ and - (last)	:class 3

(iv) In expressions where operators of like classes appear, evaluation proceeds from left to right,

e.g. A**B**C is evaluated as (A**B)**C
and A*B/C is evaluated as (A*B)/C

(v) Functions are evaluated before being used as operands.

5.1.3 Truncation

In Fortran, the quotient resulting from a division of two integers (or integer expressions) is always an integer. This property leads to ambiguity as to the result of a division such as 3/4. This result could be 1 or 0 depending on the kind of truncation employed.

In Fortran V, this ambiguity is resolved by use of the TRUNCATION statement.

5.1.4 The TRUNCATION statement

Definition	TRUNCATION t
Where t is IFIX, or IFIXF or INT, or INTF or NINT, or NINTF	

(1) The TRUNCATION statement specifies the kind of truncation to be performed when dividing integers or integer expressions, or in replacement statements (see section 5.3).

(2) If X is the exact quotient resulting from the division of two integers (or integer expressions) then the integer result I, of the division is as follows: -

(i) If TRUNCATION IFIX (or IFIXF) has been specified, or if there is no TRUNCATION specification: - I is equal to the sign of X multiplied by the largest integer which is less than, or equal to the modulus of X.

e.g. The result of
3/4 would be 0
15/16 " " 0
17/16 " " 1
-8/3 " " -2

i.e. rounding towards zero.

(ii) If TRUNCATION INT (or INTF) has been specified, I is equal to the largest integer less than or equal to X.

e.g. The result of
3/4 would be 0
15/16 " " 0
17/16 " " 1
-8/3 " " -3

Note that INT is the same as IFIX, except for negative numbers.

(iii) If TRUNCATION NINT (or NINTF) has been specified:
I is equal to the nearest integer to X.

e.g. The result of
3/4 would be 1
15/16 " " 1
17/16 " " 1
-8/3 " " -3

(3) In the absence of a TRUNCATION statement, (or *RUN option - see (6)), the IFIX truncation will be performed.

(4) The TRUNCATION statement is not executable, and should be placed before any statement which involves arithmetic expressions where a non-standard (i.e. NINT or INT) kind of truncation is required.

(5) If more than one TRUNCATION statement is given, then the first holds until the second is read, the second holds until the third is read, and so on.

(6) The kind of TRUNCATION to be performed (for a whole job) may be specified in the *RUN directive (see section 12.1(8)).

It is important to note that I*J/K may yield a different result from J/K*I.

e.g. under IFIX truncation:

```
4*3/2 = 12/2 = 6
```

but

```
3/2*4 = 1*4 = 4
```

(7) Note that non-integer subscripts and DO, or implied DO parameters are also truncated using the truncation in force when the statement concerned is compiled.

*5.1.5 Type of expressions

An arithmetic expression may contain operands of different types:-

Class 1: Integer, real, double precision, and complex, operands may be present in one arithmetic expression.

Class 2: Text, Boolean, and logical operands may be present in one expression.

Expressions which contain types of different classes are invalid.

The type of the result of an expression is the same as the type of the dominant operand.

The order of dominance of the Class 1 operand types is:

Complex
Double precision
Real
Integer

In expressions containing Class 2 operands, the only permitted arithmetic operation is subtraction.

For expressions of the form X**Y the following table shows the relationship between the type of X**Y and the types of X and Y.

Type of X	Type of Y			
	Integer	Real	Double	Complex
Integer	I	R	D	C
Real	R	R	D	C
Double	D	D	D	C
Complex	C	C	C	C

If X and Y are not Complex the value of X may be negative only if Y is of type integer.

In mixed arithmetic operations, division will be taken as integer division (i.e. division with truncation) if the expressions divided are of type integer, otherwise the division corresponds to the type of the dominant expression in the division.
For example, suppose I, J and K are integers and X is real, then, according to the rules given in 5.1.2,

Expression	Type of division
X + I/J	Integer
(X+I)/J	real
X*I/J	real
I*J/X*K	real
I*J/K*X	integer

Examples of arithmetic expressions;

a) T - 'XY'

Where T is of type TEXT (or Boolean)

b) (X-2)/((I+7.6)*3.6)

c) (1.2,2.4)**I

Where (1.2,2.4) is the complex constant 1.2+2.4i, and I is an integer variable. Note that a complex variable cannot be written as (A,B) where A and B are real variables.

5.1.6 Masking operations

Masking (or Boolean) and shifting operations may be performed on Boolean or Text values (constants or variables) by means of the following special intrinsic (built-in) functions; they are not available when HARTRAN FUNCTIONS, or F2 FUNCTIONS (or OLD FUNCTIONS) is specified (see section 8.4.2).

Function Name	Arguments	Type of Arguments	Type of result	Properties of Function
AND	2	Text or Boolean	Boolean	See below
OR	2	Text or Boolean	Boolean	See below
NOT	1	Text or Boolean	Boolean	See below
ER	2	Text or Boolean	Boolean	See below
SHIFTR	2	First Text or Boolean. Second Integer (n)	Boolean	Shifts first argument right by n bits (circular shift).
SHIFTL	2	First Text or Boolean. Second Integer (n)	Boolean	Shifts first argument left by n bits (circular shift).

The properties of these functions may be explained by assuming that each argument consists of one bit only. Each pair of bits is treated in the same manner as described below.

First Argument (a)	Second Argument (b)	RESULT			
		AND (a, b)	OR (a, b)	ER (a, b)	NOT (a)
1	0	0	1	1	0
0	1	0	1	1	1
1	1	1	1	0	0
0	0	0	0	0	1

i.e. NOT (a) produces a bit string with 1's where a has 0's, and 0's where a has 1's.
AND (a, b) produces a bit string with 1's where both a and b have 1's, and 0's elsewhere.
OR (a, b) produces a bit string with 1's where there are 1's in a or b or both, and 0's elsewhere.
ER (a, b) produces a bit string with 1's where a and b differ, and 0's elsewhere.

5.2 LOGICAL EXPRESSIONS

Logical expressions have values which are either true or false. They can be used to express the relationships between quantities.

5.2.1 Relational expressions

A relational expression is a sequence of arithmetic expressions separated by relational operators.

A relational operator is:

.EQ. or = meaning Equal to
.NE. " Not equal to
.GT. or > " Greater than
.LT. or < " Less than
.GE. or .NL. " Not less than
.LE. or .NG. " Not greater than

No two relational operators may be adjacent.

A relational expression has the value .TRUE. if the arithmetic expressions satisfy the relationship specified by the relational operators; otherwise it has the value .FALSE.

Examples of relational expressions:

a) I.EQ.J-1 (or I=J-1)

The result is true if I is equal to J-1, if not the result is false.
The expression is equivalent to I+1.EQ.J (or I+1=J)

b) I+3.EQ.J*2-K.GT.K/I (or I+3=J*2-K>K/I)

The result of this expression is true if:
I+3 is equal to J*2-K and J*2-K is greater than K/I.
If either, or both of the above conditions is not true, then the result of the expression is false.

5.2.2 Logical expressions

A logical expression is:

a logical constant
a logical variable or function reference.
a relational expression
or a sequence of the above separated by logical operators.

Where a logical operator is
.NOT. meaning negation
or .AND. " logical and
or .OR. " logical inclusive or

Two logical operators may be adjacent only if the second such operator is .NOT.. Note, however, that the sequence .NOT..NOT. is invalid, but .NOT. (.NOT. ...) is accepted.

If L is a logical expression, then (L) is a logical expression.

A logical expression may be preceded by the operator .NOT.

A logical expression has the value .TRUE. or .FALSE.

The properties of the logical operators are as follows:

.NOT. L_1 is false only if L_1 is true
L_1.AND.L_2 is true only if L_1, L_2, are both true
L_1.OR.L_2 is false only if L_1, L_2 are both false.

Where L_1, L_2 are logical expressions.

In a manner similar to that discussed for arithmetic expressions, parentheses are used to explicitly define the sequence of evaluation.

e.g. A.AND.B.OR.C.NE.D

does not have the same meaning as

A.AND.(B.OR.C.NE.D)

Where (B.OR.C.NE.D) may be described as a logical subexpression.

The order of evaluation of logical expressions is:

(i) Arithmetic expressions

(ii) Relational expressions (the relational operators are all of equal precedence).

(iii) The innermost logical subexpression, followed by the next innermost logical subexpression, and so on.

(iv) The logical operators in the following order

.NOT. (first)
.AND.
.OR. (last)

Examples:

a) A.EQ.B.AND.B.EQ.C
(or A=B.AND.B=C)
This is equivalent to the relational expression A=B=C.

b) A=B.AND.C=D.OR.K='TEXTA'
(K being of type TEXT); this expression is true if A is equal to B and C is equal to D. It is also true if K is equal to 'TEXTA'. Otherwise it is false.

c) .NOT.D.OR.X=4.2
D being of type LOGICAL and X of type REAL.
The expression is true if D is .FALSE. The expression is also true if X is equal to 4.2. Otherwise it is false.

5.3 THE ARITHMETIC REPLACEMENT STATEMENT

Definition	v_1, v_2, v_3.....= aexp
Where v_1, v_2, v_3.....are simple or subscripted variable names and aexp is any arithmetic expression	

The arithmetic replacement statement causes the values of the variables on the left hand side of the statement, to be replaced by the value of the arithmetic expression.

The generalised replacement statement defined above is equivalent to the following series of simple replacement statements.

v_1 = aexp
v_2 = aexp
v_3 = aexp
. .
. .
. .

The generalised form is useful in cases where several variables are to be set to the same value.

The variables v_i need not all be of the same type.
When the type of the expression aexp is not the same as the type(s) of the variables (v_i), the variables are assigned types as shown below:

5.3 cont

Type of variable	Type of Expression						
	Integer	Real	Double	Complex	Text	Boolean	Logical
Integer	X	I	I*	CI*	N	N	N
Real	F	X	P*	CR*	N	N	N
Double	F*	P*	X*	CD*	N	N	N
Complex	R*	R*	R*	X*	N	N	N
Text	X	X	N	N	X	X	X*
Boolean	X	X	N	N	X	X	X*
Logical	N	N	N	N	N	N	X

Where the symbols mean:-

X The variable is assigned the exact value of the expression

I The variable is assigned the value of the expression truncated to give an integer value (see section 5.1.3).

F The variable is assigned the real or double precision approximation of the value of the expression.

P The variable is assigned the value of the expression with the precision associated with the variable type.

R The real part of the variable is assigned the value of the expression with real precision. The imaginary part of the variable is assigned the value zero (0.0)

N This combination of type of expression and variable type is not permitted.

CI The variable is assigned the truncated value of the real part of the expression.

CR The variable is assigned the exact value of the real part of the expression

CD The variable is assigned the exact value of the real part of the expression. The least significant half of the variable is set to zero.

* Allowed only in simple form: v = aexp

For statements of the form:

Double precision variable = real expression,

the evaluation of the expression is carried out in double precision mode (except for functions which do not have a double precision form).

e.g. SIN would be acceptable, but TAN would not be because DTAN is not available.

For statements of the form:

text variable = text constant

only the first eight characters of the constant are stored in the variable.

The word "FORMAT" must not appear as the first name on the L. H. S. of the replacement statement.

Examples:

a) I = A(I) (I integer, A real)
Replaces the value of I, by the truncated value of the Ith element of A.

b) I = I+1
The value of I is increased by 1.

c) A(I)=J*J (A real, J integer). The Ith element of A is replaced by the value of J*J after this has been converted to type real.

d) C=I**J + D (C complex, I and J integer, and D double precision)
I is raised to the power of J and the result converted to double precision, D is added. The most significant half of the result is stored in the real part of C, the imaginary part of C is set to zero.

e) I, J(3), K(I, 3) = 2
I and J(3) are set to 2 Since I is now set, K(2, 3) is set to 2.
If the statement is written
K(I, 3), I, J(3)=2, I and J(3) are set to 2, and K(I, 3) is set to 2
where I has the value set before the statement is reached.

f) T = 'SbORbT'
T being of type text

g) B = 4B1477
B being of type Boolean

5.4 THE LOGICAL REPLACEMENT STATEMENT

Definition	$l_1, l_2, l_3, \ldots$ = lexp
Where $l_1, l_2, \ldots$ are simple, or subscripted variable names, which are of type logical, or Boolean. and lexp is any logical expression.	

The logical replacement statement causes the values of the variables on the left hand side to be replaced by the value of the logical expression.

The generalised replacement statement defined above is equivalent to the following series of simple statements

l_1 = lexp
l_2 = lexp
l_3 = lexp
. .
. .
. .
. .
. .

The word "FORMAT" must not appear as the first word on the L. H. S. of the statement.

Examples:

a) L = A.AND.B.OR.C where L, A, B and C are logical (or Boolean).

b) AL = I-J.EQ.4.OR..NOT..3.EQ.X
or
AL = I-J = 4.OR..NOT..3=X

c) G = .TRUE.

d) H = .NOT.G

5.5 THE CLEAR STATEMENT

Definition	CLEAR v_1, v_2, v_3, ...
Where v_1, v_2, v_3 . . . are simple or subscripted variable names.	

The CLEAR statement is used to set variables, entire arrays, or individual elements of arrays to zero. The statement is preferred to the use of DO loops for clearing entire arrays.

Variables of any type may be cleared, and the variables v_i need not be all of the same type. The different types are set as follows:

Type of v	Value after CLEAR v
Integer	0
Real	0.0
Double precision	0.0 (2 words)
Complex	(0.0,0.0)
Text	Blanks ('bbbbbbbb')
Logical	.FALSE. (or 1 BO)
Boolean	1 BO (or .FALSE.)

If an unsubscripted array name appears in the list, the entire array will be cleared.

Examples:

a)
```
TEXT X(6,6) Y(5)
CLEAR X,Y(2)
```
All 36 locations of X, and the second location of Y are set to blanks.

b)
```
REAL R(5),K

CLEAR R(I),K
```
K, and the Ith location of R are set to 0.0

CHAPTER 6 THE CONTROL STATEMENTS

Normally, Fortran statements are executed sequentially. The control statements may be used to control and alter this normal sequence of execution of statements in the program. This is done by referring to a label or statement number.

Fortran V labels may be names as well as statement numbers; and labels may be variable: i.e. Fortran V labels may assume different values.

6.1 LABELS (STATEMENT NUMBERS)

Definition	
A label is a sequence of 1 to 5 alphanumeric characters. If the first character is numeric, then all of the characters must be numeric. The name of a variable (assigned) label has the same definition as a variable name (see section 4.1)	

A label is punched in columns 1 to 5 of a statement, and may be referred to by other statements.

A statement number is a label consisting only of numeric characters. Other labels are named labels.

Blanks may be present in a label; if present they are ignored.

If a label begins with the characters C or X, then the C (or X) must not be punched in column 1.

No two statements may have the same label, (unless one (only) of them is a FORMAT statement).

Any leading zeros punched in a statement number (or named label) are ignored.

Examples: 1
99999
X
Y127J
ZbbZ (same as ZZ)
0ABCD (same as ABCD)
01 (same as 1)

6.1.1 Label variables

The value of a label is defined by the following rules:

(i) If a label is attached to a statement (i.e. appears in columns 1 through 5 of that statement), then the value of the label is the location of that

statement. If no label is attached to a statement, then the location of the statement is not defined. A label with more than 5 characters in its name cannot be attached to a statement.

(ii) At any given time during execution, the value of a label is equal to the value of the last label ASSIGNed to it.

Label variable names are distinct from ordinary (data) variable names (see section 6.2 (5)). This means that labels (or assigned labels) cannot be used as arguments (see section 8.3).

6.2 THE ASSIGN STATEMENT

Definition	ASSIGN l_1 TO l_2
where l_1 is a label (or a label variable), and l_2 is a label variable.	

(1) If l_1 is not a statement number, then it must be enclosed in parentheses.

e.g. ASSIGN (l_1) TO l_2

(2) If l_1 is a label, and l_2 is a label variable, the ASSIGN statement sets the value of l_2 in such a way that future control statements which refer to l_2 will actually be referring to the label l_1.

(3) Labels may be indirectly assigned by writing statements of the form:

ASSIGN $label_1$ TO var_1
ASSIGN (var_1) TO var_2
ASSIGN (var_2) TO var_3
. . .
. . .
. . .
ASSIGN (var_{n-1}) TO var_n

In this case future control statements which refer to the label variable var_n will actually be referring to the label, $label_1$.

(4) Statement numbers which are attached to FORMAT statements must not appear in an ASSIGN statement.

(5) Note that ASSIGN statements only set the values of label variables. If the same name is used for a data variable and a label or label variable, then these two uses of the name are never confused by Fortran V; the actual meaning of the name is determined by the context in which it appears.

e.g. ASSIGN 761 TO IK
A = IK/3

where the value of A is not defined, because the value of the arithmetic variable IK is not defined.
Similarly:

M = 5

cannot be substituted for

ASSIGN 5 TO M

and vice versa.

6.3 THE GO TO STATEMENT

Definition	GO TO i, (l_1, l_2, l_3....)
where i is a label or a label variable and l_1, l_2...are labels The comma preceding the left parenthesis may be omitted. The part of the statement which follows i may be omitted: if present it is ignored.	

If i is a label, the GO TO statement causes control to be transferred to the statement to which that label is attached.

If i is a label variable, the GO TO statement passes control to the statement whose label was last directly or indirectly assigned to i.

Examples: (where ILABEL and JLABEL are label variables)

a)
```
      GO TO 10
10    X = 3
```

b)
```
      ASSIGN 10  TO ILABEL
      GO TO ILABEL
10    X = 3
```

c)
```
      ASSIGN 10 TO ILABEL
                .
                .
                .
      ASSIGN (ILABEL) TO JLABEL
                .
      GO TO JLABEL, (10, 20, 30)
                .
10    X = 2
```

In this case both ILABEL and JLABEL are set to label 10.

6.4 THE COMPUTED GO TO STATEMENT

Definition	GO TO (l_1, l_2, l_3, ... l_n), aexp
where l_1, l_2, ... l_n, are labels or label variables Any of l_1, l_2, ... or l_n may be omitted and aexp is an arithmetic expression whose value should lie within the range $1 < aexp \leq n$. aexp should be of type integer. The comma preceding aexp may be omitted.	

This statement causes control to be transferred to the statement whose label is l_1, l_2, l_3 ... or l_n, depending on whether the value of aexp is 1, 2, 3 or n respectively. If any of l_1, l_2 .. or l_n are label variables, then control will be passed to the statement whose label was directly or indirectly assigned to l_1, l_2 ... or l_n.

If the value of aexp is less than 1, or greater than n, then the transfer of control is undefined. This error is automatically tested for in execution, if the routine is compiled in TEST mode. See sections 12.2 and 10.3.1 (error 11).

Null (or omitted) labels have the value "next statement".

e.g. GO TO (l_1, , l_2,) aexp

If the value of aexp is 1, control is passed to l_1
If the value of aexp is 2, control is passed to the next statement.
If the value of aexp is 3, control is passed to l_2
If the value of aexp is 4, control is passed to the next statement.

Examples:

a) GO TO (10, X, 5), I

Control is passed to the statement whose label is 10, X, or 5 depending on whether the value of I is 1, 2 or 3.

b) ASSIGN 10 TO ILABEL
GOTO (ILABEL, , 15)K+5-J(3)

Control is passed to the statement whose label is 10 or 15 depending on whether the value of K+5-J(3) is 1 or 3.

If the value of K+5-J(3) is 2, control is passed to the statement which follows the computed GOTO statement.

6.5 THE ARITHMETIC IF STATEMENT

Definition	IF (aexp) l_1, l_2, l_3
where aexp is any arithmetic expression and l_1, l_2 and l_3 are labels or label variables. Any of l_1, l_2, or l_3 may be omitted. The two commas must always be present.	

The statement causes control to be transferred to the statement whose label is l_1, l_2, or l_3 when the value of the arithmetic expression is less than zero, equal to zero, or greater than zero respectively.
If any of l_1, l_2, or l_3 are label variables, then control will be passed to the statement whose label was last directly or indirectly assigned to l_1, l_2 or l_3.

If the expression is of type complex, then only the real part is tested for zero.

Null (or omitted) labels have the value "next statement".

e.g. IF (aexp), l_1,

If the value of aexp is zero, control is passed to the statement whose label is l_1: otherwise control is passed to the statement which follows the IF statement.

Care should be taken when l_1 is a named label (or a label variable) that the sequence l_1, l_2, l_3 does not satisfy the syntax of any Fortran statement. If this occurs, the arithmetic IF may be taken as a logical IF, which has a different meaning.

e.g. IF(X) READ1, A, B

may be taken as a logical IF.

Examples:

a) IF(I**3 - K(N))1,,X

The value of I**3 - K(N) is computed. If this value is zero, control is passed to the next statement; if the value is negative control is passed to the statement whose label is 1; if it is positive control is passed to label X.

b)
```
ASSIGN 10 to L
ASSIGN (L) to X
IF (T - 'AB') 10,11,X
```

If the text variable T is equal to 'AB', control is passed to label 11, if it is not equal to 'AB', control is passed to label 10.

6.6 THE LOGICAL IF STATEMENT

The logical IF statement has two forms. These are defined below.

6.6.1 The Fortran IV logical IF

Definition	IF (lexp) stat
where lexp is any logical expression and stat is any executable Fortran V statement except a DO statement or Fortran IV type logical IF statement.	

(1) If the value of the logical expression lexp is .TRUE. the statement stat is executed, and control is then passed to the next statement (unless stat is a control statement which has altered this normal path of control).
If the value of lexp is .FALSE., the statement stat is ignored and control is passed to the next statement.

(2) Care should be taken when testing real quantities for strict equality, since rounding errors may give an unexpected result.

e.g.
```
I = 3
.
.
.
IF (I.EQ.3) GOTO 6
```

is preferable to

```
X = 3.0
.
.
IF(X.EQ.3.0) GOTO 6
```

(3) If the statement stat is a BEGIN statement (see Chapter 9), then the logical IF statement may be used to jump round, or execute, a whole block of statements, instead of just one statement.

If this facility is used, then, (if the logical expression lexp is false), control is passed to the first executable statement following the END statement which corresponds to the BEGIN.

If the logical expression is true then the block of statements following the BEGIN are executed.

The block of statements contained between the BEGIN and its END is subject to the same rules as a normal block, these rules are described in Chapter 9.

Examples:

a) IF (I.EQ.4) I = 5

b) IF (I = 4) I = 5

c) IF (A.GT.B.AND.C.EQ.'S'.AND.L)X = 3

d) IF (I = J = K = I) IF (X - 3.0) 1,2,3

6.6.2 The Hartran logical IF

Definition	IF (lexp) l_1, l_2
where lexp is any logical expression and l_1 and l_2 are labels or label variables. l_1 or l_2 may be omitted. The comma must always be present.	

If the value of the logical expression lexp is .TRUE., control is passed to the statement whose label is l_1; if the value is .FALSE. control is passed to the statement whose label is l_2.

If l_1 or l_2 are label variables, then control will be passed to the statement whose label was last directly or indirectly assigned to l_1 or l_2.

Null (or omitted) labels have the value "next statement".

If l_1 is a named label (or a label variable), care should be taken that the sequence l_1, l_2 does not satisfy the syntax of any Fortran V statement. If this occurs the Hartran logical IF statement may be taken as a Fortran IV logical IF, which has a different meaning (see section 6.6.1).

Examples:

a) IF (I.EQ.0) 12,

This has the same effect as

IF (I.EQ.0) GOTO 12

b) IF (J.LT.X) 10,20

If J is less than X control is passed to label 10, if not, control is passed to label 20.

c) ASSIGN(X) TO I
IF (K=I.OR.Z<Y),I

If K is equal to I or Z is less than Y, then control is passed to the next statement; if not, control is passed to the statement whose label is X.

6.7 THE DO STATEMENT

This statement provides a means of repeating groups of statements (looping), and changing the value of a variable during the loop.

6.7
cont

Definition	DO l v = $aexp_1$, $aexp_2$, $aexp_3$

where
l is a label or a label variable
and
v is a nonsubscripted variable of type integer, or real
and
$aexp_1$, $aexp_2$ and $aexp_3$ are any arithmetic expressions which are of type integer or real.
$aexp_3$ and its preceding comma may be omitted. If they are omitted $aexp_3$ is taken to be 1.
$aexp_1$, $aexp_2$ must always be present
An optional comma may be inserted between l and v.

If l is a named label then it must be enclosed in parentheses, or followed by a comma.

e.g. DO (l) v = $aexp_1$, $aexp_2$, $aexp_3$
or DO l, v = $aexp_1$, $aexp_2$, $aexp_3$

The DO statement causes the statements which follow it, up to and including the statement labelled l, to be executed repeatedly. These statements are said to be within the range of the DO.
The variable v is the index of the DO loop, and $aexp_1$, $aexp_2$, and $aexp_3$ are the loop parameters. $aexp_1$ is the first limit, $aexp_2$ is the second limit, and $aexp_3$ is the loop step.
At the start of the loop, the index is set to the value of the first limit, and the loop is entered.
At the end of the loop, the step is added to the index. If the step is positive, the loop is re-entered so long as the index is less than, or equal to the second limit. If the step is negative, the loop is re-entered so long as the index is greater than, or equal to, the second limit.

Upon completion of the DO, control is passed to the statement which follows the statement labelled l. When this occurs the value of the index v is not defined.

The statements within the range of the DO are always executed at least once. Thus, the number of times the range is executed is given by:-

$$N = 1 + \left[\frac{aexp_2 - aexp_1}{aexp_3} \right]$$

where the brackets indicate the largest non-negative integral value not exceeding the value of the expression within those brackets.
If the step ($aexp_3$) is zero, the loop may be executed an infinite number of times.

If any of the expressions $aexp_1$, $aexp_2$, $aexp_3$ are real, and the index (v) is an integer, then the value(s) of the expression(s) will be truncated to give integer values for the purpose of computing the number of times the loop is to be executed. See sections 5.1.3 and 5.1.4.

If the index of the DO is of type real, then care should be exercised when choosing the limits.

e.g. in the case of

```
DO 2 X = 1.0,2.0,0.1
```

the execution with X equal to 2.0 may be lost due to rounding errors. A better choice of limits would be

```
DO 2 X = 1.0, 2.001, 0.1
```

6.7 cont

The label, l, must appear in the text of the program after the DO statement.

The label, l must not be attached to any non-executable statement (see Appendix 3) or to any DO statement.

If l is attached to any IF or computed GOTO statement which contains a null (omitted) label, then the null label represents the end of the DO loop. Execution of the loop is thus continued normally, if control is passed to the null label.

l should not be attached to a statement which causes a transfer of control back into the range of the DO.

The values of any variables contained in $aexp_2$ or $aexp_3$ should not be changed by statements in the range of the DO. If this is done, the execution of the loop may be affected.

The value of the index (v) must not be changed by statements contained within the range of the DO.

The index (v) may be changed by statements outside the range of a DO, only if no transfer is made back into the range of the DO which uses that index.

A transfer of control out of the range of a DO is permitted at any time.

Control may be transferred into the range of a DO only when control has at some previous time been transferred out of it.

Subprograms may be called, from, and returned to, within the range of a DO.

There may be other DO statements within the range of a DO. All statements within the range of the inner DO must also be within the range of the outer DO. A set of DO statements which satisfy this rule is called a nest of DO's.
Nesting may be to any level. The index of an inner DO must not be the same as the index of any DO containing it.

Examples:

a)
```
      X = 1
      DO 2 I = 1, 6
    2 X = X*5
```
When the DO loop is completed, X is equal to 5 to the power 6.

b)
```
      REAL INNER (60)
      DO 3 II4 = 1, 60
    3 INNER (II4) = 3.2
```
All elements of INNER are set to 3.2

c)
```
      REAL INTER (10, 20)
      DO 4 I = 1, 20
      DO 4 J = 1, 10
    4 INTER (J, I) = 3.2
```
All elements of INTER are set to 3.2

d)
```
      INTEGER ARRAY (40)
      DO 5 INDEX42 = 1, 40, 2
    5 ARRAY (INDEX42) = INDEX42*2
```

ARRAY (1) is 2, ARRAY (3) is 6,
and so on. Even numbered elements of ARRAY are undefined.
This could also be accomplished by:

```
      DO 5 INDEX42 = 39, 1, -2
    5 ARRAY (INDEX42) = INDEX42*2
```

e)
```
      DO 6 I = 1,N
6     IF(A(I)), 8,
      GOTO 7
8     A(I) = 1.0
7     ---------
```
If statement 8 is reached, then A(I) is zero.
Thus the first zero element of A is set to 1.0. If none of the first N elements of A are zero, statement 8 is not executed.

f)
```
      DO 10 I = 1, 10
      DO (X) J = I, 10      } inner DO
X     C(J+1) = A(I, J)      }
10    A(I) = 0
```
outer DO

The inner DO is executed for J = 1 to 10, then statement 10 is executed; then the inner DO is executed for J = 2 to 10; then statement 10 is executed again. This continues until the last execution for which I is equal to 10.

6.8 THE CONTINUE STATEMENT

Because of the restriction on the kinds of statements which can end the range of a DO, it is convenient to have a statement to which a label may be attached, but which otherwise has no effect on the execution of the program.

Definition
CONTINUE

The CONTINUE statement may be placed anywhere in a source program.

No instructions are compiled for the CONTINUE statement, although it is considered to be executable.

Examples:

a)
```
      DO 15 I = 1, 20
5     IF (A(I) - B(I))  10, 15, 15
10    A(I) = A(I)+1
      B(I) = B(I)-2
      GOTO 5
15    CONTINUE
```
The CONTINUE provides a means of avoiding ending the DO with GOTO 5

b)
```
      DO (X) I=1, 10
      IF (A(I)-B(I))  Y, Z, Z
Y     A(I) = B(I)
      GOTO X
Z     A(I) = 0
X     CONTINUE
```
The CONTINUE provides a means of bypassing statement Z.

6.9 THE PAUSE STATEMENT

Definition	PAUSE x
where x is an unsigned integer constant or a primed text constant or may be omitted.	

In other versions of Fortran, the PAUSE statement is often used to halt the machine and wait for operator intervention. Since this is not possible on Atlas the PAUSE statement is treated as a CONTINUE in Fortran V.

The constant x is not printed in Fortran V.

6.10 THE STOP STATEMENT

Definition	STOP n
where n is an unsigned integer constant or may be omitted.	

The STOP statement is executable, and terminates execution of the compiled (object) program.

In Fortran V, n (if present) is not printed.

CHAPTER 7 INPUT AND OUTPUT

7.1 INTRODUCTION

The input and output (I/O) statements provide a means of transferring data between I/O devices and internal storage. On Atlas, the available input devices are:

(i) Magnetic tape units (1" or $\frac{1}{2}$" tape)
(ii) 80 - column card readers.
(iii) 5,7, or 8 - track paper tape readers.

and the output devices are:

(i) Magnetic tape units (1" or $\frac{1}{2}$" tape)
(ii) 80 - column card punches.
(iii) 5,7, or 8 - track paper tape punches.
(iv) Line printers (120 printable characters per line).

Some of these devices may be remote from Atlas and connected to it by means of a data - link.

There are four standard I/O statements: READ, which causes data to be transferred from an input device to internal storage, PRINT, PUNCH, and WRITE, which cause data to be transferred from internal storage to an output device.

The manner in which I/O devices are specified is described in section 7.2. The FORMAT statement, which is not executable, may be used in conjunction with the above I/O statements in order to specify the precise manner in which data is to be transferred.

In addition, in Fortran V there is a special I/O statement, which does not reference a FORMAT statement:

OUTPUT

which causes data to be transferred from internal storage to an output device.

In addition to the above I/O statements, there are four statements which are used to manipulate magnetic tape units. These are

REWIND, BACKSPACE, ENDFILE, and UNLOAD.

7.2 SYMBOLIC I/O DESIGNATION

Atlas provides for a very powerful and generalised means of referring to its peripheral (I/O) devices. The way in which this is done is described below.

7.2.1 Output

The particular device which is accessed by any given output statement is dependent upon:

(i) The logical device number, (or stream number) which is referred to by the statement.

(ii) The way in which this stream number is described in the job description.

Full details of the job description are given in Appendix 8. For every different stream number referred to, the job description will contain a line (statement) which relates the stream number to the name of the type of output device to be used for that stream.

e.g. If stream number 6 is referred to by a statement such as:

WRITE (6, f) list

and it is desired that this statement is to produce output on a line printer, then the job description will contain the line:

OUTPUT 6 LINEPRINTER....

Alternatively, if the output is to be produced on cards, then the job description will contain a line

OUTPUT 6 CARDS .....

In this way, the output statements in the program are independent of the device used.

7.2.2 Input

The input statements also refer to stream numbers, and for each different stream number, a descriptive line must appear in the job description. For input, this line does not describe the type of input device to be used, since this must already be fixed by the time the program is ready to be run. Instead, the line relates the stream number to a document name. This name is, in fact, the name of the document to be input on that stream and, in addition to being given in the job description, must also be declared at the start of the input itself.

e.g. for a statement of the form

READ (5, f) list

which refers to input stream 5, the job description will contain the line:

INPUT 5 name of document five

and the data to be read on input stream 5 will be preceded by the two lines:

DATA
name of document five

In this way, the job description and the input statements in the program, are independent of the type of input device used.

7.2.3 Magnetic tapes

Most of the I/O devices cannot be used for both input and output (e.g. output cannot be produced on a card reader). This is not true of magnetic tapes, however, since the same tape can be written to and read from in one program. As a result, the words INPUT and OUTPUT in the job description described above are not meaningful when a magnetic tape is to be used as the I/O device; and the word TAPE is used instead.

e.g. WRITE (6) list

would require a job description line of

TAPE 6 label on tape

and

READ (5) list

would require

TAPE 5 label on tape.

A maximum of 6 one inch magnetic tape units, and/or two $\frac{1}{2}$ inch (IBM-compatible) units may be used by any one job.

Tape numbers must not conflict with input or output stream numbers.
Tape numbers for one inch tapes must be in the range 0 through 99.
Tape numbers for $\frac{1}{2}$ inch tapes must be in the range 0 through 15.

7.3 RECORDS

Note: this section does not apply to the Fortran V statement OUTPUT.

In Fortran, input and output is done by records, and not one value at a time. An I/O statement will transfer one (or more) records to or from storage, and this record may contain many different values.

In general, a record is:

(i) a card
or (ii) a length of paper tape contained between new line characters.
or (iii) a length of magnetic tape contained between end-of-record markers.
or (iv) one line as printed by a lineprinter.

7.4 THE I/O LIST

When transferring information to or from internal storage, it is necessary to know which parts of storage (i.e. which variables) are involved.

When variables (or arrays) are transmitted by means of an I/O statement, an ordered list of the quantities to be transmitted must be included in that statement. The order of this I/O list must be the same as the order in which the information exists in the I/O medium.

A list used in an input statement (input list) has a slightly more restricted definition than an output list (or I/O list).

7.4.1 Definition

(1) A simple I/O list is a simple input list or a simple output list.

(2) A simple input list is a series of simple or subscripted variable names separated by commas.

(3) A simple output list is a series of arithmetic expressions separated by commas. Note that logical expressions are not allowed, but simple or subscripted logical variables or logical constants are allowed.

(4) If X is a simple I/O list, then an I/O list (under the control of an "implied DO") is:

$$(X,\ v = aexp_1,\ aexp_2,\ aexp_3)$$

7.4.1 cont

Where

v is a non-subscripted variable name, of type integer, or real,
v is the index of the implied DO.

and $aexp_1$, $aexp_2$, and $aexp_3$ are arithmetic expressions of type integer, or real. They are the parameters of the implied DO. $aexp_3$ and its preceding comma may be omitted. If this is done the value of $aexp_3$ is taken as one (integer 1).

(5) If Y is any I/O list, then (Y, v = $aexp_1$, $aexp_2$, $aexp_3$) is an I/O list, where v, $aexp_1$, $aexp_2$, and $aexp_3$ are defined in rule (4).

(6) If Y is an I/O list, then (Y) is an I/O list, the parentheses being redundant.

(7) If X and Y are I/O lists then X, Y is an I/O list.

(8) The execution of a list of the form

(X, v = $aexp_1$, $aexp_2$, $aexp_3$)

is exactly that of DO loop, as though each left parenthesis (except subscripting or redundant parentheses) were a DO statement with the indexing information given before the matching right parenthesis, and with the DO range extending up to that information. That is:-

```
      DO 10 v=aexp1 aexp2, aexp3
      transfer information to or from the list X
      (allowing for the changing value of the index v)
10    CONTINUE
```

The above list is also equivalent to the list

X, X, X, X .....

(allowing for the changing value of the index v).

Where the number of times that X is repeated is determined by the values of $aexp_1$, $aexp_2$ and $aexp_3$ in the same way as for a DO statement (see section 6.7).

Note, however, that (for example)

```
      DO 2 I = 1, 10
2     READ 20, A(I)
```

will read at least ten records, since each execution of the READ statement brings in a new record.
Whereas:

```
      READ 20, (A(I), I=1, 10)
```

may read more, or less, than 10 records, depending on FORMAT number 20.

(9) For a list of the form

K, A(K)

or

K, (A(I), I=1, K)

Where the definition of a subscript or an indexing parameter appears earlier in the list of an input statement than its use, the indexing (or subscripting) will be carried out with the newly read-in value.

(10) If the input or output of an entire array is desired, then only the name of the array need

be given in the I/O list, and the indexing information may be omitted.

e.g. If A has previously been declared to be an array (i.e. has been given dimension information), then the following statement is sufficient to read in all of the elements of A;

```
READ (5, 10) A
```

If A has not previously been dimensioned then only one value will be read in. The elements transmitted by this notation are stored (or output) in accordance with the description of the arrangement of arrays in storage. (see section 4.2.2)

This notation, which is sometimes called a "short list" is more efficient, both in compilation and execution than the form:

```
READ (5, 10) (A(I), I = 1, N)
```

where N is the number of elements contained in A.

(11) The I/O list controls the quantity of data that is transmitted. On input, if more quantities are present in the external record, than are in the list, then only the number specified in the list are transmitted, and the remaining quantities are ignored.

Conversely, if a list contains more quantities than are given in one input record, then more records(s) are read.

(12) If an I/O statement contains a list with an implied DO, and this I/O statement is within the range of a DO statement, then the index of the implied DO must not be the same as the index of the DO statement.

e.g.

```
      DO 20 I = 1, 6
      READ (5, 10) (A(I), I = 1, 20)
   20 CONTINUE
```

would be illegal since the READ statement changes the value of I (the index of the DO statement) within the range of the DO statement (see section 6.7)

Similarly, if a list contains nested implied DO's then none of these implied DO's may share the same index.

e.g.

```
READ (5, 10) ((A(I), I=1, 10), B(I), I=1, 10)
```

would be illegal.

7.4.2 Examples of I/O lists

Examples of I/O lists which may be used in conjunction with input or output statements. The variables may be of any type.

a) I

The value of I is transmitted to an output device, or a value obtained from an input device is transferred into I.

b) I, J, A(K+3)

Values are transferred to or from the variables I and J and the element A(K+3), where K has been set previously.

c) (A, I=1, 6)

For output, the value(s) of A is output 6 times. For input, 6 values are read into location A, all values being overwritten except the last one.

d) (A(I), I=1, 6)

Values are transferred to or from the elements A(1), A(2).....A(6), in order.
This list is equivalent to the list:
A(1), A(2), A(3), A(4), A(5), A(6)

If A has been declared as
DIMENSION A(6)
then the list is also equivalent to the list: A

e) (A(I), I=6, 1, -1)

This is the same as d), except that values are transferred in the reverse order.
i.e. A(6), A(5), ...A(1)

f) A, K, ((AR(I, J), B(I), I=1, 20), C(J), J=1, 5), X

This is equivalent to the following pseudo statements (but see section 7.4.1(8)):-

```
transfer value to or from location A
   "      "   "  "   "      "     K

DO 10 J=1,5
DO 20 I=1,20

transfer value to or from location AR(I,J)
   "      "   "  "   "      "      B(I)

20 CONTINUE
   transfer value to or from location C(J)
10 CONTINUE
   transfer value to or from location X
```

7.4.3 Examples of output lists

Examples of lists which may be used only in conjunction with output statements.

a) 1

the number 1 is written

b) 'OUTPUTbTABLE'

the characters OUTPUTbTABLE are written

c) X+Y**3

the value of X plus the cube of Y is computed, and this value is written.

d) (A(I)*B(I), I=1, 20)

The value of A(1) times B(1) is computed and written; then A(2) times B(2), and so on, up to A(20) times B(20)

e) (SIN(X)**3, X=0, PI/2, PI/180)

This outputs the cubes of the sines of angles from 0 degrees to 90 degrees in steps of 1 degree. (Functions are described in section 8.4). PI (if it means π) would previously have been set to 3.1 4 1 5 9 .....

f) LOG, .TRUE.

This writes the values .FALSE..TRUE. or .TRUE..TRUE. depending on the value of the logical variable.

7.5 UNFORMATTED (BINARY) I/O STATEMENTS

As mentioned in section 7.1 certain of the I/O statements may be used in conjunction with a FORMAT statement which gives details of how information is to be transferred. Sometimes, however, this information may not be required, and it may be adequate to transfer data to or from internal storage with no intermediate conversion (e.g. from decimal to binary or internal (binary) code to text, and so on). When this is the case, no FORMAT statement need be referred to: any output produced will be in the form of a string of binary digits, and any input presented must be in the same form.

Unformatted I/O statements are meaningful only when the I/O device is a one-inch magnetic tape unit.

7.5.1 The unformatted READ statement

Definition
READ (ldn), input list or READ TAPE ldn, input list
where ldn is any arithmetic expression of type integer, whose value is in the range $0 \leq ldn \leq 99$ and input list is defined in section 7.4.1. In either definition, the comma preceding the input list may be omitted.

The two definitions given above are equivalent, but READ is preferred to READ TAPE.

The unformatted READ statement causes one record to be transferred to internal storage from the input device whose logical number is equal to the value of expression ldn. The record is assumed to be in binary form. If the input list contains more items (words) than are present in the record read, then an error will occur (execution error 10 - see section 10.3). If the list contains less items than are present in the first record read, then the remaining items in the record are ignored.

The input list (with any preceding comma) may be omitted from either form of the unformatted READ statement. If this is done, then one record on tape number ldn will be skipped (i.e. a record is read, but no information is transmitted to internal storage).

Example: READ (10) A, (B(I), I=1, 3)

The values obtained from the record read from tape 10 are loaded into
A, B(1), B(2), and B(3)
in order.

7.5.2 The unformatted WRITE statement

Definition
WRITE (ldn), I/O list or WRITE TAPE ldn, I/O list
where ldn is any arithmetic expression, of type integer, whose value is in the range $0 < ldn \leq 99$ and I/O list is defined in section 7.4.1 In either definition the comma preceding the I/O list may be omitted.

The two definitions are equivalent, but WRITE is preferred to WRITE TAPE.

The unformatted WRITE statement causes one record to be written by means of the output device whose logical number is given by the value of ldn.
The record is written in binary form.

The I/O list (with any preceding comma) may be omitted from either form of the unformatted WRITE statement. If this is done, then one record of zero length (i.e. a record containing no information) is written on to tape number ldn.

Example: WRITE (ITAPE + 6) A,(B(I), I=1,3)

The values contained in A,B(1), B(2), and B(3) are written, in order, on to the tape whose logical number is equal to the value of ITAPE + 6.

7.6 THE FORMAT STATEMENT

The FORMAT statement is used in conjunction with certain (formatted) I/O statements in order to specify the precise manner in which values are to be transferred.

A format specification can be thought of as a piece of program in a special language, which is executed interpretively, when the corresponding I/O statement is obeyed. The various characters which can occur in the format specification correspond to "instructions" which are obeyed by the computer when the I/O statement is executed.

A summary of the conversion and control specifications available in Fortran V is given below:

Code	Used For
An	Character information
Bn*	Octal digits
D n.m	Double precision numbers
E n.m	Real numbers
F n.m	Real numbers
G n**	Real numbers
nH	Text
In	Decimal integers
Kn*	Treats characters as integers
Ln	Logical values (true/false)
On	Octal integers
nP	Scale Factor (D, E and F)
nQ*	Scale Factor (D and E only)
nR*	Scale Factor (F only)

continued

S*	Sign control
Tn	Position on line
nX	Skips columns
nY*	Position on line
nZ*	Prints or suppresses leading zeros
'chars'	Text

* These codes are not usually present in other versions of Fortran (they are all in Hartran).
** This code is usually written in the form G n.m.

7.6.1 Definition

Definition

FORMAT (format specification)

Where format specification is written as series of field specifications and/or control specifications which are normally separated by commas or slashes.

These specifications may be: -

(i) A text constant (primed or Hollerith)
(ii) S followed by two characters.

Note The characters following S, and the characters in a text constant may be chosen from the entire character set (see Appendix 2), and blanks are significant. Elsewhere, blanks are not significant and only the following characters may be used:

A B D E F G I K L O P Q R S T X Y Z
0 1 2 3 4 5 6 7 8 9
( ) / , . + -

(iii) Any of A B G I K L or O (owe) followed by an unsigned or negative integer and optionally preceded by an unsigned integer.

(iv) D or E or F followed by an unsigned or negative real constant of the form n.m or - n.m, and optionally preceded by an unsigned integer.

(v) Y or Z preceded by an unsigned integer.

(vi) Any of P Q R or X preceded by a signed or unsigned integer (of the form n or + n or -n)

(vii) T followed by an unsigned integer.

(viii) A sequence of specifications may be enclosed in parentheses and be optionally preceded by an unsigned integer. This constitutes a specification which may itself by included in a sequence enclosed in further parentheses, and so on up to depth eight.
Note Field specifications with negative or zero field widths are described in section 7.6.6, and 7.6.7.

7.6.1 cont

(1) Every FORMAT statement must have a statement number (as opposed to a named label) attached to it.
The statement numbers attached to FORMAT statements are treated separately from normal labels, and a FORMAT statement may have the same statement number as another (executable) statement. If this statement number is referred to by a control statement, then the executable statement will be referenced, and not the FORMAT statement.

(2) The FORMAT statement is not executable, and may appear anywhere in the program except as the last statement in the range of a DO.

(3) One FORMAT statement may be referred to by any number of formatted I/O statements.

(4) A sequence of characters to be input or output which are processed together is called a field.
The field width is the number of consecutive characters concerned, and must be specified in all field specifications.

In Fortran V, the specified field width may be negative in order to allow for a "format free" input of values. Full details of this facility are given in section 7.6.6.

(5) By preceding it with an unsigned integer constant, a field specification may be repeated as many times as desired.

An integer preceding a conversion code A, B, D, E, F, G, I, K, L or O specifies that the conversion is to be applied to the specified number of consecutive items in the I/O list.

e.g. The specification

4 F 10.6

is equivalent to

F 10.6, F 10.6, F 10.6, F 10.6

(6) A group of field specifications enclosed in parentheses and preceded by an unsigned integer is repeated the specified number of times. If no integer is specified, the group is scanned once only.

A parenthetical group is itself a specification and may occur within another group; and so on to depth eight.

e.g. 3(I2, F3.1)
is equivalent to:

I2, F3.1, I2, F3.1, I2, F3.1

and

2(I4, 3(I2, A8), I6)
is equivalent to:

I4, I2, A8, I2, A8, I2, A8, I6, I4, I2, A8, I2, A8, I2, A8, I6

(7) When a formatted I/O statement is executed, the relevant FORMAT statement is scanned. Whenever a field specification is reached which refers to an I/O list item, the next item in the list is processed and transmitted.
Execution of the I/O statement is ended when all list items have been transmitted and either another item is called for by a format specification, or when the end (i.e. last right parenthesis) of the format specification is reached.

(8) If the end of a format specification is reached, and items still remain to be processed in the I/O list, then the format specification is re-scanned from the last nest of repeated field specifications (due regard being paid to any associated count).

7.6.1 cont If there are no repeated groups of field specifications, the format specification is re-scanned from the beginning (first left parenthesis).

(9) A FORMAT statement may define Fortran records as follows:

(i) If no slashes or additional parentheses appear within a format specification, then a Fortran record is defined by the beginning of the specification (left parenthesis) to the end of the specification (right parenthesis). Thus, on input, a new record is read when the format control is initiated (left parenthesis). On output, a new record is written when the format control is terminated (right parenthesis).

(ii) If slashes appear within a format specification, then Fortran records are defined by the beginning of the format specification to the first slash in the specification; from one slash to the next slash; or from the last slash to the end of the format specification.

Thus, on input, a new record is read when the format control is initiated, and thereafter a new record is read upon encountering each slash. On output, a new record is written upon encountering a slash or when the format control is terminated.

Thus, both a slash, and the final right parenthesis of a format specification indicates the termination of a record.

(iii) If more than one level of parentheses appears within a format specification, then a record is defined by the beginning of the format specification to the end of the specification; and thereafter records are defined from the beginning of the last nest of repeated field specifications to the end of the format specification.

Example:

a) (A2, (I4, I8))

If the I/O list were long enough to induce repeated scanning this specification would define records as follows:

```
A2,   I4,   I8
I4,   I8
I4,   I8
I4,   I8
I4,   I8
   .
   .
   .
   .
```

b) (A2, 3(I4, I8))

This defines records as follows:

```
A2, I4,  I8,  I4,  I8,  I4,  I8
I4,  I8,  I4,  I8,  I4,  I8
I4,  I8,  I4,  I8,  I4,  I8
          .
          .
          .
          .
```

(10) Blank output records may be introduced, or input records skipped, by using consecutive

7.6.1 cont slashes in a format specification. If there are n consecutive slashes at the beginning, or end, of a format specification, n input records are skipped, or n blank records are inserted between output records, respectively. If n consecutive slashes appear anywhere else in a format specification, then the number of records skipped or blank records inserted is n-1.

(11) During repeated scanning of a format specification, the scaling factors P, Q, and R, the zero suppression control Z, and the sign control S, are not reset, but have the values as at the end of the previous scan. They are re-set (to normal if not specified) at each execution of any formatted I/O statement.

(12) In Fortran V, all records to be input or output are passed through a buffer. The maximum number of characters which this buffer can hold is 160. Thus, no attempt should be made to input or output a record containing more than 160 characters by means of a formatted I/O statement.

This limitation does not apply to the unformatted (binary) I/O statements.

If more than 160 characters are fed to the buffer then only the first 159 and the last 1 will be retained, and all of the other characters will be lost.

e.g. 10F20.10

The characters corresponding to the last two conversions are lost and the last character of the 8th conversion is overwritten.

(13) In addition to the above limitation on record length, some of the I/O devices have their own limitations.

The maximum number of characters which can be printed on one line by a lineprinter is 120 characters (121 with carriage control character - see section 7.6.2).

The maximum number of characters which can be read from (or punched on) one card by a card reader (or punch) is 80 characters.

If an attempt is made to print a record of (say) 130 characters, then two lines will be printed, and the 122nd character will be taken as the carriage control of the second line.
Similarly, if a record of (say) 90 characters is read from a card reader, then 2 cards will be read.

(14) When transmitting values on input, the type of conversion code, type of data, and type of variable in the input list should correspond.
When transmitting values on output, the type of the output value will correspond to the conversion code used; and this code need not be of the same type as the variable in the output list.
It is not normally useful to output text information using numeric conversion codes (or vice versa), but it may be useful to output a numeric value of type real as an integer, or an integer as real, and so on.

(15) Complex numbers must be input as two real numbers (using F or E conversion).
Complex numbers must be output as two values using two conversions (normally F or E or I).

(16) Commas must be used to separate field specifications in cases where ambiguities would arise if the specifications were not separated.

e.g. F15.1, 3I8

Would be ambiguous without the comma. Commas need not be inserted after H or primed text constants, after slashes, or after any of the control specifications (section 7.6.5).

7.6.2 Carriage control

When records written under format control are prepared for printing, the first character of the output record is treated as a carriage control character and not printed. The table below shows the effect of various characters on the line spacing. Note that on all I/O devices other than the lineprinter the first character in a record is treated as normal data.

First Character	Carriage Advance Before Printing
blank	One line
0 (zero) (or A)	Two lines
1	First line of next page (skip to channel 1)
2	Skip to channel 2
3	" " " 3
4	" " " 4
5	" " " 5
6	" " " 6
7	" " " 7
+	No advance (overprinting)
B	Three lines
C	Four lines
D	Five lines
E	Six lines
F	Seven lines

If any other character appears as the first character of a print record, then the carriage is advanced by one line, and the record is then printed. In this case the (illegal) carriage control character is printed and all other characters in the line are shifted right by one place.

The channel skips allow the line to be printed at a standard position on the page, regardless of the current position. The position skipped to is dependent upon the way in which the printer is set up, and may be found by enquiring at the installation where the program is to be run.

7.6.3 The non-numeric field specification

In this section it is assumed that all field widths specified are positive. The properties of negative and zero field widths are described in sections 7.6.6, and 7.6.7.

7.6.3.1 A conversion

Form	An
where n is an unsigned integer	

A conversion is used to transmit data in character (text) form. On input each character read is converted into internal code and stored. On output each internal code number (of 6 bits) is converted to a character and output. The Atlas internal code is described in Appendix 2.

On input, n (of An) characters are read.
If n is less than 8, then the characters read will be left adjusted within a word, and filled out with blanks.

```
e.g.        READ 10, CHAR
       10   FORMAT (A1)
```

will read one character, if this character is X (say), then the variable CHAR will have the value 'X' or 'Xbbbbbbb'.

If n is greater than 8, then only the first eight characters are stored, and the next n-8 input characters are ignored.
Example: -
If one whole card is to be read in and stored, then at least 10 words must be allowed in the list:

```
      TEXT CARDA (10)
      READ 5, CARDA
5     FORMAT (10A8)
```

80 characters are read, and stored 8 characters per word in the 10 elements of CARDA. If FORMAT (A80) had been used, then the first 8 characters read in would be stored in CARDA(1) and the next 72 input characters would be ignored; so that CARDA(2) to CARDA(10) would be undefined. If FORMAT (80A1) had been used, then CARDA would have been dimensioned: - TEXT CARDA (80), and each character would have been stored in one element of CARDA.

On output, n characters are written. If n is less than or equal to 8, then the output field will consist of the n left-most characters of the word.

If n is greater than 8, then the field will be output as above (with n=8) but will be preceded by n - 8 blanks.

```
e.g.      PRINT 1, 'ABCDEFGH'
      1   FORMAT ('b', A8)
```

will print ABCDEFGH.

If the FORMAT is ('b', A12), then bbbbABCDEFGH is printed.

7.6.3.2 B conversion

Form	Bn
Where n is an unsigned integer constant.	

B conversion is used to transmit octal digits between I/O devices and internal storage.

On input, n (of Bn) octal digits are read.
Any characters read under B conversion should be 0, 1, 2, 3, 4, 5, 6, 7, or blank.
An error (see section 10.3) will occur if any other character is read. Any blanks (leading, trailing, or embedded) are taken as zero.

If n is less than, or equal to 16, then n characters are stored left-adjusted within a word. The last 16-n octal digits of the word are set to zero (00).

If n is greater than 16 the first 16 characters are read and stored, and the next n-16 characters of the input record are ignored.

```
e.g.      BOOLEAN DIG
          READ 10, DIG
10        FORMAT (B8)
```

If a card (say) has its first 8 columns punched 0123b4b0 and is read as above, then DIG will have the (octal) value 0123040000000000.

On output, n octal digits are transmitted to the external record.
Each 3 bits of the word (starting from the left), are converted to an octal digit and

written. Zeros are written as zero, and not as blank. If n is less than 16, then the n left-most digits of the word are written and the remaining digits are ignored.

If n is greater than (or equal to) 16, then all 16 digits of the word are written followed by n-16 zeros (not blanks).

7.6.3.3 H and primed conversion

Forms	nHcharacter string or 'character string'.
Where n is an unsigned integer constant (not zero).	

Blanks are significant in the character string. In the H form, n must be equal to the number of characters contained in the string.

In the primed form, an apostrophe cannot be included within the string.

e.g. 'DON''T' must be written as 5HDON'T

H or primed conversion is used to transmit data in character form. Both forms of conversion have the same effect.

On input, the n characters in the string are replaced by the next n characters read. This facility can be used to change titles, dates, column headings, etc., which are to appear on an output record generated by the H or primed specification.

On output, the n characters in the string are transmitted to the external record.

Examples:

a)
```
      PRINT 20
20    FORMAT ('TABLEb2')
```

b)
```
      READ 10
10    FORMAT (80H <eighty blanks>)
      PRINT 10
```

The READ instruction reads one card (say), and the 80 characters contained on this card replace the 80 blanks of the H specification.

The PRINT instruction prints the characters contained in the H string, and thus the card read is printed. The character in the first column of the card is treated as a carriage control character for the printed line.

7.6.3.4 K conversion

Form	Kn
where n is an unsigned integer constant	

K conversion provides a means of treating characters as integers, and is most useful on input, with the value of n equal to 1.

On input, n (of Kn) characters are read. If n is greater than 6, then the first n - 6 characters are ignored, and only the last six are processed. If n is less than, or equal to 6, then n characters are processed.

7.6.3.4 cont The characters are stored (in Atlas internal code form) in a word in such a way that the word can be treated as a Fortran V integer. The corresponding list item should be of type integer.

If the specification is K1 (n=1), then the value of the integer will be equal to the internal code number of the character read (see Appendix 2).

```
e.g.        READ 1,I
        1   FORMAT (K1)
```

If the character read is a blank, then I will have the value 1 (one), since the internal code for blank is 01. If the character read is A, then I will have the value 33, the internal code for A being 41 (octal), which is equal to 33 decimal.

If n is greater than one, then the n characters read are converted so that the first character forms the most significant part of the integer, and the last character the least significant part. Note that leading blanks (except on K1) are stored as binary zero (internal code 00).

```
Example:    READ 1,I
        1   FORMAT (K2)
```

If the 2 characters read are both blank, then the value of I will be 0001 (octal), i.e. 1 in decimal.

If the characters read are "1A" then the value of I will be 2141 (octal) i.e. 1121 in decimal.

On output, n characters are written. The word to be output (i.e. the list item) should be of type integer. If n is less than, or equal to 6, then the 2n least significant octal digits of the integer part of the word are converted into n characters and output. (This is exactly the reverse of the procedure for input). Non-printable characters are printed as a decimal point, except that leading zeros (internal code 00) are printed as blanks.

If n is greater than 6, then the output field is as described above, (with n = 6) but is preceded by n - 6 blanks.

Examples:

a)
```
        PRINT 10, I
    10  FORMAT (K1)
```

If I is equal to 14, then * is printed (internal code = 16 octal).

If I is equal to 2761 (=5311 octal), then) is printed (internal code = 11 octal).

b)
```
        PRINT 10,I
    10  FORMAT (K2)
```

If I is equal to 97 (=0141 octal) then bA is printed.

If I is equal to 1 (=0001 octal), then bb is printed, the leading zero (00) being printed as a blank.

If I is equal to 2761 (=5311 octal) then K) is printed.

7.6.3.5 L conversion

Form	Ln
where n is an integer constant	

L conversion is used to transmit logical values (true or false) between I/O devices and internal storage.

On input n (of Ln) characters are read. If n is greater than 5 only the first 5 characters are processed, and the next n - 5 are ignored.

If n is less than, or equal to 5, the input field should consist of the n left-most character(s) of the words.

TRUEb

or

FALSE

the result stored being the logical value .TRUE. or .FALSE. respectively. Leading, and embedded blanks are illegal. All illegal fields (including an all-blank field) are read in as false, but a data error occurs (see error 5, section 10.3.1.)

On output, n characters are transmitted to the external record. If n is less than or equal to 5, then the output characters will be the n left-most characters of the words

TRUEb

or

FALSE

depending on whether the value of the corresponding list item is .TRUE. or .FALSE..

If n is greater than 5, the output field will be as above (with n = 5), but is followed by n - 5 blanks.

7.6.4 The numeric field specification

The numeric conversions I, F, E, G, D and O are used to specify input/output of integer, real, double precision, complex or octal-integer data.

With all numeric input conversions, in the field read, leading blanks are not significant, and trailing blanks are taken as zero. Embedded blanks (i.e. within numbers) are taken as zero, but an input error (error 5) occurs. This error can be avoided (see section 10.4). A field of all blanks is taken as zero. See also section 7.6.6.

With F, E, G and D conversion, a decimal point appearing in the input field overrides the decimal point position implicit in the field specification.

With all input conversions a plus sign may precede a positive value, and a minus sign must precede a negative value. If no sign is given, the value is taken as positive.

With all numeric output conversions, the written number is right adjusted. If the number of characters produced by the conversion is less than the field width specified, then leading blanks are inserted in the output field (leading zeros, or blank fields, may be obtained by use of the Z control specification - see section 7.6.5.5).

If the field width (n) specified is less than the number of characters produced by the conversion, then only the n right-most (i.e. least significant) characters are transmitted to the output record.

With all numeric output conversions a minus sign is output for negative values, and no sign (i.e. a blank sign) is output for positive values. These conventions may be overidden by means of the S control specification (see section 7.6.5.2).

For the real number conversions, E, F and G, accuracy is maintained to about eleven decimal digits. If fields containing more than eleven digits are transferred, then the least significant digits are lost (on input) or inaccurate (on output).

7.6.4.1 D conversion

Form	Dn.m
where n and m are unsigned integer constants	

D conversion is used to transmit double precision values between internal storage and I/O devices.

D conversion is similar to E conversion with the following exceptions:-

(i) For input, the character D may be present instead of E. As for E conversion the D character may be omitted altogether. Values punched with an E are also accepted by D conversion.

(ii) For output, the character D will be present instead of E.

e.g. for D12.5
-66.334 is converted to -6.63340D+01

As for E conversion, if the exponent exceeds 99, the D character is not written.

For double precision values, accuracy is maintained to about 20 decimal digits. If fields containing more than 20 digits are transferred, then the least significant digits will be lost (on input), or inaccurate (on output).

7.6.4.2 E conversion

Form	En.m
where n and m are unsigned integer constants	

E conversion is used to transmit real numbers between internal storage and an I/O device. Either part of a complex number may be processed using this conversion.

If double precision values are processed using E conversion, then accuracy is to single precision. In En.m, m is the number of digits present in the fractional part of the field. (See also section 7.6.4).

On input, n (of En.m) characters are read, and converted to a real number. The field read must be in one of the following forms:

$\pm x\ E \pm k$
or $\pm x + k$
or $\pm x - k$
where:
$\pm$ is an optional plus or minus sign
x is an unsigned real or integer constant
and
k is an unsigned integer constant.

7.6.4.2 cont The exponent k is the power of 10 by which the number x is to be multiplied. If a decimal point is present in x, then it overrides the implicit decimal point given by the value of m in En.m.

Note that trailing blanks on an exponent are taken as zeros - they are not ignored.

e.g. for conversion E6.0 and the input field

b1.E01

the result is 10.0, but if the input field were 1.E01b then the result would be 1.0×10^{10}

An all blank field is read as zero, but an input error occurs (execution error 5 - see section 10.3).

Examples:

Input characters	Specification	Internal Value
bb-113409E2	E11.6	-11.340900
+b471316-03	E11.6	.000471316
bb1234+5	E8.0	123400000.0
b1.36E01	E8.4	13.6
1.36E01b	E8.4	1.36×10^{10}

First, the decimal point (if not present in the input field) is positioned according to the specification; then the value of the exponent is applied to determine the actual position of the decimal point. In the first example - 113409E2 is interpreted as - .113409E02, which when evaluated (i.e. $-.113409 \times 10^{2}$) becomes -11.340900.

On output, n characters are transferred. Internal values are converted to real constants of the form:-

d.ddd ... dE+ee
d.ddd ... dE-ee
d.ddd ....dEeee
or d.ddd ....d-eee

where .ddd ....d represents m (of En.m) decimal digits, and $\pm$ ee and $\pm$ eee are interpreted as multipliers of the form

$10^{\pm ee}$ and $10^{\pm eee}$

These forms of output may be modified by using a scale factor (see section 7.6.5.1).

Internal values are rounded to m+1 digits, and negative values are preceded by a minus sign.

The field width is counted from the right and includes the exponent digits, the exponent sign (minus or plus), the letter E, the magnitude digits, the decimal point, and the sign of the value (minus or blank). If the width n is not sufficient to allow expression of an entire value, then only the n right-most digits will appear. This is not an error condition. To prevent a loss of this nature it is necessary to ensure that (in En.m)

$$n \geq m + 7$$

Note that this feature can be used intentionally (in conjunction with a scale factor) to obtain the multiplier field, which is an indication of the order of magnitude of the internal value.

e.g. for E3.0
60255.334 is converted to ·+04
0.0000072 is converted to -06

Examples:

Value	E10.3	E8.3	E6.3
-2013.55	-2.014E+03	.014E+03	14E+03
.361887	3.619E-01	.619E-01	19E-01
.0001	1.000E-04	.000E-04	00E-04

7.6.4.3 F conversion

Form	Fn.m
where n and m are unsigned integer constants	

F conversion is used to transmit real numbers between internal storage and an I/O device. Either part of a complex number may be processed using F conversion. If double precision values are processed using this conversion, then accuracy will be to single precision only.

In Fn.m, n is the field width, and m is the number digits present in the fractional part of the field. (see also section 7.6.4).

On input, n (of Fn.m) characters are read, and converted to a real number. The field read in must be in one of the following forms:

±i

±.i

±i.

±i.j

or blank

where: ± is an optional plus or minus sign
and i and j are unsigned integer constants.

An all blank field is taken as zero.

If a decimal point is present, then its position overrides the implicit position given by the value of m in Fn.m. If m is greater than n, then the implicit position of the decimal point is m-n places to the left of the first digit of the field read in.

Examples: for the specification F7.3:-

bbbbb33 is converted to 0.033
1234567 is converted to 1234.567
33bbbbb is converted to 3300.000
-1.63bb is converted to -1.630
bb-1.63 is converted to -1.630

On output, n characters are transferred. Internal values are converted to real constants, rounded to m decimal places, with an overall field width of n, (m should not be greater than n).

If a value requires more positions than are provided by the magnitude of n then only the n right-most characters are printed. This is not an error condition. In order to ensure that a loss of sign or digits does not occur, the following relation must hold true (in Fn.m)

$n \geq m+2+d$
where d is the number of digits to the left of the decimal point.

Examples:

Internal Value	Specification	Output Field
273.4	F9.4	b273.4000
273.4	F4.4	4000
-442.306	F7.2	-442.31
63	F5.1	b63.0
62.7	F3.0	63.

7.6.4.4 G conversion

Form	Gn
where n is an unsigned integer constant	

G conversion is used to transfer real numbers between internal storage and an I/O device. It combines the properties of E and F type conversions. Either part of a complex number may be transferred using G conversion. If double precision values are transferred by means of G conversion, then accuracy will be to single precision only. See also section 7.6.4.

On input, n (of Gn) characters are read, and converted to a real constant. The field read in may take any of the forms accepted by E and F type conversions, and is processed in the same way.

If a decimal point is not explicitly provided in the input field, the position of the decimal point will be assumed to be at the right-hand end of the magnitude part of the field.

e.g. If read on G6:
bb1277 is converted to 1277.
1277bb is converted to 127700.
b-25E3 is converted to -25000.
-.25E3 is converted to -250.

On output, n characters are transferred. Internal values are converted to real constants. The form of the constant is dependent on the magnitude of the data, and conversion is either E type or F type according to the following rule:

The F form is used if the value can be expressed without either leading zeros or an exponent, otherwise the E form is used. The conversion chosen is the one which allows for the maximum number of significant digits; i.e. the maximum accuracy compatible with the field width, n. The following rules also apply for output G conversions.

(i) If n (of Gn) is less than 7, then the F form is always used.
(ii) A maximum of 13 significant decimal digits will be printed, whatever the value of n. (The 13 does not include sign, decimal point, or exponent.)
(iii) If n is greater than 6, and if the number is such that the F form is used, then four blanks are printed at the right hand end of the output field (i.e. in the place where an exponent would otherwise appear). In general for Gn, where 6<n<20 the number of significant digits printed is n - 6.

Examples: If the specification G10 is used

Internal Value	Output Field
1.0	b1.000bbbb
-1	-1.000bbbb
1.000606	b1.001bbbb

continued

57.9999	b58.00bbbb
0.1	b1.000E-01
0.00123	b1.230E-03
1234567881.0	b1.235E+09
-6666666	-6.667E+06
1.7E+10	b1.700E+10

7.6.4.5 I conversion

Form	In
where n is an unsigned integer constant	

I conversion is used to transfer decimal integer values between internal storage and an I/O device. Double precision values and either part of complex values should not be input using I conversion; but may be output by this means. (see also section 7.6.4.)

On input, n (of In) characters are read and converted to a decimal integer. The input field may contain a signed or unsigned integer constant, or may be blank. If the field is blank, its value is taken as zero. Any values read in should lie within the range for an integer constant (see 3.1).

The input field must not contain a decimal point or an exponent.

Examples: if specification I6 is used

Input Field	Internal Value
bbbb10	10
b-bb1b	-10
1bbbbb	100000
776543	776543
b-11bb	-1100
+b11bb	1100

On output, n characters are written. Internal values are converted to integer constants. Real and double precision values are truncated to give the nearest integer (NINT) and then output.

If the field width n is not sufficient to contain the output field, then only the n right-most characters of the field are output.

Examples: if specification I5 is used

Internal Value	Output Field
10	bbb10
-1001	-1001
100000	00000
7.53	bbbb8
-113.7653	b-114

***7.6.4.6** O conversion

Form	On
where n is an unsigned integer constant	

O conversion is used to transmit octal integers between internal storage and an I/O device. Real, double precision, and complex values may not be processed using O conversion.

On input, n (of On) characters are read and converted to a decimal integer. The input field should be a signed or unsigned octal constant, or be blank. If the field is blank, its value is assumed to be zero. An octal constant is a combination of the digits, 0, 1, 2, 3, 4, 5, 6 or 7 but not 8 or 9.

Examples: if specification O6 is used

Input field	Internal Value (decimal)
bbbbb7	7
bbbb10	8
bbbb1b	8
1bbbbb	32768
b+6001	3073
-bb333	-219

On output, n characters are written. An internal value, which should be in INTEGER form, is converted to an octal integer. If the field width n is not sufficient to contain the number, then only the n right-most characters are printed.

Examples: if specification O5 is used

Internal Value (decimal)	Output Field
1	bbbb1
8	bbb10
-64	b-100
1100	b2114
32769	00001

7.6.5 The control specifications

The control specifications are used to modify the input or output of the format specifications described above.

Leading zeros may be printed, or blank fields obtained, by means of Z; plus signs etc. (normally suppressed) may be printed by means of S; spacing may be controlled by means of X, Y and T; and the magnitudes of numbers may be controlled by means of the scale factors P, Q and R.

*7.6.5.1 P, Q and R specification

Forms	iP iQ and iR
where i is a signed or unsigned integer constant	

Two scale factors: Q (for E and D type conversion), and R (for F type conversion) are maintained during processing of a FORMAT statement.

These specifications cause the scale factor to be set to i, where the scale factor is treated as a multiplier of the form:

10^{i} for output
and 10^{-i} for input

at the beginning of each formatted I/O operation, before any processing occurs, the scale factor Q has the value 1 and R has the value 0.

These values may be altered by means of the specifications iR and iQ. The specification iP sets them both to the same value (i).

If the specification iQ is used, then for E and D type output, the mantissa is normalised to the range

$$10^{i-1} \leq \text{mantissa} < 10^{i}$$

(so that there are i integer places), and the decimal exponent is reduced by i. Q is also effective for E type output of output G conversions. On input, Q is not effective.

If the specification iR is used, then for F type output (and F type output of output G conversions), the number written has the value of 10^{i} times the value of the corresponding list item. On input the number read is multiplied by the 10^{-i} before assignment to the list item.

Scale factors are effective only within: -

Output D or E conversions
Output E type G conversions
Input and output F conversions
Input and output F type G conversions.

Any number of P, Q or R specifications may be present in one FORMAT statement, thereby causing the value(s) of the scale factor(s) to be changed several times during a formatted I/O operation. If a FORMAT is restarted within a single I/O operation due to the number of items in the I/O list, then the scale factors are not re-set to their standard values.

The scale factors do not affect I conversions.

Examples:

a) Input

	Internal Value					
External Value	F7.3	2RF7.3	-2RF7.3	E10.3	2QE10.3	-2QE10.3
14.633 -0.234	14.633 -0.234	0.14633 -0.00234	1463.3 -23.4	14.633 -0.234	14.633 -0.234	14.633 -0.234

b) Output

	Output Field					
Internal Value	F8.3	2RF8.3	-2RF8.3	E11.3	2QE11.3	-2QE11.3
14.633 -0.234	bb14.633 bb-0.234	1463.300 b-23.400	bbb0.146 bb-0.002	bb1.463E+01 b-2.340E-01	b14.633E+00 -23.400E-02	bb0.001E+04 b-0.002E+02

***7.6.5.2** S specification

Form	Sc_1c_2
where c_1 and c_2 are any characters, blanks being significant.	

S control is used to modify the output of numerical information (i.e. information written using D, E, F, G, I, or O conversions).

The character to be output immediately to the left of the left-most digit is specified to be: -

c_1 for zero or positive numbers

c_2 for negative numbers

In the absence of the S specification, c_1 is blank and c_2 is -.

If a FORMAT is re-started within a single I/O operation due to the number of items in the I/O list, then c_1 and c_2 are not reset to the standard values. Several S specifications may appear in the same FORMAT statement.

The S specification has no effect on input conversions.

Examples:

Internal Value	Specification	Output Field
1	S-b, I2	-1
-1	S-b, I2	b1
-76.3	S**, F6.2	*76.30
.001	S00, E10.3	01.000E-03

7.6.5.3 X specification

Form	nX
where n is a signed or unsigned integer constant	

X specification is used to adjust the position in the line where the next character is to be output, or the position in an input record from where the next input character is to be taken. The position of the next character of the line in the external medium to be processed is identified by a pointer. This pointer is advanced automatically when fields of specified widths are processed. It can also be explicitly adjusted by means of X control. nX moves the pointer n places relative to its present position. The value of n must not be such as to move the pointer outside the record.

On input, if n is positive, then the next n input characters are ignored. If n is negative, then the previous n characters are "re-read".

e.g. with F5.2, 6X, I3
and the input string
32.6345AB12366

the characters 45AB12 are not processed, and the numbers read will be 32.63 and 366.

The negative specification is useful since it enables the same characters to be read two (or more) times under different conversions. Zero fields can thus be distinguished from blank fields (see example b)).

Examples:-

a)
```
      READ 10, J, A

10    FORMAT (I3, -3X, F3.0)
```

The same value is now available as a real number (A) and as an integer (J).

b)
```
      TEXT T
      READ 20, K, T
20    FORMAT (I4, -4X, A4)
```

The same value is now available as an integer (K) and as a sequence of characters (T). The value of K is zero both if zero(s) are punched and if the field is left blank. Which

characters are actually present can be determined by testing the value of T.

e.g. IF (T='bbbb') GO TO 25
if statement 25 is executed then blanks were read, and not zeros.

On output, if n is positive, then n blanks are inserted into the output record. If n is negative, then the previous n characters in the output string are deleted and further output creation begins at the left-most position of those n characters.

e.g. The specifications

'FORMULA', -4X, 'TRANSLATING'

will create

FORTRANSLATING

as the output string.

7.6.5.4 T and Y specification

Forms	Tn or nY
where n is an unsigned integer constant	

T or Y specification adjusts the record pointer in the same way as X specification, but instead of being a relative adjustment, T or Y adjusts the pointer to the specified absolute character or column position (n).

Tn has exactly the same effect as nY.
The value of n should be less than 160 (see section 7.6.1.(12)).

Example: On output, the specifications

60Y, 4H****, -4X, S*-, I4

will cause an integer to be output in columns 60 to 63 with preceding asterisks instead of blanks.

The pointer may be "backspaced" in a similar way to X specification, but the value of n is again absolute rather than relative.

7.6.5.5 Z specification

Form	nZ
where n is an unsigned integer constant	

Z specification is used to obtain or suppress leading zeros in numeric output conversions. It has no effect on input conversions.

nZ specifies that not less than n digits are to be output in I or O type fields; and that a minimum of n digits are to be output before the decimal point in E, D, F and G type fields.

It applies to all numeric conversions after the Z control is processed, and is not re-set to the standard value of 1Z if the FORMAT is re-scanned. Several Z specifications may appear in the same FORMAT statement. Note that the specification 0Z provides a very simple means of printing blank fields for zero values, without affecting the output of non-zero values.

In the absence of a Z specification, 1Z is assumed so that all leading zeros are suppressed, except a single zero for a zero integer, or for a zero integer part of a real number. The field width(s) specified in the numeric conversions must be

sufficient to allow for the n digits output under nZ.

Examples:

Internal Value	Specification	Output Field
1	3Z, I6	bbb001
0	6Z, I6	000000
3.2	3Z, F6.2	003.20
-4.1	3Z, E10.1	-004.1Eb00
0	0Z, I6	6 blanks
0.0	0Z, F8.4	8 blanks

7.6.6 "Format free" input

The use of fixed field widths as described above is not always convenient, as data is sometimes provided as numbers of variable lengths which are separated by spaces. (This is often the case when the data is punched on paper tape - see section 7.6.9).

In order to deal with this kind of data, Fortran V allows the field widths (n) of all of the numeric conversions described in section 7.6.4 (i.e. D, E, F, G, I and O) to be negative (i.e. preceded by a minus sign).

The number of characters accepted by a numeric input conversion (with a negative field width) is dependent on the data presented, and not on the format specification. The value of the negative field width is irrelevant so long as it is not zero (see 7.6.7). Thus, both F-1.3, and F-6.3 have the same effect. The rules for the input of numbers in free format are

(i) spaces before digits are ignored.
(ii) a space after digits terminates the number.
(iii) End of record (e.g. newline) is treated as space.
(iv) A number is started by any non-blank character.

If a blank, or end of record is read then reading of the number is terminated at that point and the characters read in so far are presented for conversion:

The characters which terminate reading of a number are;

blank (or space)
tab (see section 7.6.9)
newline (7 track tape)
line feed (5 track tape)
End of card
End of magnetic tape record.

If an end of record is found when the input list is not satisfied, then a new record is read and processing continues until all values in the list have been read in. If the list is satisfied and characters remain to be processed in the input record, then these characters are lost.

If numbers having exponents (for D, E, or G conversion) are to read, then no blanks may appear between the exponent and its mantissa (since this would terminate the read). Similarly, zero values must be explicitly punched, and cannot be left blank. It will be seen that completely blank records are ignored.

If a real number is read, and no explicit decimal point is punched in the input field, then the (implicit) position of the decimal point is decided as for normal (positive field width) conversion.
e.g. if 51234b is read on F-8.2, then the result stored is 512.34. The blank terminates the read, and the decimal point is two digits from the right-hand end.

If a negative field width is attached to a non-numeric input conversion (i.e. A,B,K or L), then the buffer pointer is moved back by the number of columns given by the field width, and the corresponding list item is assigned one of the following values

```
A  conversion  :  'bbbbbbbb'
B  conversion  :  16BO
K  conversion  :  O
L  conversion  :  .FALSE.
```

e.g. A-8 is equivalent to -8X,AO
(see section 7.6.7)

If a negative field width is attached to any output conversion, then the corresponding list item is ignored (skipped), and the buffer pointer is moved back by the number of columns given by the field width.

e.g. on output
F-10.6
is equivalent to
-10X, F0.0

Examples:

```
      READ 10,I,J,X,Y
10    FORMAT (2I-3, F-3.1, F-6.2)
```

If the input string is
2b34b7.6b885b.....
Then
I will be 2
J will be 34
X will be 7.6
Y will be 8.85

The .2 of F6.2 specifies the position of the decimal point. With the same instructions, if the input string is

```
      bbb123bb4b-6.7bbb34b......
Then  I  will be 123
      J will be 4
      X will be -6.7
      Y will be 0.34
```

*7.6.7 Zero field widths

In all of the conversions dealt with in sections 7.6.3 and 7.6.4, the field width specified may be zero. On input conversions, a field of width zero does not take any characters from the input string and is effectively the same as reading an all blank field.

The actual values assigned are:

```
A0  conversion  :  'bbbbbbbb'
B0  conversion  :  16B0
K0  conversion  :  0
L0  conversion  :  .FALSE.
D0.0, E0.0, F0.0, G0 conversions  :  0.0
I0,O0  conversions  :  0
```

For output conversions a zero field width means "ignore the next item in the output list". No characters are output.

Examples:

a)
```
      READ 10, K, J
10    FORMAT (I0, I3)
```
with an input string of
123....

K would be zero and J would be 123.

b)
```
      PRINT 10, K, X, J
10    FORMAT (I2, F0.0, I3)
```
If K were equal to 12 and J were equal to 345 the output string produced would be
12345

7.6.8 Variable formats

Format specifications may be specified at the time of program execution. The specification must include its surrounding parentheses but not a statement label nor the word FORMAT. The specification should be stored into a TEXT array or (if the specification is less than 9 characters long), into a single TEXT variable. H and primed fields may be included.

The specification may be stored in one of three ways

(i) by reading it in under A conversion
(ii) by setting it up in DATA statement or in a TEXT type statement.
(iii) by setting it up by one or more arithmetic replacement statements.

The name of the array or variable is then used in I/O statements in place of the usual FORMAT statement number. If a TEXT array is used then only its name should be given in the I/O statements which refer to it, this name should not be subscripted.

Example: Assume that the following characters are punched on a card:

(F6.2, 10X, I8, E13.4)
This card could be read in as follows:

```
      TEXT F(3)
      READ 10, (F(I), I=1, 3)
10    FORMAT (3A8)
```

Subsequent I/O statements can now refer to the array F as though it were a FORMAT statement.

e.g.
```
      READ F, A, K, X
```
or
```
      WRITE (6, F)A, K, X
```
and these statements would be equivalent to:

```
      READ 10, A, K, X
```
or
```
      WRITE (6, 10)A, K, X
```
with
```
10    FORMAT (F6.2, 10X, I8, E13.4)
```

The specification could also be stored in the array F as follows:-

a)
```
      TEXT F(3)/' (F6.2, 10X, I8, E13.4)'/
```
b)
```
      TEXT F(3)
      DATA F/' (F6.2, 10X, I8, E13.4)'/
```

or

c)
```
TEXT F(3)
F(1)='(F6.2,10'
F(2)='X,I8,E13'
F(3)='.4)'
```

*7.6.9 Special features of paper tape input

In addition to the "format free" input described in section 7.6.6, the following features facilitate data input from paper tape. This section does not apply for input from cards or magnetic tape.
All erases, and redundant upper and lower case characters are always ignored, and the characters tab, backspace, query, and inner and outer set shift are treated specially (see below).

Other characters are stored until a maximum of 160 characters is reached, any further characters in the record are then ignored.

Inner and outer set shifts are stored only when there is actually a character of that set to store. Since space appears in both sets, the sequence A erase space space B, which appears in internal code as A shiftout er sp sp shiftin B, would actually be stored as A sp sp B.

The effect of tab depends on the number of characters which have been stored from the record. If there are 6 or more, then tab is treated as space. If there are less than 6, then the number of stored characters is increased to 6, by planting the appropriate number of spaces.

Note that a record containing tab is not a null record.

The character backspace has the effect it would have on a Flexowriter print-out, so that the character preceding the backspace is overwritten by the character which follows it. Compound characters (e.g. $\leq$) are not constructed. Backspace is ignored if it would cause characters to be planted before column one.

If the character ? (query) is punched in a line of paper tape input, then the whole of that line (up to the ?) is ignored, and input is begun again at the character which follows the query. This provides a convenient method of deleting mis-punched lines.

The character (normally ?) which causes lines to be ignored can be changed by the user by calling the library subroutine SKIPCH(C).

The argument C must be set to 64s+K; where s is 0 for inner set, and 1 for outer set characters; and K is equal to the internal code of the desired character stored as a Fortran V integer.

e.g. CALL SKIPCH(33)

causes paper tape records to be ignored when A appears.

If s is set to 2 (e.g. CALL SKIPCH (128)) then the facility for ignoring records can be removed. The standard behaviour is as though CALL SKIPCH(12) were executed on entry to the main program.

Records containing no characters other than backspace, erases, run-out, and lower case are ignored. Such records are null records.

The library subroutine NULSET(N) can be used to cause null records to be accepted, and not ignored. If the argument N is not zero, then null records will not be ignored, but will be treated as a blank card image. The standard action is as though CALL NULSET(0) were obeyed on entry to the main program.

7.7 THE FORMATTED READ STATEMENTS

Definition
READ (ldn, fn), input list or READ INPUT TAPE ldn, fn, input list or READ fn, input list
Where ldn is any arithmetic expression of type integer, whose value is in the range $0 \leq lnd \leq 15$ (or 99 - see below) and fn is a statement number attached to a FORMAT statement, or the name of a TEXT array or variable in which a format specification has been stored. and input list is defined in section 7.4.1 The comma following the right parenthesis in the first definition is normally omitted.

The first two definitions are equivalent, but READ (ldn, fn) is preferred to READ INPUT TAPE.

The third form is equivalent to:

READ (0, fn) input list

i.e. it reads from input stream number zero (logical input device number zero).

The formatted READ statements cause one (or more) records to be transferred to internal storage from the input device whose logical number is equal to the value of the expression ldn. If the input device is a one inch magnetic tape then the value of ldn should be less than 99, for other devices, the value should be less than 16.

If the input list contains more items than are present in the first record read, then further records will be read until the list is satisfied provided that the associated FORMAT statement is such as to cause extra records to be read, e.g. the specification (2015) should not be used to read 80 column cards, as the last 4 (of 20) list items would be undefined. (see 7.6.1 (9)). The FORMAT statement referred to may also cause extra records to be transferred (see section 7.6).

If the list contains less items than are present in the first record read, then the remaining items in the record are ignored.

The input list may be omitted from any form of the formatted read statements. If this is done then one record on device number ldn will be skipped (i.e. a record is read, but no information is transferred to internal storage). There is an exception to this rule which is described in section 7.6.3.3. More than one record may be read if slashes are present in the FORMAT statement.

Records containing more than 512 words (4096 characters) cannot be read from half-inch magnetic tape. This limitation does not apply to one inch (Ampex) tape.

All reference to the format number fn may be omitted from the formatted READ statements, (Except READ INPUT TAPE)

e.g. READ (ldn,) input list
and READ, input list

the commas must be present.
If this is done the format specification

(6G20)
is referenced.

Examples:

a)
```
      READ 10, A,B,C
10    FORMAT (3F10.6)
```

b)
```
      READ (I+4, 10)A, B, C
10    FORMAT (3F10.6)
```

a record is read from the device whose logical number is equal to I+4.

c) READ (5, VF) (A(J), B(J), J=1, 6)

where VF is a TEXT variable or array containing a format specification.

d) READ (I/J+1, X)C, V, K, (A(J), J=K*2, -1, -1)

7.8 THE FORMATTED WRITE, PRINT AND PUNCH STATEMENTS

Definition
WRITE (ldn, fn),I/O list or WRITE OUTPUT TAPE ldn, fn, I/O list or PRINT fn, I/O list or PUNCH fn, I/O list
where ldn is any arithmetic expression of type integer, whose value is in the range $0 \leq ldn \leq 15$ (or 99 - see below) and fn is a statement number attached to a FORMAT statement, or the name of a TEXT array or simple variable in which a format specification has been stored. and I/O list is defined in section 7.4.1 In the first definition, the comma following the right parenthesis is normally omitted.

The first two definitions are equivalent, but WRITE (ldn, fn) is preferred to WRITE OUTPUT TAPE.

The third form (PRINT) is equivalent to

WRITE (0, fn) I/O list

i.e. it writes to output stream zero (logical device number zero), which is normally the line printer (see Appendix 8). Similarly, the fourth form (PUNCH) is equivalent to

WRITE (15, fn)I/O list

In the Fortran V system, output stream 15 is normally reserved for the 80 column card punch, so that cards are normally produced by the PUNCH instruction. See Appendix 8. The formatted WRITE or PRINT or PUNCH statements cause one (or more) record(s) to be transferred from internal storage to the output device whose logical number is equal to the value of the expression ldn. If the output device is a one-inch magnetic tape unit, then the value of ldn should be less than 99, for other devices, the value should be less than 16. The number of records written is dependent on the FORMAT statement referenced.

The I/O list may be omitted from any form of the formatted WRITE or PRINT or PUNCH statements. If this is done, the record(s) created are dependent on the corresponding FORMAT statement.

```
e.g.      PRINT 10
      10  FORMAT (6HbTITLE/)
```

All reference to the format number fn may be omitted from the formatted WRITE or PRINT or PUNCH statements, (except WRITE OUTPUT TAPE).

```
e.g.      WRITE (ldn, )I/O list
          PRINT, I/O list
          and
          PUNCH, I/O list
```

the commas must be present. If this is done, then the format specification

```
          (6G20)
```

is referenced.

If half inch tapes are used, then any records written should not contain more than 512 words (4096 characters). This limitation does not apply to one inch (Ampex) tapes.

Examples:

```
a)        PRINT 10, A, B, C
      10  FORMAT (3F10.3)

b)        WRITE (ITAPE, 10) A, B, C
      10  FORMAT (3F10.3)
```

One record is written to the device whose logical number is equal to the value of ITAPE.

```
c)        WRITE (6, R) X, Y, (A(I), I=1, 3)
```

where R is a TEXT variable or array containing a format specification.

```
d)        WRITE (I*2, FMT)  X+Y, 2.7,  (K, A(K)**2, K=10, J-3)
```

7.9 THE MAGNETIC TAPE MANIPULATION STATEMENTS

The following statements enable magnetic tapes to be manipulated. They should not be used to control other devices.

7.9.1 The REWIND statement

Definition	REWIND ldn_1, ldn_2, ldn_3, ldn_4....
where ldn_1, ldn_2... are arithmetic expressions of type integer	

The REWIND statement causes the magnetic tapes whose logical unit numbers are equal to the values of the expressions ldn, to be rewound i.e. to be positioned so that the next record read or written is the first record on the tape.

Examples:

a) REWIND 1

b) REWIND 1, 2, ITAP+7, J*2

7.9.2 The BACKSPACE statement

Definition	BACKSPACE ldn_1, ldn_2, ldn_3, ...
where ldn_1, ldn_2...are arithmetic expressions of type integer	

The BACKSPACE statement causes the magnetic tapes whose logical unit numbers are equal to the values of the expressions ldn, to be backspaced by one logical record, i.e. the tape is moved backwards by one record.

If a tape is currently positioned at the "rewound" position, then the BACKSPACE statement has no effect.

7.9.3 The UNLOAD statement

Definition	UNLOAD ldn_1, ldn_2, ldn_3...
where ldn_1, ldn_2...are arithmetic expressions of type integer.	

The UNLOAD statement causes the magnetic tapes whose logical unit numbers are equal to the values of the expressions ldn, to be:

(i) Rewound
(ii) Dis-engaged

Once a tape is dis-engaged it cannot be accessed again by the program, and it may be physically removed from the tape unit. The most efficient use of Atlas is obtained if a tape is dis-engaged (or UNLOADed) as soon as it is known that it will not be needed again by the program.

7.9.4 The ENDFILE statement

Definition	ENDFILE ldn_1, ldn_2, ldn_3,.....
where ldn_1, ldn_2..are arithmetic expressions of type integer	

The ENDFILE statement causes end-of-file marks to be written on the tapes whose logical numbers are equal to the values of the expressions ldn.

When an end-of-file mark is encountered by a READ operation, reading is terminated and an error message is printed. It is, however, possible to override this feature and to take special action upon encountering an end of file. (see section 10.4). It is not possible to read past (i.e. beyond) an end-of-file marker.

7.10 THE USE OF MAGNETIC TAPES

In Fortran, the practice of over-writing records on magnetic tape is definitely not recommended in cases where later records on the same tape are to be preserved.

If it is desired to modify one or more records on a magnetic tape, then the tape should

be "copied" to a new tape, the changes being made in the process of copying.

7.11 THE OUTPUT STATEMENT

Definition	OUTPUT (ldn, n)list

where

ldn is any arithmetic expression of type integer whose value is in the range $0 \leq ldn \leq 15$

and

n is an unsigned integer constant

and

list is a series of arithmetic expressions, or slashes, separated by commas. (see below).

ldn may be omitted. If omitted its value is taken as zero. The comma must still be present. n may be omitted (with the comma omitted). If omitted, the value of n is taken as 12. If both n and ldn are omitted the OUTPUT statement is written as:

OUTPUT list

which is equivalent to

OUTPUT (0, 12) list

The OUTPUT statement differs from the other Fortran V output instructions in that it is not record-oriented. A new record is not always started when the OUTPUT statement is executed and a record is not terminated by the ending of the OUTPUT statement. If new records are required to be started, then this is explicitly stated by the introduction of slashes into the list, (this is similar to the slashes in a FORMAT statement).

The OUTPUT statement causes the values of the arithmetic expressions in its list to be written on to the output device whose logical number is equal to the value of the expression ldn. The output device must not be a magnetic tape.

The field width allowed for each value is equal to n columns (or 12 if n is not specified). Integer values are written in the same manner as for the conversion In-1, 1X (see section 7.6.4.5). If the value of the integer is too large to fit into n-1 columns, then an asterisk (*) is printed to indicate the overflow.

Real values are written in the same manner as for the conversions Gn-1, 1X (see section 7.6.4.4). The mantissa is normalised to the range $1.0 \leqslant \text{mantissa} < 10.0$ i.e. an exponent is printed if the mantissa would otherwise lie outside this range.

Text variables are written in the same manner as for the conversion A8, n-8X. Text constants (e.g. OUTPUT 'XYZ TABLE') are written out in full, so that every character (including blanks) in the constant is printed. Note that the field width (n) is ignored for text constants.

A double precision value is output as a real number to single precision only; and only the real part of a complex value is output (the imaginary part is not printed).

Logical and Boolean values should not be present in the list of an OUTPUT statement.

7.11 cont Note that if a nonstandard value for the field width n is chosen, the OUTPUT statement may cause some numbers to be split across the end of one line and the start of the next line. This will be the case when 120 is not exactly divisible by n. (120 is the number of printable characters per line). It could also occur when text constants are present in the list (see above).

In addition, it should be noted that the OUTPUT statement, may cause values to be printed on the same line as previously executed PRINT or formatted WRITE statements.

The first character of each line printed by means of an OUTPUT statement is always printed, and is not treated as a carriage control character.

The presence of K consecutive slashes in the list causes K-1 blank lines to be written.

Each list item (including slashes) should be separated from the next item by a comma. If this is not done then the meaning of the list may be ambiguous: -

e.g. OUTPUT X,/Y

prints two values, (X then on a new line Y)

but

OUTPUT X/Y

prints one value i.e. the value of X/Y.

However, OUTPUT X//Y

would not be ambiguous, since X//Y is not a legal arithmetic expression.

If an apparently real variable in fact contains an integer (unstandardised) value, then the OUTPUT statement prints the variable as a real number, followed by the character /.

e.g. -3.0000E03/
This situation could arise as follows:

```
REAL X
EQUIVALENCE (I, X)
I = 6
OUTPUT X
```

The variable X, although real, contains an integer value (6), and is printed as

6.0000/

Other ways in which this situation could arise are described in sections 4.5(10), 4.6 (6), 8.3 (9), and 8.13.2 (5).

If an unsubscripted array name is given in the list then every element of the array is output in the same way as for a 'short list' (see 7.4.1 (10)).

CHAPTER 8 SIMPLE PROGRAM STRUCTURE

A Fortran V program consists of one main program together with any number (or none) of subprograms.

The main program and subprograms may communicate with each other by means of arguments (parameters) and by COMMON or PUBLIC variables.

The main program and its subprograms may call other subprograms provided that the calls are non-recursive. That is, a program may not call itself, directly or indirectly, e.g. If program A calls subprogram B, then subprogram B may not call subprogram A, or any subprogram which calls A.

There are two kinds of subprograms: subroutine and function; in the following discussion the term subprogram refers to both.

Subprograms may be compiled independently of the main program and independently of each other. When the program is to be executed, one main program and all of its associated subprograms (if any) must be present.

Subprograms may be defined by the user, or may be pre-programmed and contained in the system library.

In Fortran V, a powerful new facility has been introduced which enables subprograms to be nested i.e. one program may contain other subprograms. This facility is known as block structure, and is described in Chapter 9. A subprogram which is not nested in another is also called a segment.

8.1 THE END STATEMENT

Definition
END

An END statement must be the physically last statement of all programs or subprograms (including nested subprograms).

The END statement is not executable, but in Fortran V the END statement will effect termination of the program or subprogram in the absence of a RETURN statement.

8.2 MAIN PROGRAMS AND SUBPROGRAMS

8.2.1 Main programs

A main program is comprised of a set of Fortran V statements, the first of which (other than comment lines) is not a subprogram definition statement, namely: -

a FUNCTION statement
or a SUBROUTINE statement
or a BLOCK DATA statement

and the last of which is an END statement. Main programs are also referred to as programs in this manual, and main programs or subprograms are also referred to as routines.

Main programs may contain any Fortran V statements (including FUNCTION and SUBROUTINE statements) except a BLOCK DATA statement. See also Chapter 9.

8.2.2 Subprograms

Subprograms are program units which may be called by other programs and may be in any of the following categories:

(i) Intrinsic (or built -in) function subprograms
(ii) Basic external function subprograms
(iii) Statement function subprograms
(iv) FUNCTION subprograms
(v) SUBROUTINE subprograms
(vi) Library subprograms

Some library subprograms are described in Appendix 7.

8.3 ARGUMENTS (PARAMETERS)

(1) Arguments provide a means of passing information between a subprogram and the program or subprogram which called it.
There are two kinds of arguments:

Actual Arguments, and
Dummy Arguments (or Formal Parameters)

Actual arguments are used in the statement which calls the subprogram (CALL or function reference), whilst dummy arguments are used in the statement which defines the called subprogram (SUBROUTINE, FUNCTION, or statement function).

Dummy arguments are merely "formal" argument definitions and are used to indicate in the called subprogram, the number, the order and the types of the actual arguments being used in the calling program.

(2) Dummy arguments do not actually exist, i.e. no storage is reserved for them, but they do identify to the called subprogram the actual arguments used the calling program or subprogram.

(3) The actual arguments defined by the calling program or subprogram to which a dummy may correspond are:

simple (scalar) variables
array elements (subscripted)
non-subscripted array names
any arithmetic expressions
logical variables or constants
subprogram names

Labels (or assigned labels), may not be given as arguments.

If a subprogram name appears as an argument, then it must be declared in an EXTERNAL

8.3 cont statement (see section 8.9) unless the subprogram name has already been defined in the same segment.

An actual argument list is a series of actual arguments separated by commas and enclosed in parentheses.

(4) A dummy argument itself may be classified within the called subprogram as:-

a scalar variable
an array
a subprogram

A dummy argument list is a series of dummy arguments separated by commas, and enclosed in parentheses.

The table below shows the permissable correspondences between actual and dummy arguments.

ACTUAL	DUMMY		
	Scalar	Array	Subprogram name
Scalar or Array Element	yes	yes*	No
Array name	yes*	yes	No
Expression	yes	No	No
Subprogram name	No	No	yes

*See paragraphs (7) and (8) below.

(5) Within a subprogram, its dummy arguments may be used in the same way as any other scalar, array, or subprogram names, with certain restrictions, namely, dummies may not appear in the following kinds of statements

COMMON
PUBLIC
DATA

and they may not have values assigned in a Type statement.

(Since dummies do not actually exist, the reason for the above restrictions is clear). Furthermore, classification of a dummy as a simple (scalar) variable, an array, or a subprogram name, occurs in the same manner as for other (actual) names, in both implicit and explicit classifications.

(6) Dummy arguments should agree in number and type with the actual arguments to which they correspond.

When a dummy corresponds to a variable in the calling argument list, any reference to the dummy (in the called subprogram) is really a reference to the actual argument (in the calling program or subprogram).

Thus, not only will the dummy have the value which the actual argument had at the time of the call, but any value subsequently assigned to the dummy will actually be assigned to the actual argument, thus effectively returning a result through the argument list.

On the other hand, when a dummy corresponds to an expression or a constant in the actual argument list, the expression merely serves to initialize the value of the dummy, and the value of the dummy should not be changed within the called subprogram.

This is particularly important when the dummy corresponds to an actual argument

8.3 cont

which is a simple constant. If the dummy to which the constant corresponds is assigned a new value, then the value of the constant may also be changed.

(7) Dummy scalars (i.e. simple variables) are single valued entities which have the values of the calling arguments to which they correspond.

Dummies which are not explicitly declared to be arrays or subprograms are treated as scalars.

A dummy argument may be declared to be an array by the presence of its name in an array declaration within the called subprogram.

Since a subprogram may be compiled separately from its calling program, the fact that a calling argument is an array does not of itself define the corresponding dummy to be an array. As with all dummies, a dummy array does not actually occupy any storage; instead, the calling subprogram assumes that the actual argument supplied in the calling statement defines the first (or base) element of an actual array and calculates subscripts based on that location.

(8) Normally, a dummy array should be given the same dimensions as the actual array (or it may be a simple variable) to which it corresponds. This is not necessary, however and sometimes useful operations can be performed by defining different dimensions for the dummy and calling arguments.

```
e.g.    DIMENSION A(10, 10)          SUBROUTINE PART (B)
        CALL PART(A(1, 6))           DIMENSION B (50)
              .                            .
              .                            .
              .                            .
                                     END
```

In this case, the one dimensional array B corresponds to the last half of the two dimensional array A (i.e. elements A(1, 6) through A(10, 10)).

However, since the subprogram assumes that the calling argument defines the first element of an array, if the calling statement were

```
        CALL OUT (A)
or      CALL OUT (A(1, 1))
```

the dummy array B would correspond to the first half of the array A.

Similarly, if an array corresponds to something other than an array, then the latter will correspond to the first element of the array. This is true whether the dummy is an array and the calling argument is not, or vice versa.

Thus, if the calling argument is a scalar, and the dummy is an array, any reference in the subprogram to elements of the array other than the first, will access whatever happens to the stored near the scalar.

Care should be taken when creating correspondences of this nature.

(9) If a dummy has a different type from its corresponding actual argument, then an execution error may occur.

```
e.g.    REAL X                       SUBROUTINE S(I)
        CALL S (X)                         .
        A = B/X                            .
```

```
      I = 3
       .
       .
      RETURN
      END
```

Would cause a division error, because, on return to the calling program, X contains an integer value (3), although X is a real variable.

8.4 FUNCTION SUBPROGRAMS

Function subprograms are programmed procedures which are often used to provide solutions to mathematical functions and are used in a manner similar to that of normal mathematical notation. For example, there is an intrinsic cosine function whose name is COS; thus allowing

$$y = \cos x$$

to be written

```
Y = COS (X)
```

All kinds of function subprograms are referenced in this way.

Function references may be used in the same manner as variable references in any expression.

e.g. `X = (-B+SQRT (B**2 - 4*A*C)) / (2*A)`

Where SQRT is the name of the square root function, and (B**2 - 4*A*C) is the calling argument list.

Associated with each function reference is one value which is returned for the function. Consequently in order that the value returned for a function is of the proper data type the following conventions have been established.

The intrinsic (built-in) and basic external functions are typed automatically by the Fortran V compiler. Tables giving details of these functions are given in Appendix 4.

Functions whose names are not declared to be of any particular type are typed implicitly according to the first letter of the function name (in the same way as for variables) (see 4.3). The IMPLICIT statement is also effective for function names.

Functions defined external to the program or subprogram in which they are defined, which are to be typed other than implicitly (as above), must be explicitly typed; that is, their names must appear in a Type statement in all of the programs or subprograms in which they are referenced.

Statement functions which are not to be typed implicitly must be explicitly typed by the appearance of their names in Type statements in all of the programs or subprograms in which they are defined.

FUNCTION subprograms which are to be typed explicitly must have their type declared in the FUNCTION statements which define them (see section 8.6), or by the appearance of their names in Type statements within the FUNCTION subprograms themselves.

8.4.1 Intrinsic and basic external functions

Intrinsic functions are used to evaluate commonly used mathematical functions, and are supplied by the Fortran V compiler.

Everytime an intrinsic function reference appears, the machine instructions required to evaluate the function are compiled in-line with the instructions for the expression in which the reference was made.

Basic external functions are subprograms which are supplied from the system library and are accessed by standard calling sequences (see Chapter 11). In the source program, basic external functions are referenced in the same way as intrinsic functions.

All other subprograms are called using standard calling sequences including:

FUNCTION subprograms
Statement function subprograms
SUBROUTINE subprograms
Library subprograms

When intrinsic or basic external functions are referenced, the number and type of the arguments must correspond to the table given in Appendix 4.

8.4.2 Names of intrinsic and basic external functions

As will be seen from the above comments, intrinsic functions provide for more efficient execution than external functions.

Many Fortran V functions are available as both intrinsic and basic external functions; and the same function may have several different names.

These different names have been introduced into Fortran V in order to improve compatability with other versions of Fortran. A complete list of available functions is given in Appendix 4.

Three sets of function names are available in Fortran V. These sets of names correspond to the names used in:

Atlas Fortran (Hartran), (see ref. 2)
A.S.A. Fortran (Fortran IV) (see ref. 1)
Fortran II (see ref. 4)

The standard set of names used in Fortran V is the Fortran IV (or A.S.A.) set. The other sets of names may be made standard for any main program or subprogram by insertion of one of the following statements in that main program or subprogram.

Definition	
F4	FUNCTIONS
ASA	FUNCTIONS
F2	FUNCTIONS
OLD	FUNCTIONS
HARTRAN	FUNCTIONS

ASA FUNCTIONS, and F4 FUNCTIONS are equivalent statements.

The inclusion of one of these statements in a program or subprogram causes the compiler to recognise only ASA (or Fortran IV) function names in those statements which follow the FUNCTIONS statement. Other function names will either be recognised as basic external functions or, if not available in this form, a function with the corresponding name must be supplied by the user. (see Appendix 4 for a table of available intrinsic

and basic external functions).

OLD FUNCTIONS, and F2 FUNCTIONS are equivalent statements. Their use, and effect, is the same as for the above paragraph, except that Fortran II names are recognised, and not ASA or Hartran names.

The use and effect of HARTRAN FUNCTIONS is the same as in two paragraphs previous except that Atlas Fortran (Hartran) names are recognised and not ASA or Fortran II names.

Some of the Hartran names are the same as the Fortran II names, but most of the ASA names are different from the other two sets. Most of the external names are the same as the ASA names.

In the absence of a FUNCTIONS statement in any main or subprogram, ASA FUNCTIONS is assumed. Once a FUNCTIONS statement has been given in a main or subprogram it remains effective for all later statements in the program.

The FUNCTIONS statement should appear in the text of the program before any references to system functions.

The type of FUNCTIONS to be used for a whole job may be specified on the *RUN directive (see section 12.1 (9)).

8.5 STATEMENT FUNCTIONS

Statement functions are function subprograms, which are defined in a single expression.

Definition	fname ($name_1$, $name_2$, ... $name_n$) = exp
where fname, and $name_1$, $name_2$..... are variable names and exp is any logical or arithmetic expression.	

fname is the name of the function and must not be the same as the name of any other function, subprogram, or variable of the program or subprogram containing the statement function.

The $name_i$ are the dummy arguments of the statement function, and must be dummy scalars; they must not be dummy arrays or subprograms.

The expression exp should contain at least one reference to each of the $name_i$. Other references in the expression are unrestricted, with the exception that the identifier of the function (fname) may not appear. For example, any other statement function already defined may appear, and subscripted array names may appear.

Examples:
```
G(X, I) = X * B(I, I+7, 3)+4.2
F(X) = A*X**2 + B*X+C
EX (THETA) = CMPLX (COS(THETA), SIN(THETA))
TRUFAL (A, B, C) = A=B=C.OR.A<B<C
```

Since each $name_i$ is a dummy and does not actually exist, the $name_i$ may be the same names as other names in the main or subprogram (except those referenced in the expression exp), without conflict. However, if a dummy is explicitly typed by the presence of its name in a type statement, any other use of that name will have the same data type.

If a statement function is to be explicitly typed, then its name (fname) must appear in a

type statement before the statement function appears.

A statement function may be referenced only in the main or subprogram in which it appears unless its name (fname) is given as an actual argument after the function has been defined. In this case the function name should not appear in an EXTERNAL statement.

Statement function definitions must precede all references to the functions in executable statements in the main or subprogram in which they appear.

8.6 THE FUNCTION STATEMENT

Functions which cannot be defined in a single statement may be defined as FUNCTION subprograms. These programs begin (other than comment lines) with a FUNCTION statement.

Definition
type FUNCTION fname ($name_1$, $name_2$....$name_n$)
where type is either not present or is one of: - INTEGER REAL DOUBLE PRECISION or DOUBLE LENGTH COMPLEX LOGICAL TEXT BOOLEAN and fname, $name_1$, $name_2$, ... are variable names

fname is the name of the function and must not be the same as the name of any other function, subprogram, or variable of any main or subprogram in the same job. The function name (fname), is public, and may be referenced by any main or subprogram.

The $name_i$ are the dummy arguments of the function and may take any of the forms described in section 8.3.(4). If a dummy argument is a subprogram name, then the corresponding actual argument must be declared in an EXTERNAL statement within the calling program.

The dummy arguments, $name_i$, may be used for any purpose within the FUNCTION subprogram, with the exception described in section 8.3.(5).

A FUNCTION subprogram must have at least one dummy argument; and must contain at least one RETURN statement.

Within the FUNCTION, the name (fname) is treated as though it were a scalar variable, and may be used like any other scalar, but fname should normally be assigned a value for each execution of the FUNCTION. The value returned for a FUNCTION is the last value assigned to its name (fname) prior to the execution of a RETURN statement.

Example:

```
      .                          FUNCTION MAXIMUM (L, M)
      .                          MAXIMUM = L
      .
I = MAXIMUM (J, K)               IF (L>M)GOTO 2
```

```
.                    MAXIMUM = M
.                  2 RETURN
.                    END
```

Where I is set to the larger value of J or K. The IF statement could also take the form

```
IF (MAXIMUM>M) GO TO 2
```

The data type of the FUNCTION fname may be explicitly declared in the FUNCTION statement itself, or may be declared in a type statement within the subprogram. If the latter form is used, the type statement must precede the first reference to the name (fname) in any executable statement within the subprogram.

e.g. LOGICAL FUNCTION T(A)

is equivalent to

```
FUNCTION T (A)
LOGICAL T
```

If the type of fname is not explicitly declared, then it will be implicitly typed as described in section 8.4.

The type of a FUNCTION subprogram should correspond to its use in any calling programs.

A FUNCTION subprogram may include any Fortran V statements (including other FUNCTION and SUBROUTINE statements), except a BLOCK DATA statement. See also Chapter 9.

A FUNCTION subprogram may change the values of its argument(s), or of variables contained within PUBLIC or COMMON storage. This is known as a side-effect of the function.

Warning: In the evaluation of expressions, no attempt is made to provide for side-effects of functions. Therefore, functions called in an expression should not change the values of any variables appearing in the expression.

Note: care should be taken to ensure that the type of the function is the same in the function itself, and in its call.

e.g.

```
REAL X                    INTEGER FUNCTION X(J)
   .                             .
   .                             .
   .                             .
A = B/X(J)                END
```

would cause an error, since an integer value would be returned for X, although it is declared as real in the calling program.

8.7 THE SUBROUTINE STATEMENT

SUBROUTINE subprograms, like functions, are self contained programmed procedures. However, unlike functions, subroutines do not have values (and hence not types) associated with their names, and they are not referenced in expressions. Instead, subroutines are accessed by means of CALL statements (see section 8.10.)

Subroutine subprograms begin (other than comment lines) with a SUBROUTINE statement.

Definition
SUBROUTINE sname ($name_1$, $name_2$.....$name_n$) or SUBROUTINE sname
where sname is a variable name, which is the name of the subroutine and $name_1$, $name_2$... $name_n$ are variable names.

sname is the name of the subroutine, and the $name_i$ (if any) are its dummy arguments. The dummy arguments may take any of the forms described in section 8.3.(4). If a dummy argument is a subprogram name, then the corresponding actual argument must be declared in an EXTERNAL statement within the calling program.

The dummy arguments may be used for any purpose within the subroutine, with the restrictions described in section 8.3.(5).
If the SUBROUTINE has no arguments, then the second definition is used.

A subroutine subprogram should normally contain at least one RETURN statement, unless execution is to be terminated within the subroutine. The RETURN statement(s) should be positioned so that it is the last statement executed for each execution of the subroutine.

Subroutine subprograms may return values to the calling program or subprogram(s) by assigning values to the $name_i$, or by changing the values of COMMON or PUBLIC variables.

Subroutine subprograms may contain any Fortran V statements (including other SUBROUTINE or FUNCTION statements), except a BLOCK DATA statement. See also Chapter 9.

Examples:

a)
```
      SUBROUTINE PRINT
```

b)
```
      SUBROUTINE READ ARRAY (A,B,C)
```

c)
```
      SUBROUTINE OUTPUT (A)
      DIMENSION A (100)
      PRINT 1, A
    1 FORMAT (9H1TABLEb2., 10(1X, F6.3))
      RETURN
      END
```

Which prints the array corresponding to A, starting on a new page, with a heading, and with 10 numbers to a line.

8.8 ADJUSTABLE DIMENSIONS

Note: Dynamic Arrays are described in Chapter 9.

Since a dummy array does not actually occupy any storage, its dimensions are used only to locate its elements, and not to allocate storage for them. Therefore, the dimensions of a dummy array need not be defined in the called program in the normal way. Instead any (or all) of the dimensions of a dummy array may be specified by means of scalar

8.8 cont variables rather than by constants. This permits the calling program to supply the (adjustable, or variable) dimensions of the dummy array each time the subprogram is called.

The absolute dimensions of an array must be declared in a calling program. The magnitudes of the adjustable dimensions of an array, declared in the called subprogram, should be less than or equal to the absolute dimensions of that array, as declared in the calling program. Adjustable dimensions cannot be used in a main program.

The adjustable dimensions may be passed to the called subprogram as

(i) an argument
or (ii) a COMMON variable
or (iii) a PUBLIC variable

The adjustable dimensions declared in the called program may be arithmetic expressions comprised of integer constants and scalar variables whose values are defined as in (i), (ii), and (iii), above. All specifications for variables used in the dimensions must precede the dimensioning statement.

The name of the adjustable dummy array itself must be a dummy argument, and must not be in COMMON or PUBLIC.

Note that the last dimension of a multi-dimensional dummy array is not arbitrary in Fortran V.

The last dimension of a multi-dimensional array must be as large as the largest value used for the subscript.

The use of adjustable dimensions means that the definition of the DIMENSION statement given in section 4.2.1 has to be extended to allow the dimensions to be any arithmetic expressions of type integer, rather than merely integer constants. Any variables used in such expressions must be previously defined as above.

The definitions of the type statements given in section 4.4 are similarly extended, but when adjustable arrays are declared in type statements, no initial data values may be assigned to the array by means of the type statement.

Adjustable arrays must not appear in any EQUIVALENCE, COMMON, PUBLIC, or DATA statements.

Adjustable dimension specifications of an array must appear after other specifications for the array. In no case can the length of the array be changed after its dimensions have been specified as adjustable.

e.g.
```
DIMENSION D(M,N)
COMPLEX D
```
would be in error.

It should be specified as

```
COMPLEX D
DIMENSION D(M,N)
```

or (preferably) as

```
COMPLEX D(M,N)
```

Examples:

a)

```
INTEGER A(10, 10)
      .
      .
I = 9
      .
      .
J = 2
      .
CALL S(A, I, J)
      .
```

```
SUBROUTINE S(B, L, M)
INTEGER B(L, M)
      .
      .
      .
      .
      .
      .
RETURN
END
```

b)

```
DIMENSION X(5, 6)

I = 4
  .
  .
  .
A = SUM (X, 4, I)
```

```
FUNCTION SUM (X, L, M)
DIMENSION X(L, M+2)
      .
      .
RETURN
END
```

c)

```
DIMENSION A(10, 10)
PUBLIC I
COMMON J
  .
  .
  .
  .
  .
I =
  .
  .
J =
  .
  .
CALL X (A)
  .
```

```
SUBROUTINE X(A)
PUBLIC I
COMMON J
DIMENSION A (I/3, J*4 - 6)
      .
      .
      .
      .
      .
      .
      .
      .
      .
      .
RETURN
END
```

8.9 THE EXTERNAL STATEMENT

As described in section 8.3.(3) (see also Chapter 9) "actual" arguments which are subprogram names must be declared in an EXTERNAL statement.

Definition
EXTERNAL $name_1$, $name_2$, ... $name_n$
where $name_1$, $name_2$..... $name_n$ are variable names.

Each name appearing in an EXTERNAL statement is explicitly defined to be the name of a subprogram.

Intrinsic function names which appear in EXTERNAL statements cause the names to be treated as external function references rather than instrinsic function references.

Specifically, intrinsic function names which appear in an EXTERNAL statement, may be passed as subprogram arguments. Not all of the intrinsic functions are available as external functions (see Appendix 4). In such cases the user must provide his own function of that name.

EXTERNAL statements are not executable, and must appear in the text of the program before any reference to the names which are to be treated as external.

Examples:

a)
```
EXTERNAL SUM, SUBX, SIN, COS
```

b)
```
EXTERNAL COPY                    SUBROUTINE S(J, DUM)
      .                                .
      .                                .
CALL S (I, COPY)                 CALL DUM (A, B, C)
      .                                .
      .                                .
      .                          RETURN
      .                          END
```

In Subroutine S the CALL statement will, in fact, call subprogram COPY.

c)
```
      .                          SUBROUTINE SUBR (P, Q, R)
      .                                .
EXTERNAL Z                             .
      .                          C = Q(P, R*3)
CALL SUBR (A, Z, B)                    .
      .                                .
      .                          RETURN
      .                          END
```

The replacement statement C = Q(P, R*3) actually accesses function Z.

If the CALL statement were (say)

```
CALL SUBR (A, Z(S, T), B)
```

then the EXTERNAL statement would be incorrect, because function Z is not now an argument: it is executed first, and the result becomes the argument. Also, Q could not now be used as a function.

8.10 THE CALL STATEMENT

Definition
CALL sname (a_1, a_2, a_3, ...a_n) or CALL sname
where sname is a variable name, which is the name of a subroutine. and a_1, a_2, a_3, ...a_n are each arithmetic or logical expressions, or subprogram names.

The CALL statement causes control to be transferred to the first executable instruction of the SUBROUTINE whose name is sname.

The a_i are the actual arguments to be passed to the called subroutine, and are described in section 8.3. If the called subroutine requires no arguments then the argument list is omitted, as in the second definition above.

Arguments appearing in a CALL statement may be

constants (of any type)
simple (scalar) variables
array elements (subscripted)
Array names (non-subscripted)
Arithmetic expressions
or subprogram names (excluding statement names)

If a subprogram name is used as an argument, then this name is not followed by an argument list, since this argument form is only used to provide the called subroutine with a subprogram reference. In this sense, the subprogram reference is merely a name, and as such, has no value associated with it. (But see the note in example c) above).

Furthermore, when a subprogram name is used in this manner, it must appear in an EXTERNAL statement which is given in the text of the program before the CALL statement(s) in which it is used, (unless the subprogram is a sub-block or statement function which has already been defined in the segment: (see Chapter 9).

Examples:

a) CALL MATRIX (A,B,C/D+43.2)

b) CALL S47T (1,2,X*Y)

c) CALL OUTPUT ARRAYS (A(4), B(I+J), 'TABLE 47')

8.11 THE RETURN STATEMENT

Definition
RETURN

The RETURN statement causes control to be transferred from a subprogram back to the main or subprogram which called it.

If the called program is a SUBROUTINE, then the RETURN statement causes control to be transferred back to the first executable statement following the corresponding CALL statement.

In the case of a function subprogram a return occurs to the evaluation of the expression in which the function was referenced.

8.12 THE PUBLIC STATEMENT

Normally, except for external or subprogram names, any names of variables used in a program or subprogram have no connection with names used in other subprograms. i.e. the same name does not mean the same variable.

Use of the PUBLIC statement enables variables to be referenced by name in more than one program or subprogram.

Definition
PUBLIC $name_1$ (i_1, i_2, ...) $name_2$ (i_3), $name_3$...
where $name_1$, $name_2$, ...are variable names and i_1, i_2, i_3, ...are unsigned integer constants.

When a variable name is placed in PUBLIC statements in more than one program or subprogram, all references to that name in those programs or subprograms will access the identical data item.

The variable name should be classified identically, in those programs or subprograms, in respect of dimensions and type.

The dimensions (not adjustable) of arrays may be declared in the PUBLIC statement; when subscripts appear in the list, the associated name is the name of array, and the subscripts (i) are the dimensions of the array.

A public variable is only public to those main or subprograms which contain a PUBLIC statement containing the name of that variable.

Within one main or subprogram, a PUBLIC statement must not contain any names which are:

(i) Declared in a COMMON statement
(ii) Labelled COMMON block names
(iii) The same as other (different) variables or function names appearing in the same program.
(iv) Declared in an EXTERNAL statement.
(v) Assigned initial data values by means of DATA or type statements, unless these statements appear in a BLOCK DATA subprogram. See section 8.14.
(vi) In the dummy argument list (if any) of the subprogram.

The PUBLIC statement is not executable, and must appear in the text of the program before any reference to names contained in the statement. (See Appendix 3)

If a public variable is also declared in a DIMENSION and/or a Type statement, then the order of the PUBLIC/DIMENSION/Type statements is immaterial. If a public variable also appears in an EQUIVALENCE statement, then the PUBLIC statement must appear first.

Example: PUBLIC MATX (3, 6, 7), THETA, A(4)

8.13 THE COMMON STATEMENT

The COMMON statement is used to assign data to a particular region of storage called COMMON storage. Since this area of storage is fixed, the COMMON statement provides a means by which more than one program or subprogram may reference the same data.

Definition	COMMON c_1, c_2, c_3, ...c_n
where each c_i is of the form	
	/bname/, v_1, v_2, v_3...v_m
or	//, v_1, v_2, v_3.....v_m
or	v_1, v_2, v_3.....v_m
and	
	bname is a variable name
and	
	each v_i is a subscripted or non-subscripted variable name. Any subscripts must be unsigned integer constants. The commas following the second slash of each pair of slashes are optional.

8.13 cont Examples:

a) COMMON A,B(4,5), MAT2

b) COMMON TA, B3/XX/C(5),D//E, F(6)

c) COMMON /NB5/X, Y/XX/K

(1) The dimensions (not adjustable) of arrays may be declared in the COMMON statement. When subscripts appear in the list, the associated name is the name of an array, and the subscripts are the dimensions of the array.

(2) The COMMON statement is not executable, and must appear in the text of the program before any reference to the variables in its list. (See Appendix 3).

If a COMMON variable appears in a DIMENSION and/or a Type statement, then the order of the COMMON/DIMENSION/Type statements is immaterial.

If a COMMON variable also appears in an EQUIVALENCE statement, then the COMMON statement must appear first.

(3) Each bname is the name of a labelled COMMON block, and the list of names that follows it contains the names of variables or arrays which are to be placed in that block. If no bname is specified (as in example a), or if a blank name is specified (as in example b) the variables in the following list are placed in blank (or unlabelled) COMMON storage.

(4) Labelled COMMON blocks are discrete sections of the COMMON region, and are thus independent of each other, and of unlabelled COMMON.

In the examples above, A,B,MAT2,TA,B3,E and F are in unlabelled COMMON; C,D and K are in COMMON block XX; and X and Y are in COMMON block NB5.

(5) A COMMON variable may not appear in:

A PUBLIC statement
A dummy argument list

A labelled COMMON variable may only appear in a DATA statement (or have initial data values assigned in a type statement), when these statements are contained within a BLOCK DATA subprogram. (See 8.14.)

A variable which is in unlabelled COMMON may not have initial data values assigned in a Type statement, and may not appear in a DATA statement.

(6) Any labelled COMMON block may be referenced by any number of programs or subprograms. References are made by block names (bname), which must be identical if it is desired to reference the same COMMON block (i.e. bname is effectively a public name).

All labelled COMMON blocks need not be defined in any one program or subprogram; only those blocks containing data required by the program or subprogram need be defined.

(7) The variables defined as being in a particular COMMON block do not necessarily have to correspond in type or in number between the programs or subprograms in which the block is referenced. This is also true of unlabelled COMMON. See also section 8.13.2(5).

However, the definition of the overall length of a COMMON block must be the same in all of the programs or subprograms in which it is defined.

Example:

```
FUNCTION F(I)                    SUBROUTINE X
                                 DOUBLE LENGTH P(20)
COMMON /BB/ A(40)                COMMON /BB/ P
```

Both references to block BB correspond in size. The array A (not being explicitly declared) is of type real and hence is 40 words in length. The array P being of type double precision, is also 40 words in length.

(8) Reference may be made to the name of a labelled COMMON block more than once in any program or subprogram. Multiple references may occur in the same COMMON statement; or the block name may be specified in any number of individual COMMON statements.

In both cases, the compiler links together all variables defined as being in the block into a single labelled COMMON block of the appropriate name.

The variables are linked together in the order in which they appear.

(9) Block names (bname) must not be the same as: -

names which are contained in PUBLIC statements.
or subprogram names.

Block names do not conflict with names other than the above.

(10) Blank, or unlabelled COMMON is an area of COMMON storage which is not discrete, although it is separate from the block region; i.e. there is only one such area, and empty block name specifications always refer to it.

In addition, as opposed to labelled COMMON, blank COMMON areas defined in various programs and subprograms need not correspond in size.

Example:

The following two subprograms define blank COMMON areas of different lengths, and yet both may be portions of the same executable program.

```
FUNCTION Z(B)                    SUBROUTINE K
COMMON A (50), E, F              COMMON TT (100)
      .
      .
```

Subprogram Z defines a COMMON length of 52 words, and a length of 100 words is defined in K.

(11) Reference may be made to blank COMMON any number of times within a program or subprogram. The multiple references may occur in a single COMMON statement, or in several individual COMMON statements. In both cases, all variables defined as being in blank COMMON, are linked together and placed in the blank COMMON area. The variables are linked together in the order in which they appear.

8.13.1 Arrangement of COMMON

Each separate COMMON block, and the unlabelled COMMON area, contain, in the order of their appearance, the variables declared to be in that block, or area.

The variables in each section of the COMMON region are arranged from low-address storage towards high-address storage. The first variable declared as being in a particular section is placed the low-address word(s) of that section; succeeding variables are placed in higher addresses; until the last variable declared to be in the

8.13.1 cont

section is placed in the highest address word(s) of the section.

Array variables are stored in the normal manner (see section 4.2.2) within the COMMON region.

Examples:

a) The statements

```
COMMON A,B/XX/C(4)
COMMON /XX/E(2,2)//D(3,2)
```

produce the following arrangement of COMMON storage,

Item	Block XX	Unlabelled COMMON
1	C(1)	A
2	C(2)	B
3	C(3)	D(1,1)
4	C(4)	D(2,1)
5	E(1,1)	D(3,1)
6	E(2,1)	D(1,2)
7	E(1,2)	D(2,2)
8	E(2,2)	D(3,2)

b) The statements:

```
SUBROUTINE ASUB
REAL I, K(4), PI
COMPLEX Z (4), ROOT
COMMON Z/BB/I
COMMON/BB/K//PI
```

```
SUBROUTINE BSUB
COMPLEX T (2)
REAL Q (8), I
COMMON Q/BB/T,I
```

Produce the following arrangement:

Item (word)	Block BB		Unlabelled COMMON	
	in ASUB	in BSUB	in ASUB	in BSUB
1	I	T(1)	Z(1)	Q(1)
2	K(1)	T(1)	Z(1)	Q(2)
3	K(2)	T(2)	Z(2)	Q(3)
4	K(3)	T(2)	Z(2)	Q(4)
5	K(4)	I	Z(3)	Q(5)
6			Z(3)	Q(6)
7			Z(4)	Q(7)
8			Z(4)	Q(8)
9			PI	

Each COMPLEX item requires two words.

In BSUB a reference to T(1) will access the words in which I and K(1) are stored; similarly, a reference to I will access K(4).

Putting item 9 (PI) into COMMON is not useful unless another program or subprogram exists which defines at least 9 words of blank COMMON.

Note: Each labelled COMMON area begins at the start of an Atlas block of 512 words. Use of many short (less than 512 words) labelled common blocks may waste store on Atlas.

8.13.2 COMMON/EQUIVALENCE interaction

(1) The EQUIVALENCE statement is described in section 4.5.
No storage allocation declaration is permitted to cause conflict in the arrangement of storage.

(2) Each COMMON, PUBLIC and EQUIVALENCE statement determines the allocation of the variables declared in them. Therefore, no EQUIVALENCE set may contain references to more than one variable (or more than one element of one array) which has been previously allocated (or referenced). Similarly, COMMON or PUBLIC statements should not contain the names of any variables which have previously been declared to be PUBLIC or in COMMON.

When the above rule is violated Fortran V accepts the first reference, and rejects (with an error message) all later (illegal) references.

(3) It is sometimes permissable for an EQUIVALENCE statement to cause a segment of the COMMON region to be lengthened beyond the last item defined to be in that segment. It is not, however, permissable for an EQUIVALENCE statement to cause a segment to be extended beyond the first item declared to be in that segment.

Example:
```
COMMON/AB/I(5), J/BB/K(4), L
DIMENSION P(8), Q(5)
EQUIVALENCE (I, P), (Q(4), K(2))
```

The first EQUIVALENCE set is a permissable extension of the block AB; the second set illegally defines an extension of block BB. If the illegal extension were carried out (it would not be), the storage arrangement would be: -

Item	AB	BB
-	-	Q(1)) illegal
-	-	Q(2))
1	I(1)=P(1)	K(1)=Q(3)
2	I(2)=P(2)	K(2)=Q(4)
3	I(3)=P(3)	K(3)=Q(5)
4	I(4)=P(4)	K(4)
5	I(5)=P(5)	L
6	J=P(6)	
7	P(7)	
8	P(8)	

(4) The fact that COMMON blocks may be lengthened by EQUIVALENCE declarations in no way nullifies the requirement that labelled COMMON blocks of the same name must be of the same length in all of the programs or subprograms in which they are defined.

(5) Note: care should be exercised when variables of different types are made equivalent either by EQUIVALENCE or by COMMON or PUBLIC statements. For example, if a variable, which appears to be of type REAL, in fact contains an INTEGER value, then the wrong instructions may be compiled.

e.g.
```
SUBROUTINE A          SUBROUTINE B
COMMON I              COMMON X
I = 4                 Y = 1/X
CALL B                   .
   .                     .
   .                     .
```

would cause an execution error is SUBROUTINE B, since a floating point division would be compiled, and the value contained in X is, in fact, an integer.

8.14 THE BLOCK DATA STATEMENT

As mentioned in sections 8.12, and 8.13.(5), initial data values may be assigned (by means of DATA or Type statements) to PUBLIC or block COMMON variables, only in a BLOCK DATA subprogram.

Definition
BLOCK DATA

The BLOCK DATA statement is the first statement (other than comment lines) of a BLOCK DATA subprogram.

The BLOCK DATA subprogram may contain only the following kinds of statements

IMPLICIT
DIMENSION
any Type Statements (in which values may be assigned)
PUBLIC
COMMON
DATA
Comment Lines,

and the last statement of the subprogram must be an END statement.

Since the BLOCK DATA subprogram has no name, it may not be called by any other program or subprogram; it is not executable, and may not contain any executable statements.

The only purpose of the subprogram is to assign initial data values to block COMMON or PUBLIC variables.

Data may not be assigned to variables which are in blank common.

All items in a COMMON block must be given, even though they might not appear in the type or DATA statement(s).

Data may be entered into any number of COMMON blocks, or PUBLIC variables, in one BLOCK DATA subprogram.

Any number of BLOCK DATA subprograms may appear in one job, although there is never need for more than one.

Example:
```
BLOCK DATA
COMMON/X/A, B(10), C, D, I(5)
INTEGER B
PUBLIC P, Q, R(3)

REAL I(5)/5*1.0/
DATA (B(K), K=1, 10), C, P, Q, R/10*0, 6*1.0/
END
```

The values of A and D are not defined.

CHAPTER 9 PROGRAM BLOCK STRUCTURE AND DYNAMIC ARRAYS

9.1 INTRODUCTION TO BLOCK STRUCTURE

Fortran V program block structure is a generalisation of ordinary Fortran subprogram structure (see Chapter 8). It is very similar to the program block structure of languages like Algol or PL/1.

Normally, Fortran programs are broken up into segments, consisting of a main program, and function or subroutine procedures. Compilation of each segment proceeds independently. Names of variables and labels are private to each routine, and do not conflict with use of the same names or statement numbers in other segments. Communication between segments is effected by the COMMON and PUBLIC statements for variables, and by special provision for making procedure names public.

This segmental structure is very convenient for dividing a large program into manageable pieces, but it does have certain disadvantages:

(i) Tedious book-keeping may be required, such as the repetition of long COMMON declarations in many segments.

(ii) Names are either private or fully public causing difficulties in segmenting routines designed for general use.

(iii) Entry of subsections must take the form of procedure calls.

These disadvantages may be overcome by the use of block structure.

In Fortran V, outer segments are still compiled separately. However, within such a segment, program blocks may be written which contain variables and labels (statement numbers) that are private or local, to the block.

Since the entire segment is compiled as a unit, information about variables and labels of the outer block is available when compiling the inner block. These variables or labels are global to the sub-block (or inner block).

9.1.1 An example of block structure

FUNCTION or SUBROUTINE statements can appear within a routine, and indicate the start of a procedure sub-block. Consider the following example:

```
SUBROUTINE OUTER
DIMENSION A (10)
INTEGER N
      .
      .
FUNCTION POLY (X)
INTEGER I
POLY = A(1)
IF (N.EQ.1) GO TO 2
```

```
      DO 1 I = 2,N
1     POLY = X*POLY + A(I)
2     RETURN
      END
      DO 1 I = 1,10
            .
            .
      Y = POLY (Z)
1     CONTINUE
2     Z = Z-1
            .
      IF (Z) 2,3,2
3     RETURN
      END
```

The inner procedure POLY is a sub-block function for evaluating the polynomial with coefficients stored in A. Note that A and N are global to POLY, but I is local to POLY, and has no connection with the variable I used in the DO loop after the first END. This statement is the first executable statement of OUTER. The statement numbers 1 and 2 of POLY are also local to it.

9.1.2 Use of block structure

Block structure is particularly useful for large programs, or for writing sections of programs to be inserted into other programs. It will frequently be found that the use of block structure economises on storage space, without loss of running time.

The compound logical IF statement (9.4) is of general use, and provides some of the flexibility of Algol compound IF statements. When storage space requirements are crucial, or difficult to estimate, dynamic arrays (9.5) may be found useful.

A brief summary of block structure is given in 9.7.

9.2 BLOCK STRUCTURE DEFINITIONS

9.2.1 Program blocks

(1) A block heading is one of the following:

- (i) a SUBROUTINE statement
- (ii) a FUNCTION statement
- (iii) a BEGIN statement (see 9.2.2)

(2) A program block consists of all statements between a block heading and a matching END statement. Block headings and END statements match up in the same way as left and right brackets do in arithmetic expressions. When an END statement is encountered in compilation, it is matched with the last block heading not already matched by an END statement. (See example 9.1.1.)

The block heading is not considered as part of the program block (9.3.2).

A program block with a FUNCTION or SUBROUTINE heading is a procedure block.

(3) It follows from the definition that two program blocks are either disjoint, that is, do not overlap, or that one is contained (i.e. nested) within the other. Program blocks may be nested to any depth. If a program block A is contained in a program block B, then A is called an inner block of B, and B is called an outer block of A.

(4) A program block not contained in another block is called a program segment, or subprogram (or main program - see Chapter 8). Program segments are compiled independently. Object cards, when specified, are produced only for program segments, including cards for any inner blocks of the segment, (i.e. cards cannot be produced for an inner block without its containing block(s)).

Program segments can be subroutines, functions, or a main program.

Note: A main program which contains any inner blocks must start with a BEGIN statement. Otherwise, the first END statement encountered will terminate the program segment.

9.2.2 The BEGIN statement

Definition
BEGIN

The BEGIN statement is an executable statement, and can be labelled. It serves as the block heading for a BEGIN block. BEGIN blocks are terminated by an END statement which matches the BEGIN statement.

BEGIN blocks differ from procedure blocks only in the way the block is entered and left.

9.2.3 Entering and leaving blocks

BEGIN blocks are entered when control reaches the BEGIN statement, and left when control reaches the END statement. A RETURN statement in a BEGIN block means a return from the procedure block containing the BEGIN block.

A procedure block is entered by a call, i.e. a CALL statement for a subroutine block, or use in an expression for a function block. Procedure blocks are left by a RETURN statement. Control in a procedure block should normally never reach the END statement, but in Fortran V procedures, if control reaches the END statement, a RETURN is executed.

A procedure block cannot be entered by control reaching the block heading. If control apparently reaches the heading, then the first executable statement after the END statement of the procedure block will be executed.

Transfers into any block are not allowed.

Transfers out of a block are permitted (see 9.3.2).

Differences between Procedure and BEGIN blocks

	Procedure	BEGIN
Block entered by:	CALL or use in expression	control reaching BEGIN
If control reaches heading:	go to statement after end of block	enter block
Block left by	RETURN	control reaching END
Effect of RETURN	return from block	return from outer procedure.

9.3 GLOBAL AND LOCAL ITEMS

Privacy for variables, labels and procedures is provided by block structure.

Essentially, a block cannot refer to items that are local to an inner block, or local to a completely separate block. On the other hand, a block can refer to entities of a block in which it is embedded (or nested).

Definition
An item (variable, label or procedure) is local to a program block if it is declared within the block, i.e. inside the block, but not inside an inner block. An item is global to a program block if it is not local to the block but is local to an outer block.

Global and local are relative terms. An item is local to the block in which it is declared, but global to an inner block.

The definition of a declaration depends on whether the item is a variable, a label, or a procedure; these definitions are given below.

9.3.1 Variables

(1) A variable is (explicitly) declared in a block when it appears in a specification statement (see Appendix 3), or is a dummy argument of the block.

Example:
```
SUBROUTINE SUBA (W)
REAL X, Y (100)
INTEGER I, N
DIMENSION A(5), B(10)
COMMON C, D, E
SUBROUTINE SUBB(X)
DIMENSION C(10)
INTEGER B
J = I
V = X
D = X
W = Y
RETURN
END
INTEGER J
    .
    .
    .
END
```

The variables B and C are local to SUBB, and are therefore quite different from the variables B and C of SUBA. Note that none of the properties of a variable in an outer block apply to a local variable of the same name in an inner block. In the example, the integer variable B of SUBB is not an array, although the real variable B of SUBA, is dimensioned. The variable X is also local to SUBB, since it is a dummy argument.

The variables I, D, W and Y are global to SUBB, since they are not explicitly declared in SUBB.

(2) In Fortran V, the order in which declarations appear is important. The scope of a declaration extends over the program text from the line in which it appears to the end of the program block.

Thus, in the example above, the variable J is not global to SUBB, since SUBB does not lie within the scope of the declaration.

(3) Fortran allows the use of implicit variables, that is, variables which do not appear explicitly in a declaration. In Fortran V an implicit variable is considered to be implicitly declared when it is used in any executable or DATA statement. The scope of an implicit declaration extends over the program text, from the first executable or DATA statement in which the variable is used, to the end of the block containing the statement.

Thus, in the example above, the variable V will be considered to be local to SUBB, since it is implicitly declared in the statement

```
V = X
```

Note, however, that if a variable with the name V had been used in the text before the statement SUBROUTINE SUBB(X) this statement would have been taken as a reference to the global V.

In order to avoid confusion, variables that are desired to be local to a block should be explicitly declared before any executable or DATA statements in the block.

(4) Local variables that are not dynamic arrays (9.5) are assigned fixed storage locations. Values given to the variables are still available when the block is re-entered.

Example:
```
BEGIN
INTEGER N
DATA N/0/
N = N+1
   .
   .
END
```

N is increased by one each time the block is entered.

9.3.2 Labels (statement numbers)

Labels are declared in a block if the label is attached to a statement, i.e. appears in columns 1 through 5 of a statement. The scope of the label declaration extends over the whole block.

If a global label, i.e. a label not declared in the block, is referred to in a control statement (in an inner block), then control jumps out of the block. Control will jump to a label with the same name or number (as that in the control statement) in the nearest outer block in which the label is declared.

Example:
```
      DO 3 I = 1,100
               .
               .
               .
      IF (A(I)) 4,5,2
 2    BEGIN
      B = A(I)
 3    C = B/A(I)
21    IF(ABS(A(I) - C) -1.0E-7)5,5,2
 2    A(I) = 0.5*(A(I)+C)
      GO TO 3
      END
 4    A(I) = 0
 5    ---
```

(Example continued overleaf)

```
      .
      .
      .
3     CONTINUE
```

This example illustrates several aspects of label declarations.

(i) The range of the DO loop includes the BEGIN block and terminates on the second statement number 3 (i.e. CONTINUE). The DO loop does not terminate on the first statement number 3 since that statement is in the inner block.

(ii) DO loops starting in an inner block must terminate before the END statement for that block.

(iii) The first statement number 2 is local to the outer block, although it is attached to the BEGIN statement for the inner block. This illustrates the point raised in 9.2.1 that a block heading is not inside the program block it heads.

(iv) Transfer statements may be written before or after the declaration of the label. In statement number 21, the transfer to statement number 2 goes to the next statement, which has the statement number 2 that is local for the inner block. Similarly, the statement

GO TO 3

transfers to the first statement number 3.

Labels attached to FORMAT statements follow the same declaration rules as other labels. An input/output statement may refer to a FORMAT in an outer block, provided there is no FORMAT with the same statement number in the inner block.

*9.3.3 Assigned GO TO statements

Assigned GOTO statements can be used to cause a transfer out of a program block. They cannot cause a transfer into a program block. A statement of the form

ASSIGN l_1 TO l_2

is treated as a declaration for l_2, which is therefore local to the block.

Examples:

a)

```
      .
      .
      .
      FUNCTION SQT(X)
      IF  (X)  1,2,2
1     GO TO N
2     SQT  =  SQRT  (X)
      RETURN
      END
       .
       .
       .
      ASSIGN 2 TO N
      Z  =  -B+SQT(DISC)
       .
       .
```

```
 2    PRINT 121
12    FORMAT (13H  Z  IS  COMPLEX)
```

If DISC is negative, control transfers out of the function to the outer block statement number 2 (PRINT).

b)
```
      ASSIGN  1  TO  L
      ASSIGN  2  TO  M
      BEGIN
      IF  (I)  21,  22,  23
21    GO  TO  L
22    GO  TO  M
23    ASSIGN  3  TO  L
      END
 2    GO  TO  L
```

Since an ASSIGN statement for L has appeared in the inner block, L is a local label for the inner block, and is not the same as the label L of the outer block.

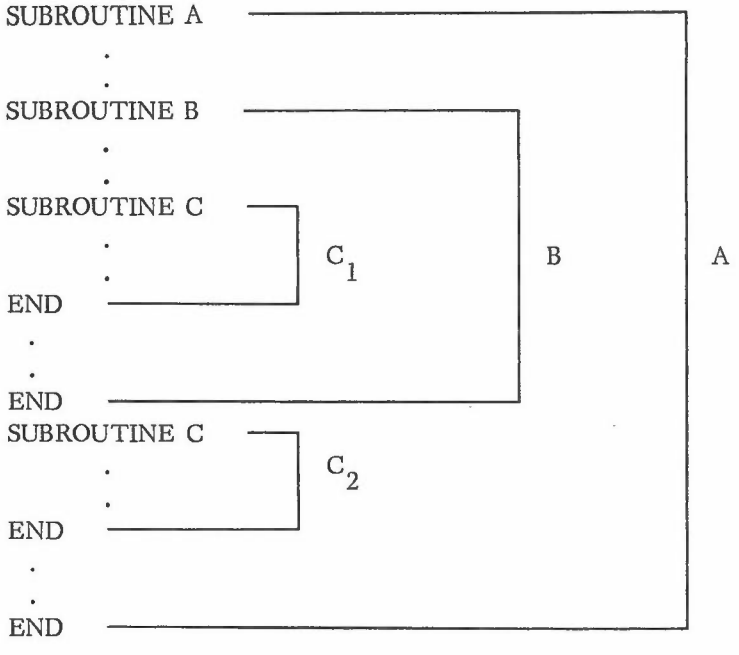
At statement 21, L is unassigned and the GOTO will cause an error. At statement 2, the GOTO refers to the outer L, which is still set to jump to statement number 1, even if statement 23 has been executed.

9.3.4 Procedures

A subroutine or function procedure is declared in a block by the appearance of a SUBROUTINE or FUNCTION statement in the block. From the definition in 9.2.1 the SUBROUTINE or FUNCTION statement starts an inner block, but is itself in the outer block.

The scope of the procedure declaration extends over the entire block in which it appears.

(1) Example:



Subroutine B is local to A, and can be called anywhere in block A. Inside block B, a CALL statement for C results in a call to the first subroutine C, which is local to C. Outside block B, a CALL statement for C will result in a call to the second C.

(2) If a procedure block is a program segment, that is, it is not contained in another block, then the procedure name is public. The procedure, is in fact, an ordinary Fortran subprogram (Chapter 8). Procedure names for inner blocks cannot be made public

even by use of EXTERNAL or PUBLIC declarations. However, inner block procedure names can be used as arguments, and can therefore be called from other blocks.

Example:

A trapping routine for zero division. Division by zero is trapped by Atlas (section 10.3). By use of the library routine AFTERR, a procedure can be called when a trap occurs. A return from the trap procedure may however cause erratic results.

The computation in the example computes the column average of a table. If the column is all zero, the trap routine sets the average to zero.

```
      DIMENSION TABLE (50,10)  AV(50,10)
      INTEGER I,J
      SUBROUTINE TRAP
      INTEGER I
      DO 2 I=1,50
2     AV(I,J)=0.0
      GOTO 4
      END

      CALL AFTERR (1,TRAP)
      DO 4 J=1,10
      SUM=0.0
      DO 2 I=1,50
2     SUM = SUM + TABLE (I,J)
      DO 3 I=1,50
3     AV(I,J) = TABLE (I,J)/SUM
4     CONTINUE
```

If SUM is zero, an error trap occurs in statement 3, calling the TRAP subroutine. This clears the column, and transfers to continue the loop.

(3) The location of a procedure in a block is not crucial. If the procedure has been written using local implicit variables, then it is preferable to put the procedure at the start of a block.

(4) Fortran V procedures must not be used recursively (see Chapter 8).

9.4 COMPOUND LOGICAL IF STATEMENTS

The BEGIN statement (9.2.2) can be used as the successor statement in a Fortran IV type logical IF (6.6.1(3)). In this case the entire BEGIN block is effectively a compound successor statement.

If the logical expression is true, control enters the BEGIN block. Otherwise, none of the statements of the BEGIN block are executed.

Example:

A compound statement which interchanges X and Y if X is greater than Y.

```
IF (X.GT.Y) BEGIN
            REAL Z
            Z = X
            X = Y
            Y = Z
            END
```

The declaration REAL Z ensures that any variables called Z in block(s) containing the

compound IF, will not be overwritten.

Any legal Fortran statements can be written in the BEGIN block, including simple or compound logical IF statements. Transfers out of the block are permitted.

Example: A compound statement which finds the first non-zero element of an array.

```
      I = 0
2     IF (I.LT.N)    BEGIN
      I = I+1
      IF (A(I).EQ.0.0)  GO TO 2
      END
```

The transfer to statement number 2 is a jump out of the block.

The following rules, which are essentially re-statements of block structure properties will minimise the possibility of errors.

(i) If compound logical IF statements are used in a main program, then the main program must start with a BEGIN statement.

(ii) Variables and labels that are explicitly declared in the BEGIN block are local to the block, and have no connection with variables or labels of the same name outside the block.

(iii) Variables that are not declared in the BEGIN block will be the same as variables with the same name outside the block only if they are declared or used before the BEGIN block.

9.5 DYNAMIC ARRAYS

Dynamic arrays can be used to set up a variable amount of storage at run time, and can effect a considerable saving in store requirements.

(1) A dynamic array is an array which is declared in a DIMENSION or type statement with adjustable dimensions (8.8) where the array name is not a dummy argument. The array is local to the block or subprogram in which it is declared.

The amount of storage required for the array is set up when the block or subprogram is entered. All variables used in the dimensions must be declared before the declaration, for the array appears, and their values must be known when the block is entered. This means that the variables must be one of (or combinations of) the following:

(i) a dummy argument
(ii) a COMMON or PUBLIC variable
(iii) a global variable
(iv) a function

If functions are used in the adjustable dimensions, side-effects should be avoided.

Example: A program which reads in the storage requirements for several arrays.

```
BEGIN
READ 101, M, N
BEGIN
DIMENSION A(M,N), B(N,M), C(M*N)
    .
    .
    .
END
END
```

9.5 cont The first BEGIN is needed because this is a main program with inner blocks. The second BEGIN heads the block in which the dynamic arrays are defined. Within this block the arrays can be referred to by ordinary Fortran V statements.

The amount of store used is precisely that needed for the individual run and can be specified accordingly in the Job Description. If the arrays were not dynamic, the maximum amount of store ever used would have to be specified. Thus, in many circumstances, the use of dynamic arrays can reduce the store required for a job.

(2) A dynamic array remains available until control leaves the block. It is available if control enters inner blocks. A dynamic array can be passed as an argument to a procedure block or subprogram, which may use adjustable dimension statements for the array.

(3) When control leaves the block, by means of a RETURN statement, or by reaching the end of a BEGIN block, or by a transfer out of the block, the store for the dynamic array is returned to an area of available store.
This area is known as a stack.

On a re-entry to the block, a different area of store may be assigned to the dynamic array. Therefore, the initial values of the array are unpredictable and values stored in the array are lost when the block is left.

(4) Different routines or blocks can use the same storage area for dynamic arrays. This occurs automatically if the blocks are disjoint (see 9.2.1(3)), and one does not call the other; since the space is returned to the stack when the block is left, and taken by the next block requiring space for dynamic arrays.

Example: A routine for computing the square of a matrix, and storing the result over the original matrix.

```
      SUBROUTINE SQMAT (A,N)
      DIMENSION A(N,N),B(N,N)
      DO 2 I=1,N**2
2     B(I) = A(I)
      CLEAR A
      DO 3 I=1,N
      DO 3 J=1,N
      DO 3 K=1,N
3     A(I,J) = A(I,J)+B(I,K)*B(K,J)
      RETURN
      END
```

A is an argument array with adjustable dimensions, and B is a dynamic array. The space used temporarily by B is available for dynamic arrays of other routines after the return from SQMAT.

(5) The following points apply to the use of dynamic arrays:

(i) Dynamic arrays can be dimensioned in Type statements or DIMENSION statements. All specifications for the array must precede the dimensioning.

(ii) Dynamic arrays should not be specified as COMMON or PUBLIC nor should they appear in EQUIVALENCE statements.

(iii) Specifications for variables used in the dimensions must precede the dimensioning statement.

(iv) The variable dimensions themselves must be in the form described in section 9.5.(1).

9.6 MISCELLANEOUS STATEMENTS AND BLOCK STRUCTURE

9.6.1 COMMON and PUBLIC

COMMON and PUBLIC statements are not normally needed in inner blocks. COMMON and PUBLIC statements are often used to declare variables at the start of a program segment, and can then be referred to as global variables by statements in the inner blocks.

Labelled and unlabelled common areas start from their beginning at the start of every block.

```
Example:  COMMON A,B
          BEGIN
          COMMON C,D
```

The variables A and C refer to the same common store location, as do B and D.

A COMMON statement is a declaration, so that the variables contained in its list are local. However, COMMON statements may result in variables in the outer block being overwritten. Thus, in the example above C is not the same as a variable C of the outer block, but assignment to C will overwrite A.

The usual case of identical COMMON statements will result in variables with the same name using the same store locations.

PUBLIC statements in an inner block have the same effect as PUBLIC statements in a program segment provided that no procedure block is declared with the same name as a PUBLIC variable. These variables are, in fact, public in the sense described in 9.3.4(2).

Because of the way Fortran V handles multi-dimensional arrays, considerable store saving may be made by re-compiling as a program block a program which has a number of routines using the same COMMON and PUBLIC variable declarations. The suggested procedure is:

(i) Remove all *FORTRAN lines.

(ii) Remove the specifications (including dimensioning statements) from all routines.

(iii) Place at the head of the program a *FORTRAN line followed by a BEGIN statement, and one set of the specifications.

(iv) The main routine should be the last routine in the source deck. It should be followed by the *ENTER line. An additional END statement is not necessary.

9.6.2 EQUIVALENCE

EQUIVALENCE statements cannot be used to set up equivalence between local variables of an inner block and global variables. Effectively, an EQUIVALENCE statement is a declaration for all variables appearing in it.

Dynamic arrays must not appear in EQUIVALENCE statements.

9.6.3 EXTERNAL

An EXTERNAL statement informs the Compiler that a name, is, in fact, the name of a procedure. It is not a declaration for the procedure. The EXTERNAL statement is not needed if the procedure block has already appeared, or if a CALL statement has

appeared, or if the function name (with arguments) has appeared in an expression.

9.6.4 IMPLICIT

The effect of an IMPLICIT statement extends over all statements after the appearance of the IMPLICIT statement until the end of the program segment.

9.7 SUMMARY OF BLOCK STRUCTURE

Program block structure is a new concept in Fortran. The detailed definitions given in this chapter may seem strange at first, even to experienced Fortran programmers. The following summary and simplified rules may be helpful.

Program blocks can be subroutines or functions (procedure blocks) or BEGIN blocks. Procedure blocks are called, either by CALL statements or by appearance in an expression. BEGIN blocks are essentially 'compound statements' that is, they are used as if they were single statements; for example in a (Fortran IV) logical IF.

Both kinds of program blocks can be embedded as sub-blocks (inner blocks) to any depth. Variables and labels that are explicitly declared in a program block are local i.e. private, to the block, and have no connection with variables or labels of the same name outside the block. Procedures embedded in a block are also local to the block, and cannot interfere with user or library sub-programs.

Variables that are not declared in a program block will be the same as variables with the same name outside the block only if these are declared or used before the appearance of the block.

Transfers out of a block are legitimate, but transfers into the body of a block are not allowed.

Fortran V combines block structure and ordinary Fortran sub-program segmentation. A program segment is a block which is not contained in another block.

A main program which contains any sub-blocks must start with a BEGIN statement. In other words, a main program is a BEGIN block, but for compatibility with ordinary Fortran, the BEGIN statement need not appear if there is only one END statement.

CHAPTER 10 TRACING AND EXECUTION ERRORS

In Fortran V, some powerful instructions have been introduced which greatly facilitate the checking and correction (debugging) of programs.

These statements enable the values and names of numeric variables to be printed automatically whenever these variables appear in a statement which could change their value. Similarly, in order to follow the path of control through a program, a statement exists which prints the names of labels whenever the statements attached to these labels are executed. The tracing statements are only effective for routines compiled in TEST mode (see section 12.2 (9)).

10.1 THE TRACE STATEMENT

Definition	TRACE $name_1$, $name_2$, ...$name_n$
where $name_1$, $name_2$, ... are nonsubscripted numeric variable or array names.	

(1) The TRACE statement causes the names and new values of the variables in its list to be written whenever these variables appear:

(i) in any READ statement
(ii) on the left hand side of a replacement statement
(iii) as the index of an input or output list, or of a DO statement.

Note that the TRACE statement is not effective when values are changed by means of CLEAR, or ASSIGN, or "machine language", statements and that label variables cannot be traced.

(2) The values and names are written on to output stream number zero (see Appendix 8). The layout of the output is as follows:

columns 1 to 8 : the name of the variable
columns 9 and 10: are blank
column 11 is : The character =
column 12 is : blank
columns 13 to 23 : The value of the variable
column 24 is : blank

a maximum of five of these fields may be written in one line (record). The name of the variable is right adjusted and filled out with blanks if it contains less than eight characters.

If the number of characters in the value is less than 11, then the value is preceded by the appropriate number of blanks (spaces). Negative values are preceded by a minus sign, positive values are not signed. Leading zeros are not printed.

If the variable is of type integer, then the output field is the same as it would be if it were written using the specification I11. (see section 7.6.4.5).

10.1 cont If the variable is of type real, then the output field is the same as it would be if it were written using the specification G11 (see section 7.6.4.4).

Double precision variables are given to single precision only. Only the real parts of complex variables are printed. Text, Boolean and logical variables cannot be traced, and should not be present in the list of the TRACE statement. If a real variable appears in the list of a TRACE statement, and the variable in fact contains an integer (unstandardised) value, then a real value with a possible exponent is printed, followed by a slash.

e.g. X = 1.000 E00/
or
X = 1.000/

(3) Each new value is not necessarily printed as a new record (line). The field is only printed as a new record if:

5 trace fields (120 characters) have already appeared in the current record.

or a combination of trace fields and fields produced by OUTPUT, PRINT, or WRITE statements have caused the current record to exceed 120 characters in length (on the printer).

Note that TRACE output may appear on the same line (or record) as a line produced by a previously executed formatted PRINT or WRITE instruction.

(4) The TRACE statement is effective only in the program, or subprogram in which it appears and only when that program or subprogram is compiled in TEST mode; i.e. when a TEST option appears in its *FORTRAN line. (If this is not the case, then all TRACE statements are ignored - see section 12.2(9)).

(5) Any number of TRACE statements may appear in a program or subprogram, all of the variables in their lists will be traced. The TRACE statements may appear anywhere in the program, but tracing will begin only at the point where the TRACE statement appears; i.e. only those references which textually follow the TRACE statement will be traced.

```
e.g.  2   I = 6
          TRACE I, J
          J = 4
          I = J*7
          GOTO 2
```

will cause the following line to be written

```
J =  4  I =  28  J =  4  I =  28
```

and so on.

(6) If the name of an array appears in the list, then trace output is produced whenever any element (or the name of the whole array) appears in a READ, on the left hand side of a replacement statement, of as the index of an actual or implied DO.

If the name of the array appears as a "short list" in a READ statement, then each element of the array is traced.

When an array is traced the subscripts are not printed, only the array name itself is output in the trace field.

(7) Note that TRACE statements can produce a large amount of output, unless used with discretion.

Example:
```
SUBROUTINE G(I)
REAL X(10)
TRACE J, X, K, L
DIMENSION M(3)
READ 5, (X(K), K=1, 10, 2)
CLEAR M
J=I*3
  .
  .
  .
```

Assuming values for X, and I, this could produce the following output (each field would actually occupy 24 columns).

```
K=  1 X=  6.0000 K=  3 X=  -4.3000        K=  5 X=  1.0000E-01
K=  7 X=  7.3012 K=  9 X=  1.3006Eb02     J=  27
```

10.2 THE TRACE PATH STATEMENT

Definition	TRACE PATH FROM $label_1$ TO $label_2$
where $label_1$ and $label_2$ are statement numbers or named labels, which are both attached to executable statements. If $label_1$ is a named label, then it must be enclosed in parentheses. i.e. TRACE PATH FROM ($label_1$) TO $label_2$	

(1) The TRACE PATH statement is used to trace the path of control through a program or subprogram, and causes the labels of executable statements to be printed whenever those statements are executed.

(2) The TRACE PATH statement is effective only for programs which are compiled in TEST mode, i.e. if a TEST option appears in the *FORTRAN line (section 12.2 (9)). (If this is not the case, then all TRACE PATH statements are ignored).

Any number of TRACE PATH statements may appear in a program or subprogram. Each statement must textually precede the first labelled statement in the routine.

(3) The TRACE PATH statement is effective only for those labels which are executed between executions of $label_1$ and $label_2$ ($label_2$ is not included). i.e. The trace is "switched on" when $label_1$ is executed and "switched off" when $label_2$ is executed. If a subroutine or function (not an intrinsic function) subprogram is called between $label_1$ and $label_2$, then the name of the subprogram is printed on entry, and the characters ← RETURN are printed on return from the subprogram.

If the called subprogram was compiled in TEST mode, then path tracing will continue in that subprogram, even when it contains no TRACE PATH statement itself.

(4) The trace output is written on to output stream zero. The layout of each field is:-

columns 1 to 10:	are blank
columns 11 and 12:	the characters →
columns 13 to 20:	the name of the label (or of the called subprogram)
columns 21 to 24:	are blank

The label is right adjusted within its field of eight columns, and preceded by the appropriate number of blanks. On return from a subprogram, the characters → RETURN are printed. Note that if a label is defined as having leading zeros, then these zeros are not printed (see section 6.1).

Each field is not necessarily started on a new line (or record). The action taken is described in section 10.1 (3).

(5) In addition to the above tracing, when any logical IF statement is encountered between $label_1$ and $label_2$ one of the words "TRUE" or "FALSE" are printed depending on whether the value of the associated logical expression is true or false.

(6) TRACE and TRACE PATH statements may be used together without restriction in any program or subprogram. Note that the TRACE PATH statement can produce a very large amount of output when subprograms in TEST mode are being repeatedly called when a TRACE PATH is effective.

Example:

```
      .
      .
      .
      TRACE PATH FROM 5 TO 6
5     I=4
3     IF(I)6, 6, X
X     I=I-4
      IF (I.EQ.1)  I=I*3
      ASSIGN 6 TO J
      GO TO J
```

will cause the following to be printed

```
→5→3→X    FALSE
```

If the following statement were inserted before statement 5

```
      TRACE I
```

then the output would be:

```
→5   I=   4→3→X   I=   0   FALSE
```

10.3 EXECUTION ERRORS

Several types of program errors may occur when the program is in execution. Typical execution errors are: division by zero, and the reading of illegal characters on a numeric FORMAT specification. If an execution error occurs, Fortran V takes a special action; this action consists of: -

(i) terminating execution of the program
(ii) Printing the execution error and sub-error numbers
(iii) Printing an error tracing giving the storage location where the error occurred. If the error occurred in a subprogram, the positions of the statements which passed control to that subprogram are also given.
(iv) The contents of the accumulator, and of all non-zero index registers (B lines) are printed.
(v) The machine instructions in the area where the error is assumed to have occurred are printed.

In the case of input/output errors, the contents of the I/O buffer are printed, so that the character which caused the error can be identified.

This error output is always printed on output stream zero. An example is given opposite. This output could be produced by:

```
      READ 10, X
10    FORMAT (I6)
```

if used to read in the characters

```
      123.45
```

```
INPUT DATA FAULTY.          = HARTRAN EXECUTION ERROR    5
 SUB-ERROR= 00000030  B119= 04000010  B121= 00000540  ACC EXPONENT= 01400000     INDICATORS= 0000 0040
 ACCUMULATOR HIGH HALF= 0300000000001730  ACCUMULATOR LOW HALF= 0300000000000000
OCCURRED IN THE ROUTINE /IIWD     AT LOCATION 00012470
CALLED FROM THE ROUTINE /IOH      AT LINE 0000001
CALLED FROM THE ROUTINE /RSI      AT LINE 0000031
CALLED FROM THE ROUTINE           AT LINE 0000001


THE I/O BUFFER CONTAINS
123.45


OCTAL CONTENTS OF NON-ZERO B-REGISTERS.
    B1= 00012470   B83= 00000004   B84= 77777620   B85= 00000020   B86= 00000040   B88= 00000005   B89= 0400...1
   B90= 00012470   B91= 00000030   B92= 77777770   B93= 00000030   B94= 00000030   B95= 00025550   B96= 00....1
   B97= 40100677

DUMP OF PROGRAM NEAR PRESUMED CONTROL LOCATION
 OCTAL  FUNCTION BA  BM   OCTAL          OCTAL
LOCATION  CODE            ADDRESS      HALFWORDS
00012400     210 127  89 00011310    1043773100011310
00012410     121   1   0 00012470    0504020000012470
00012420     121 127   0 00024341    0507760000024341
00012430     000   0   8 60006300    0000001060006300
00012440     000   0   0 00013330    0000000000013330
00012450     000   0   0 00013330    0000000000013330
00012460     000   0   0 04000010    0000000004000010
00012470     101  89   0 00006070    0406620000006070
00012500     202 127  89 00012240    1013773100012240
00012510     113   0   0 00012424    0454000000012424
00012520     121  81   0 07000134    0506420007000134
00012530     121  82   0 42453044    0506440042453044



JOB TERMINATED.
```

10.3.1 List of execution errors

Error Number	Sub-Error	CAUSE
1	0	Division overflow
2	0	Exponent overflow
3		Argument out of range, (probably negative). B119 usually equal to address of argument
	0	SQRT
	1	ALOG, LOGF
	2	ARSIN, ARCOS, ASINF, ACOSF
4	→	Incorrect FORMAT specification. Sub-error is equal to the illegal character in Atlas internal code. B119 often equal to address of I/O list item.
5		Input data illegal for FORMAT conversion used
	1	Error character in mantissa
	2	Error character in exponent
	3	Exponent in I conversion
	4	Exponent in F conversion
	5	No exponent in E conversion
	6	Non-octal digit in B or O conversion
6	→	Input ended (e.g. 7/8 card read) Sub-error equal to input stream number involved.
7		One inch (Ampex) tape error
	0	End of file marker read on tape
	1	Variable tape error. (see ref. 7)
	2	Tape Fail - i.e. tape in poor condition
9	-	Supervisor - detected error (see section 10.3.2)
10	-	Binary (unformatted) tape record too short for input list. B119 is equal to the address of the first list item which cannot be filled with information from the logical record. (Does not apply to BCD (formatted) records).
11	-	Computed GO TO out of range, i.e. the transfer is undefined. This error is tested for only if the routine is compiled in TEST mode. See section 12.2 (9).

12	→	Half-inch tape error. Top seven digits of sub-error give selected tape number, and bottom digit = 6 if selected in BCD (formatted), or = 0 if selected in binary (unformatted). B119 = 0 : End of file encountered in writing B119 = 4 : Read parity failure B119 = 10 : End of tape encountered in writing B119 = 14 : Bad tape on writing B119 = 20 : No tape left: failure to detect end of file. B119 = 30 : Attempt to read record of more than 512 words (4096 characters). (N.B. $\frac{1}{2}$ inch tape only) B119 = 40 : Machine error
13	0	I/O buffer not free - "recursive" call of I/O routine. See 10.4.3.
14	0	Excess blocks - job has run out of storage, B119 often contains address of variable requiring extra block.
15	→	Undefined tape unit: incorrect job description. The sub-error is equal to the undefined unit number.

Note: Error number 8 cannot occur in Fortran V jobs.

10.3.2 Supervisor detected errors

There are many different Supervisor-detected errors. In many cases, the error is due to an incorrect, or insufficient Job Description. (see Appendix 8). Common errors are:

Error	Cause, Or action to cure
OUTPUT EXCEEDED	Increase allowance for this stream
C TIME EXCEEDED	Increase COMPUTING time
E TIME EXCEEDED	Increase EXECUTION time
EXCESS BLOCKS (error number 14)	Increase execution STORE
OUTPUT NOT DEFINED	Insert missing stream number
TAPE NOT DEFINED	Insert missing TAPE number
TAPE FAIL	Magnetic tape in poor condition
WRONG TAPE MODE	Job is trying to write to a file-protected (inhibited) magnetic tape
ILLEGAL FUNCTION	Control has been passed out of program area:- often due to overwriting program by exceeding array bounds.
SV OPERAND	Attempt to access illegal (e.g. -ve.) address.

Other errors may occur: if the cause and cure is not obvious, the reader should consult Reference 7.

10.3.3 Interpretation of error output

The execution of a program may have produced:

```
END TAPE = HARTRAN EXECUTION ERROR 7
OCCURRED IN THE ROUTINE /RAT AT LOCATION 01041050
CALLED FROM THE ROUTINE       AT LINE 0000025
```

Since /RAT has been called from a routine with a blank name, that routine must be the main program. Inspection of the main program would show that there is an unformatted READ statement at line 25, which caused the system routine /RAT to be called. It is this statement which has read an end of file marker.

Similarly:
```
COMPLEX RESULT = HARTRAN EXECUTION ERROR 3
OCCURED IN THE ROUTINE X AT LOCATION 00001600
CALLED FROM THE ROUTINE Y AT LINE     0000005
CALLED FROM THE ROUTINE    AT LINE    0000050
```

This would indicate that a complex result has occurred in subprogram X, which was called at line 5 of subprogram Y, whilst subprogram Y was called at line 50 of the main program.

To locate the error more exactly, the loading map must be consulted (see Chapter 12.1(3)). From this map we may find that the absolute entry point of routine X is at 0000171.0 (octal). In addition, the source listing of routine X must be consulted; this may show that the relative entry point of X is located at 0000165.0 (this is the octal number of words from the start of the routine).

This implies, therefore, that the origin (start) of routine X is at location 171 - 165 = 4 (octal).

Hence the error must have been detected at location 160.0 - 4 = 154 (octal) of X. Using the length (given in decimal on the source listing) of X as a guide, the approximate position of the source statement which caused the error may be found. This approximation is usually sufficient to locate the error, but if it is not, it may be necessary to obtain an object listing (machine instructions) by the use of *FORTRAN LIST. (see section 12.2(7)).

Sometimes, the error information given may not be accurate: e.g. if those parts of the program which are examined by the tracing procedure have been overwritten.

If the trace is not correct, then using the loading map, the two routines must be found, whose entry points (absolute) are nearest to, but on either side of, the error location. The position of the error in the source routine can them be found in the manner described above.

It is important to realise that for many errors, control may have passed through as many as four instructions before the error is detected. Since one of these errors may be a jump instruction, it is clear that the dump of program printed out may not in fact contain the instruction which caused the error.

10.4 THE ERROR STATEMENTS

Some of the errors described in section 10.3 may be considered to be relatively minor for some applications, and it may not be desirable that execution be terminated. Sometimes it may not even be necessary to print the error information. In order to deal with this situation, the error statements (they are calls to library subprograms) have been provided. These statements allow the user to take any action desired on the occurrence of an execution error.

Note that the error statements are <u>not</u> effective for execution errors of type 9, 14, or 15.

10.4.1 To continue execution

In order to continue execution after an execution error has occurred, the statement

```
CALL CONTXQ (n)
```

must be positioned, in any program or subprogram, in such a place that it is executed before the execution error occurs. Once the statement has been executed, it remains effective for all programs and subprograms until the job is finished.

n is the error number after whose occurrence it is desired to continue execution. If n is zero, then execution will be continued after the occurrence of all types of errors, except numbers 9, 14 and 15. If execution is to be continued after more than one type of error, then the above technique (CALL CONTXQ (0)) may be used, or more than one CALL CONTXQ statement may be given.

e.g.
```
CALL CONTXQ (6)
CALL CONTXQ (5)
```

Note that once an error 6 or 7 (end of data) has occurred, it is not possible to read more data from that stream. Provided that this is not attempted, execution continues normally. The error output is still printed each time the error(s) occurs. As soon as an error type (n) occurs which is not in an executed CALL CONTXQ(n) statement, then execution of the program is terminated.

10.4.2 To terminate execution

In the same way that the CALL CONTXQ(n) statement is used to continue execution, the CALL ENDXQ(n) statement may be used to "turn off" a previously executed CALL CONTXQ(n) statement. As with CONTXQ, if n is zero, then execution will be terminated after the occurrence of any execution error. ENDXQ is used in the same way as CONTXQ, but it has exactly the reverse effect, i.e. it restores the normal error action for error type n.

10.4.3 To take special action

In addition to continuing execution when an error occurs, it may be desired to avoid printing the error message, and to take some special action: e.g. adding to an error count, or printing one's own message etc. This may be done by insertion of the statements

```
EXTERNAL sname
CALL AFTERR (n, sname)
```

in such a position that the CALL is executed before the error occurs. (i.e. usually at the head of the main program).

sname is name of a SUBROUTINE which must be supplied by the user, and n is the error number which will cause that subroutine to be called.

Once the CALL AFTERR statement has been executed, it remains effective for all programs and subprograms until the job is finished, or until a new CALL AFTERR statement (with the same n) is executed.

The same, or different subroutines may be entered for different error numbers if the appropriate CALL AFTERR statements are given; and the subroutine(s) sname may be explicitly called by the user if desired. The statement CALL AFTERR (0, sname) causes the routine sname to be entered on the occurrence of all types of errors except numbers 9, 14 and 15.

On occurrence of the execution error(n) the subroutine (sname) is entered, no standard error output is printed. With certain restrictions (below) any action may be taken in sname, which follows all the rules applying to normal subroutines. Often, an error count (which may be in COMMON) will be added to. The restrictions on the action taken in sname are as follows:

(i) AFTERR should not be called in sname.
(ii) If sname is called for an input/output type error (numbers 4, 5, 6, 7, 12 and 13) then no input or output instructions (except the Fortran V OUTPUT statement) may appear in sname.

Error messages may still, however, be printed by setting a PUBLIC or COMMON indicator and testing this indicator after return from sname; or by using an OUTPUT instruction.

The CALL AFTERR statement is not effective for execution errors of types 9, 14, or 15. If it is desired to continue execution on return from sname, then a CALL CONTXQ(n) statement should have previously been executed. If this is not done, then execution of a RETURN statement(s) in sname will cause execution of the program to be terminated.

Example:

```
      PUBLIC NERRORS
      DATA NERRORS/0/
      EXTERNAL BADCARD
      CALL AFTERR (5, BADCARD)
            .
            .
      READ 10, (A(I),I=1,10)
   10 FORMAT (10F8.4)
            .
            .
```

```
      SUBROUTINE BADCARD
      PUBLIC NERRORS
      NERRORS=NERROR+1
      RETURN
      END
```

The value of NERRORS can be printed out later, in order to find the number of illegal (mispunched) fields read in.

*10.4.4 Advanced features of AFTERR

The rescue routine (sname) is called with two arguments (but these need not be present as a dummy argument list if their values are not required).

e.g. after

```
      EXTERNAL RESSUB
      CALL AFTERR (1,RESSUB)
```

both of the following are accepted

```
      SUBROUTINE RESSUB
```

and

```
      SUBROUTINE RESSUB (I1,I2)
```

The first argument (I1) contains the execution error number, and the second (I2) is an array containing the useful information printed in the standard monitor dump. The array I2 contains 9 elements, so that the following statement should appear in RESSUB.

```
      DIMENSION I2(9)
```

The elements of I2 contain the following information:

*10.4.4 cont

I2(1) Sub-error number
I2(2) Accumulator contents
I2(3) Accumulator exponent
I2(4) B1 (return link)
I2(5) Presumed control location
I2(6) B119
I2(7) Indicators i.e. V store line 6 (see section 7.8 of ref. 7)
I2(8) Low half of accumulator
I2(9) B121

Note that although many of these quantities are of type integer, some, such as I2(2), and I2(8) may be of type real. Also, the values in the other elements may not be strictly integral, and may contain a non-zero octal fraction digit (this is often the case with I2(6) and I2(9)).

Since the rescue routine can contain a CALL statement, it is possible (by entering another routine with a different setting of a parameter) to continue the program whether or not an appropriate CALL CONTXQ(n) has occurred; but for some errors an ensuing error will immediately terminate the job.

To illustrate the use of this technique we may consider a job which is to copy card images from a half inch tape through a format conversion to binary records on a one inch tape. We will give the half inch tape logical number 4, and the one inch tape logical number 9. The following program will achieve this:-

```
*FORTRAN
      EXTERNAL IBTRAP
      CALL AFTERR (12, IBTRAP)
      CALL MAIN (1)
      END

*FORTRAN
      SUBROUTINE MAIN (KK)
      DIMENSION XX(5)
      GO TO (1, 10) KK
 1    READ (4, 100) XX
100   FORMAT (5E16.7)
      WRITE (9) XX
      GO TO 1
 10   ENDFILE 9
      STOP
      END

*FORTRAN
      SUBROUTINE IBTRAP (I1, I2)
      DIMENSION I2(9)
      IF (I2(6))  10, 11, 10
C     END OF FILE MARKER
11    CALL MAIN (2)
C     OTHER HALF INCH TAPE ERROR
10    CALL OFFIO
      K1=I2(6) + I2(6)
      PRINT 100, K1
100   FORMAT ('bHALFbINCHbTAPEbERROR', I3)
      STOP
      END
```

This example also illustrates the fact that when an error occurs within an active I/O statement, the rescue routines provided by the user cannot use any standard Fortran I/O statement without causing another error (Error 13).

Note, however, that the Fortran V OUTPUT statement may be used. In the above example the statement giving rise to the error 12 is

```
READ (4, 100) XX
```

If this I/O statement is active, the subroutine IBTRAP must not initiate another I/O statement unless library routine OFFIO is first called. This is done at statement number 10.

A call of OFFIO severs the linkage between an active I/O statement, and the library I/O processing routines. It follows therefore, that control cannot be transferred back to the disabled I/O statement; i.e. a RETURN statement must not be used after a CALL OFFIO has been executed.

Errors (in I/O statements) can be dealt with after completion of the I/O statement by using a variable or array, in COMMON or PUBLIC storage.

The example below uses such a technique to detect an error on a particular I/O statement.

```
*FORTRAN
        COMMON K,KK(10),I,XX(10)
        EXTERNAL RESCUE
        CALL CONTXQ(5)
        K=0
        CALL AFTERR (5, RESCUE)
15      FORMAT (10F8.2)
        READ (0, 15)(XX(I), I=1, 10)
        IF (K) 20, 21, 20
20      PRINT 200, (KK(I), I=1, K)
200     FORMAT ('bERRORSbINbELEMENTS', 10I3)
21      CONTINUE
            .
            .
        END

*FORTRAN
        SUBROUTINE RESCUE(J1, J2)  (or just SUBROUTINE RESCUE)
        COMMON K,KK(10),I,XX(10)
        K=K+1
        KK(K)=I
        RETURN
        END
```

In this way the need to record the presence of an input error (error 5) is delayed until the input statement is completed.

*10.4.5 Miscellaneous error routines

The statement

```
CALL RSTERR
```

Nullifies any CALL AFTERR statements which have been executed, and restores the normal error action (for all error numbers).

The statement

```
CALL BDUMP
```

prints the numbers, and contents of all non-zero index registers on whatever output

stream was last selected. (This is n for WRITE (n,f); 0 for PRINT, or 15 for PUNCH depending on the last output statement executed.)

The tracing information, which shows the path of control through the program (as described in section 10.3) may be obtained, on output stream zero, by executing the statement:

```
CALL TRACE
```

(This has no connection with the Fortran V TRACE statements).

The position of all magnetic tapes used by a job may be obtained (on the currently selected output stream) by executing the statement

```
CALL WHTPS
```

The standard error printing can be obtained by the user in his own rescue routine by calling the library routine ERRDG as follows

```
SUBROUTINE RESCUE (K1,K2)
CALL ERRDG (K1,K2)
```

Other useful library routines are described in Appendix 7.

CHAPTER 11 MACHINE LANGUAGE INSTRUCTIONS

*11.1 DESCRIPTION OF MACHINE LANGUAGE

Sometimes, Fortran statements are not suitable, or not sufficient to perform certain operations. This is not usually the case, but may be true of certain special applications. To cope with these applications, Fortran V permits Atlas "machine language" instructions to be written at any point in a program, or subprogram. These instructions all start with a number, and can thus be distinguished from other Fortran statements. This means that no special "entry" directives or column 1 punchings are required. The instructions, as usual, are punched in columns 7 through 72 with a label (if desired) in columns 1 through 5. Continuation cards may be used.

Definition	function, ba, bm, operand

where

function is an Atlas function (operation) code, or an extracode.

and

ba and bm are both unsigned integers, less than 128; they are the index registers (B lines) of the instruction.

and

operand is:

(i) A Fortran variable name, which may be subscripted

(ii) a decimal integer, which may be signed.

(iii) a decimal integer (which may be signed) followed by a decimal point, followed by one octal digit (0 through 7).

(iv) (i) followed by signed versions of (ii) or (iii)

(v) An octal integer, up to 8 digits in length, and preceded by an asterisk. If there are less than 8 digits, they are left-adjusted and filled with zeros. (e.g. *77 = *77000000).

(vi) A label (or statement number), which is enclosed in parentheses.

(vii) Any Fortran V constant (described in Chapter 3) preceded by the character "=".

The properties of the Atlas function codes are described in Reference 7. No checking is done to see whether the machine instruction would cause an error.

The various operands are used as follows:-

(i) Variable name: The address of the variable (modified by any subscript) is used in the instruction.

(ii) Decimal integer: The value of the integer is used in the instruction.

*11.1 cont

(iii) as (ii), but octal fraction (character address) also allowed.

(iv) Name modified by (ii) or (iii): The address of the variable is modified by the constant and used in the instruction

(v) octal constant: The value of the octal constant (half-word) is used in the instruction

(vi) Label: The address of the statement to which the label is attached is used in the instruction.

(vii) = Constant: The constant is stored by the compiler and its address is used in the instruction.

If the operand is a subscripted variable name, then the subscripts may take any form legal in Fortran V. Additional machine instructions will be generated if the subscripts are not integer constants.

Additional machine instructions may also be generated if the program or subprogram is compiled in TEST mode.

A Fortran V program or subprogram may, quite legally, contain only machine language instructions (there must be an END statement, and if a subprogram, a SUBROUTINE or FUNCTION statement). Normally however, the use of PUBLIC or COMMON and RETURN statements can be useful.

The use of machine language instructions may require a knowledge of the instructions compiled for a standard calling sequence in Fortran V. This is the same as for Hartran, and is described below.

An instruction of the form CALL SUBR (a_1, a_2, ...a_n) causes compilation of the following instructions:-

```
121,  1,  0,  AR
121,  127,  0,  AS
 ln          AL
  0          An
  0          An-1
  .           .
  .           .
  0          A2
  0          A1
```

Where A_1, A_2, ...A_n, are the addresses of the arguments a_1, a_2....a_n.

AR is the return address.
AS is the address of the entry point of the called routine.
ln is the line number of the Fortran CALL statement. } { This word is used for
AL is the address of a link store. } { error tracing only, and may be omitted.

i.e. (i) B1 is loaded with the return address.
(ii) Control is transferred to SUBR.
(iii) A link word for execution tracing (this need not be present).
(iv) The addresses of the arguments in reverse order. The address being in the low halfword. The high halfword is zero.

Use of machine language also requires a knowledge of which index registers (B lines) are compiled into the machine language instructions generated by the compiler from standard Fortran V statements.

The B lines which are not used by generated instructions are B81 through B89, and these registers may be freely used in machine language instructions.

Note however, that these registers (81-89) are not saved by Fortran V, and they should therefore be used as 'working' registers only.

Examples:

```
      COMMON P(7,7)                      SUBROUTINE A(J)
        "                                COMMON Z(7,7)
      CALL A (I3)              LABEL     101, 81, 0, J+19.4
        "                                165, 82, 82, *00000077
                                         1066, 82, 0, Z(J+5,6)
                                         121, 127, 0, (LABEL)
                                         101, 127, 0, (66)
                                         334,0,0, = 8B777
                                         342,0,0, = 3.14159
                                   66    RETURN
                                         END
```

Note; this subprogram is not useful, it is merely intended to illustrate the facilities described above.

CHAPTER 12 THE COMPILER DIRECTIVES

The compiler directives are used to inform the compiler about the manner in which a routine is to be compiled and to say whether execution of a program is desired, or merely compilation.

There are also special directives; e.g. to permit a program or a library to be written on to a magnetic tape.

With the exception of one (SAVE PROGRAM), the compiler directives are all written with an asterisk in column 1 (one) of the statement, the rest of the line (columns 2 through 72 on cards) being used for the directive. Continuation cards are not permitted and blanks are not significant (except where stated otherwise).

12.1 THE *RUN DIRECTIVE

Definition	*RUN option list
where option list is a series of options separated by commas. The permitted options are:- NOMAP MAP COMPILE GO GO n (where n is an unsigned integer) IFIX or INT or NINT HARTRAN or F2 or F4 The * is in column 1.	

(1) Every Fortran V job must have a *RUN directive present as the first line following the Job Description. This directive gives information about the job as a whole.

(2) If no option is specified, then the standard options GO, NOMAP, IFIX, and F4 are assumed.

i.e. *RUN
is equivalent to

*RUN GO, NOMAP, IFIX, F4

and

*RUN MAP, F2
is equivalent to

*RUN MAP, GO, IFIX, F2

and so on.

(3) If MAP is specified, then a loading map is printed on output stream zero. This map consists of a list of the names of all routines loaded, together with the absolute storage locations (in octal) of their entry points and their origins. In addition, the names, origins, and lengths, of common and public blocks are listed.

12.1 cont The library routines loaded will also be given, so that for a typical job, the map will comprise about 40 printed lines.

(4) If MAP is not specified, or if NOMAP is specified, then the loading map is not printed.

(5) If GO, or GOn is specified, then (subject to certain conditions) the compiled program will be entered (executed) when compilation is complete. The conditions are:

(i) a *ENTER directive must be present (see section 12.3), and
(ii) If GO is specified, then the program is executed only if it contains no source errors (see Appendix 6).
(iii) If GO n is specified, then the program is executed only if it contains not more than n source errors. (GO is equivalent to GO 0). This enables programs to be executed even though they may contain errors. Note, however, that the machine instructions (if any) generated for incorrect statements, are not well defined, and will often cause execution errors.

If a statement is syntactically incorrect, then it is ignored, and the compiled program is the same as it would be if the statement were not present.

(6) When a program is executed, the words "PROGRAM ENTERED" are printed below the loading map (if present); after this, at the top of a new page, are printed the words "EXECUTION STARTED ON" followed by the time and date.

When the program is not entered the words "EXECUTION DELETED" are printed, and the job is then terminated. The above lines always appear on output stream zero.

(7) If COMPILE is specified, then the compiled program is not executed, even if a *ENTER directive is given.

The use of the COMPILE option is recommended when it is not desired to execute a program, since a considerable saving in storage used by the compiler is effected.

Since routines are not loaded when COMPILE is specified, the loading map will not include library routines.

(8) If IFIX (or INT or NINT) is specified, then the form of truncation assumed for all routines of the job will be IFIX (or INT or NINT). (see sections 5.1.3 and 5.1.4). The TRUNCATION statement (see section 5.1.4) may be used to override, for a particular routine, the form specified in the *RUN directive. If this is done, then at the end of the routine, the form of truncation is set back to that specified on the *RUN directive.

If no truncation specification appears, then IFIX is assumed.

(9) If HARTRAN is specified, then the intrinsic (built-in) function names assumed for all routines in the job will be the same as the names used in Atlas Fortran (Hartran). If F2 is specified then Fortran II names are assumed, and if F4 is specified, then Fortran IV (A.S.A.) names are assumed. The FUNCTIONS statement (see section 8.4.2) may be used to override, for a particular routine, the specification on the *RUN directive. If this is done, then at the end of the routine, the kind of functions used is set back to the kind specified on the *RUN directive. If no functions specification is given, then F4 (i.e. A.S.A.) function names are assumed.

(10) If conflicting options are present in the *RUN directive then the last option (s) appearing are the effective ones.

Examples:
```
*RUN GO 5,MAP
*RUN
*RUN COMPILE, NOMAP
```

12.2 THE *FORTRAN DIRECTIVE

Definition	*FORTRAN optionlist

where optionlist is a series of options separated by commas.
The permitted options are:-

SOURCE
NOSOURCE
CARDS
CARDS n (where n is an unsigned integer)
NOCARDS
LIST
NOLIST
TEST
PRODUCTION
a primed Text constant

The * must appear in column 1.

(1) A *FORTRAN directive must precede every Fortran V program segment presented. A program segment is either a main program, or a subprogram which is not contained within another subprogram. The *FORTRAN options specified for the outer main or subprogram will also apply to any (nested) subprograms which it may contain. A *FORTRAN directive may not be inserted anywhere within a program segment. The *FORTRAN directive is used to inform the compiler about the manner in which a particular program segment (or routine) is to be compiled.

(2) If no option is specified, then the standard options

NOSOURCE
NOCARDS
NOLIST
PRODUCTION

are assumed; and the value of the Text constant will be taken as the first six characters of the name of the outer-most program (or subprogram) of the program segment. If there are less than six characters in this name, then the constant consists of the name left adjusted and filled out with blanks. For a main program, it will be all blanks.

(3) If SOURCE is specified, then a source listing of the routine is printed on output stream zero. This listing is started on a new page, and the layout is shown overleaf.

The line number is the one referred to in execution error diagnostics. (see section 10.3)

The DO LEVEL is a number showing the number of DO statements within which the statement is contained (i.e. the depth of nesting). If the DO LEVEL is more than nine, then an asterisk is printed (this does not imply an error).

The statements are printed exactly as they are read in, including blank lines on cards. On paper tape consecutive newline characters do not produce blank lines; the presence of one (at least) blank character between the newlines will cause the blank line to be printed. Erase characters on paper tape (and all non-printable characters) are printed as a decimal point. The tabulate character on paper tape is printed as one blank (space), this may cause labelled statements to be printed slightly offset (see also Appendix 1).

The "List of Identifers and Properties" (see the example overleaf) is also produced if SOURCE is specified, except for nested subprograms (blocks); (i.e. the list is printed for outer blocks only). This list gives details of:

continued on page 140

```
                                FORTRAN V. LISTING OF SOURCE PROGRAM.
LINE NO.  DO LEVEL

                    *FORTRAN SOURCE
         1                SUBROUTINE PRINT(X)
         2                COMMON Z(5),P,QUARTS,/AA/SQUARE,THETA VALUES(10),TOTAL
         3                INTEGER X(5),NOS(9),SQUARE
         4                TEXT BOX(9)/9*' '/,XX/'X'/
         5                DATA NOS/2,9,4,7,5,3,6,1,8/
         6                PUBLIC Y,NX
                    C     TEST STATEMENTS AND EXAMPLE OF ERROR FOLLOW.
                    X     PRINT 11,SQUARE
                    X 11  FORMAT(14H PRINT ENTERED,I7)
         7                PRINT 10,(Z(K),K=1,5))

SYNTAX ERROR IN LIST

         8             83  IF(X(5).EQ.9) BEGIN

       START BLOCK LEVEL 2

         9                INTEGER ZZ
        10                ZZ=X(5)
        11                X(5)=Z(5)
        12                Z(5)=ZZ
        13                END

       RETURN BLOCK LEVEL 1

        14              43  DO 13 I=1,NX
        15       1          DO 3,J=1,9
        16       2          IF(X(I)=NOS(J)) GO TO 3
        17       2          BOX(J)=XX
        18       2    99999 X(3)=X(3)+1
        19       2          SQUARE=SQUARE/4+MOD(X(3),4)
        20       2          IF(SQUARE-1024) 99999,2,3
        21       2          Y=Y+1000.
        22       2        3 CONTINUE
        23       1          Z(4)=Z(4)+SQUARE
        24       1       13 CONTINUE
        25
        26                2 PRINT 10,BOX
        27               10 FORMAT(1H0,9X,A1,3H I ,A1,3H I ,A1/10X,9H---------/10X,A1,3H I ,A1
                           1,3H I ,A1,/10X,9H---------/10X,A1,3H I ,A1,3H I ,A1//)
        28                 IF(Y-1.E5) ,83,43
        29                 CALL SUMMARISE
        30              EXIT RETURN
        31                 END
```

LIST OF IDENTIFIERS AND PROPERTIES

LOCAL STORE

NAME	TYPE	DECIMAL	NAME	TYPE	DECIMAL	NAME	TYPE	DECIMAL
X	INTEGER ARRAY	DUMMY	BOX	TEXT ARRAY	0	XX	TEXT	9
NOS	INTEGER ARRAY	10	Y	REAL	PUBLIC	NX	INTEGER	PUBLIC
I	INTEGER	21	J	INTEGER	22			

COMMON STORE AA

NAME	TYPE	DECIMAL	NAME	TYPE	DECIMAL	NAME	TYPE	DECIMAL
SQUARE	INTEGER	0	THETAVAL	REAL ARRAY	1	TOTAL	REAL	11

COMMON STORE //

NAME	TYPE	DECIMAL	NAME	TYPE	DECIMAL	NAME	TYPE	DECIMAL
Z	REAL ARRAY	0	P	REAL	5	QUARTS	REAL	6

LABELS

NAME	OCTAL ADDRESS	NAME	OCTAL ADDRESS	NAME	OCTAL ADDRESS
83	0000 423	43	0000 467	99999	0000 524
3	0000 566	13	0000 604	2	0000 411
EXIT	0000 640				

SUBROUTINES AND FUNCTIONS

NAME	TYPE
SUMMARIS	SUBROUTINE

ROUTINE PRINT
ENTRY TO THIS ROUTINE 0000 641.0
LENGTH OF THIS ROUTINE 165

12.2 cont

(i) The layout of local storage, dummy arguments, and PUBLIC for the routine. The layout of COMMON storage for the routine.

(ii) The names, and locations of labels used.

(iii) The name, relative entry point and length of the routine.

(i) The storage layouts give the names and types of all variables used followed by their locations (in decimal) relative to the start of the routine's local data area (or relative to the start of COMMON storage).

(ii) The list of labels comprises a list of the names of all labels (or statement numbers) used in the routine followed by their relative locations, in octal (i.e. relative to the origin, or start, of the routine). FORMAT statement labels are not listed.

(iii) "ENTRY TO THIS ROUTINE" is then printed, the octal number following being the relative location of the entry point of the routine. The entry point is usually near the end of the routine.

The decimal number printed after "LENGTH OF THIS ROUTINE" is the number of machine instructions which have been compiled for the routine: it does not include the storage space used for variables and constants.

(4) If NOSOURCE is specified, or if SOURCE is not specified, then none of the output described under (3) is printed. Note, however, that source statements containing errors, together with the error message, will always be printed.

(5) If CARDS is specified, then object (BAS) cards will be punched for the routine, provided that there are no source errors. If CARDS n is specified, then object cards will be punched if there are not more than n errors. (CARDS is equivalent to CARDS 0). Note that cards will not be punched if TEST is specified.

CARDS n (with n nonzero), should only be used when a source error occurs which does not affect the execution of the routine. If cards are required, then a line OUTPUT 15------ must appear in the Job Description (see Appendix 8).

(6) If NOCARDS is specified, or if CARDS (or CARDS n) is not specified, then no object cards are punched.

(7) If LIST is specified, then a listing of the machine instructions, and program constants, is printed (on output stream zero).

Each instruction is preceded by its absolute octal location. Each instruction is of the form:

function code,	B_a, B_m,	Address Part
(octal)	(decimal)	(octal)

The positions of the various source statements may be found by comparing the addresses in the list of statement labels with the addresses printed in the object listing.

The list of constants is a list of constant values used by the routine, preceded by their absolute octal locations. The constants are printed as sixteen octal digits per word. Text constants may comprise several words. FORMAT specifications (not variable FORMAT) are contained in the constant list, as are various constants used for setting up arrays: so that the list of constants may be considerably greater than the number simple constants used in the routine.

(8) If NOLIST is specified, or if LIST is not specified, then none of the output described under (7) is printed.

LIST should only be specified if the user is expert in the use of Atlas machine language.

(9) If TEST is specified, then the routine is compiled in TEST mode, and any TRACE or TRACE PATH statements it contains will be effective. Note that routines compiled in TEST mode will usually be longer, and less efficient than if they were compiled in PRODUCTION mode.

The use of TEST should, therefore, be restricted to routines which are being debugged, and which contain TRACE statements.

When TEST is specified, a check is done in execution for each subscripted array reference to see whether the value of the subscript (or the product of the subscripts if there are more than one) is larger than the dimension (or product of the dimensions) specified for the array.

If the dimensions are exceeded the message

*** - OUT OF RANGE

is printed on output stream zero, where *** is the name of the array. Note that each individual subscript is not checked, but only the product of the subscripts. Note that if TEST is specified, then object cards are not produced, even if CARDS (or CARDS n) has been specified.

If a routine is compiled in TEST mode, a check is done in execution for each computed GOTO statement, to see whether the value of the arithmetic expression would cause an undefined transfer of control (see section 6.4). This error is execution error 11, and the action taken is described in section 10.3.

When TEST is specified, lines which contain an X in column 1 are compiled, the X itself being ignored (i.e. treated as blank). See section 2.3.

(10) If PRODUCTION is specified, or if TEST is not specified, then any TRACE or TRACE PATH statements contained in the routine are ignored; and no computed GOTO checks are done in execution, so that execution error 11 cannot occur.

In addition, when in PRODUCTION mode, any lines containing an X in column 1 are ignored, i.e. they are treated as comments. This allows for the conditional compilation of statements, which is useful for testing purposes. This facility is described in detail in section 2.3.

(11) If a Text constant is specified, then its first 6 characters are punched in the identification field (columns 73 to 80) of any object cards produced for the routine. Blanks are significant in the constant. See also (2), above.

(12) If conflicting options (e.g. TEST and PRODUCTION) are specified, then the last one to appear will be effective.

Examples:
```
*FORTRAN SOURCE, CARDS, 'SUB B'
*FORTRAN SOURCE, TEST
*FORTRAN
```

12.3 THE *ENTER DIRECTIVE

Definition	*ENTER
This must be punched in columns 1 through 6	

The *ENTER directive is used to execute the program once it has been compiled. The program is entered at the first executable statement of the main program.

The directive must appear as the last line following all source and/or object routines

presented. Any lines following the *ENTER directive are treated as program data on input stream zero, and are not read by the Compiler.

*ENTER is only effective if the *RUN directive satisfies the conditions described in section 12.1 (5).

12.4 THE *END DIRECTIVE

Definition	*END
This must be punched in columns 1 through 4	

This directive causes execution of the program to be deleted, and is normally used with *RUN COMPILE. When a *END directive is encountered, the Compiler does not read any further lines, and the job is terminated.

12.5 THE *INPUT DIRECTIVE

Definition	*INPUT n
Where n is an unsigned integer constant. The * must be in column one.	

When a *INPUT directive is encountered, the Compiler takes its further input from input stream number n.

This directive is especially useful when a job contains source routines on paper tape together with object routines on cards. Such a job would be arranged so that the last line on the paper tape was *INPUT 1 (say), followed by ***Z. The object cards would then be read in on input stream 1. A line

INPUT 1 name of input

would appear in the Job Description, (see Appendix 8), and the object cards would be preceded by the cards

DATA
name of input

*12.6 THE SAVE PROGRAM INSTRUCTION

Definition	SAVE PROGRAM
where the S is punched in, or after column 7 (as for normal Fortran statement)	

The SAVE PROGRAM instruction is used to write (i.e. save) a compiled (object) program on to a magnetic tape.

SAVE PROGRAM is an executable statement. When the statement is encountered, the program and all data areas which have been initialised (or referenced) are copied on to tape. When the program is brought down from tape, as described below, execution begins at the first executable statement after the SAVE PROGRAM. Normally, the SAVE PROGRAM statement will be the first executable statement of the main program.

If this facility is used, then the document title of the Job Description (see Appendix 8) must be in the form:

Job number, name and run name *E* program title

The "E" will cause the program to be executed as well as stored. If execution is not required, the E is omitted, but the asterisks must always be present.

In addition the following line must be present in the job description:

TAPE 99 tape title*WRITE PERMIT

Before this facility is used the magnetic tape must be "initialised" for Compiler LOAD. This is described in A.C.S. publication LSP9.

After the program has been stored as described above, it may be executed at any time by adding the following lines to the usual Job Description

TAPE 99 tape title*WRITE INHIBIT
COMPILER LOAD (replacing COMPILER FORTRAN)
file number/program title

followed by the data (if any) for input stream zero. In the usual way, other input documents may also be present. The "file number" is that output by the writing run. (See A.C.S. publication LSP 9; the LOAD SYSTEM).

If magnetic tapes are used in a job, the logical numbers used must be the same in both the Job Description for the compile run, and the Job Description for the load run. The actual magnetic tape titles need not be the same.

*12.7 MAKING A PRIVATE LIBRARY

Normally, when a Fortran Job is executed, a library of routines is supplied by the compiler. These routines include many subprograms used to deal with input and output, etc. Sometimes, however, it may be convenient for the user to supply his own library, which may contain routines which he often uses, but which are not sufficiently general to be included in the compiler's own library. The following set of directives allow such a private library to be written on to a magnetic tape, and used. If this is done, then the private library will supplant the system library, and therefore it will be necessary to include in the private library, copies of all system routines needed. These routines may be obtained from Atlas Computing Service.

*12.7.1 The *MAKE LIBRARY directive

Definition	*MAKE LIBRARY TAPE n BLOCK m
where n and m are unsigned integers. The * must appear in column one.	

(1) This directive is used to write a private library of routines on to magnetic tape number n, starting at block number m. All routines which follow the directive must be in object (BAS) form. These routines are written on to tape n in order, starting at block m; and writing is terminated by the appearance of a *LBEND directive (see below).

(2) Main routines should not be put into the library; but the library may contain any form of subprogram except BLOCK DATA.

(3) Several libraries may be put on to the same tape, by modifying the value of m.

Example: *MAKE LIBRARY TAPE 1 BLOCK 201

*12.7.2 The *LBEND directive

Definition	*LBEND
This is punched in column 1 through 6	

This directive is used to terminate the writing of a library after a *MAKE LIBRARY directive has been used.

*12.7.3 The *LIBRARY TAPE directive

Definition	*LIBRARY TAPE n BLOCK m
where n and m are unsigned integer constants. The * must appear in column one.	

This directive is used in order to access a private library which has previously been made as described in section 12.7.1. When the directive is encountered, magnetic tape number n is positioned to block number m. The start of a private library should be in block number m: i.e. a library should have been made using *MAKE LIBRARY TAPE i BLOCKj, where i = n and j = m.

The *LIBRARY TAPE directive may appear anywhere in the program (but not within a routine). The directive is only useful if the program is to be executed, and should appear somewhere before the *ENTER directive.

When the library is loaded, (immediately prior to execution) only the routines which have been called (or indirectly called) by the job, are loaded.

Example: *LIBRARY TAPE 1 BLOCK 201

CHAPTER 13 OBJECT (BAS) CARDS (and Arrangement of Decks)

13.1 OBJECT CARDS

Once a program, or subprogram, has been debugged, it is wasteful to keep re-compiling it every time the program is to be executed.

To avoid this, it is possible in Fortran to produce a "machine language" version of each routine compiled. This version is punched on 80 column cards, which are called object (or binary) cards. These cards contain machine instructions etc., which are in a form close to machine language; in fact the language is Binary and Arbitrary Symbolic (BAS).

Object cards are produced for each routine which had CARDS (or CARDSn) specified in its *FORTRAN directive, even if the source routine is punched on paper tape, but see also section 12.2.

The object cards produced for a routine take the form:

(i) a *BAS card followed by

(ii) object cards which have the + (or 10) position of column one punched; followed by

(iii) The last object card, which does not have the + position of column one punched.

13.2 ARRANGEMENT OF ROUTINES

The arrangement of a typical job is as follows:

(i) Job Description
(ii) *RUN directive
(iii) Source routines (if any), each preceded by a *FORTRAN directive.
(iv) Object routines (if any)
(v) *ENTER or *END directive
(vi) Data (if any) for input stream zero
(vii) End of document marker (7/8 card, or ***Z on paper tape).

In addition, separate documents, to be read on input streams other than zero, may be present.

If the source routines are on paper tape, and object routines (always on cards) are to be loaded, then the *INPUT directive must be used (section 12.5).

A job may consist only of source routines, or only of object routines or a mixture of both. If both are present, then all source routines must come first: once a *BAS card has been read, no more source material can be processed.

If a job is to be executed, then no two routines of the same name should be presented. If two routines of the same name are presented, then both are compiled (if source), but

the first one is the one which is used in execution. For all practical purposes, there is no limit on the sizes of routines which may be compiled in Fortran V, since no fixed length tables are used in the compiler.

*13.3 DETAILS OF OBJECT CARDS

Each BAS card is of the form:

Columns	1- 4	*BAS
"	9-16	Name of routine
"	17-24	Entry point relative to origin (start) of routine.
"	25-32	Length of routine.
"	33-40	Number of auxiliary entry points defined in routine.
"	41-48	Length of unlabelled COMMON defined in routine.
"	49-71	The time and date when the object cards were punched is given.
"	72-78	Identification of routine (see section 12.2)
"	79-80	0 0 (start of numbering)

The contents of columns 9-16 are blank in the case of a main program; each of the next 4 fields contain an octal integer, and usually the first and last columns will be zero and left blank.

Following the *BAS card are one or more object cards. These cards all have the 7 and 9 positions of column one punched. They are serially numbered in columns 79 and 80.

If a routine contains more than 100 object cards, then columns 77 and 78 are also used for the three or four digit serial numbers.

Each object card contains a checksum of all internal information. The use of this checksum may be supressed by punching the 0 (zero) position of column one.

The object cards contain information which can be loaded into any part of storage in such a way that the routine may operate there. Such object routines are known as relocatable, and they may be loaded in any order.

The amount of information contained in the card is punched as a count in rows 2 to 6 of column one. This is a binary count with the units position in row 6, and is a count of the number of words of information on the card.

The other columns of object cards are used as follows:

Columns 2, 3 and 4; contain the relocation bits to indicate whether a particular 24 bit half word is to be incremented by the value of the initial loading address of the routine. The punching is described below.

Columns 5 and 6: Contain a 24 bit checksum of all information on the card.

The checksum is such that the logical sum (with end-around-carry) of the information in columns 1 to 8, and of the information words in column 9 onwards is 77777777 octal.

Columns 7 and 8: Contain the loading address of the first information word to be loaded, relative to the origin (start) of the routine.

Columns 9 onwards: Each word of information occupies 4 columns, the first in columns 9 to 12, the second in 13 to 16, and so on up to column 72. From a group of columns, the lowest numbered columns represent the more significant binary digits of a word, with the + row most significant; and the 9 row least significant.

Each word is treated for relocation as two separate 24 bit halfwords. The relocation

bits in column 2 refer to the halfwords in columns 9 to 32; the bits in column 3 refer to the halfwords in columns 33 to 56, and those in column 4 to the halfwords in columns 57 to 72.

The relocation bits of a column are taken in turn from the + row down to the 9 row. A punch indicates that the 24 bit halfword it to be incremented by the initial loading address of the routine. The + row refers to the first pair of columns, the - row to the second, and so on.

APPENDIX 1 PROGRAMS ON PAPER TAPE

Fortran V programs may be punched on 5 or 7 track paper tape as well as cards.

A job consisting of a mixture of paper tape and card routines may be processed by using the *INPUT directive (see section 12.5).

The rules for punching paper tape programs are given below, and apply to both 5 or 7 track unless otherwise specified. For the purpose of counting columns on paper tape, all characters are significant except erases, upper case (figure shift) and lower case (letter shift) characters, which are ignored. Spaces are significant for column counting.

(1) Labels (statement numbers) can be started in column 1, and must not contain more than 5 characters.

Labels may be separated from the statement by a tabulate character, or alternatively the line may be punched by inserting sufficient blanks so that the label field (+ blanks) occupies 5 columns.

The latter method must be used on 5 track tape, since the tabulate character is not available.

If the latter method is used, column 6 (which will normally be blank) forms part of the statement proper, and not part of the label.

A tab appearing anywhere before column 6 causes the following character to be taken as the first character of the statement proper.

(2) Continuation of statements on to new lines cannot be done on paper tape. A punch in column 6 of any paper tape line is taken as the first character of the statement punched in that line.

The maximum number of characters which will be accepted in one paper tape line (i.e. one statement) is 1000.

If elegant source listings are required, then no line should exceed 96 characters in length.

(4) There is no provision for an identification field on paper tape lines (as in columns 73 to 80 on cards) any identification field punched is taken as part of the statement proper.

(5) With the provision given below, mis-punched characters (in statements or in labels) may be deleted by manually back-spacing the tape, and punching an erase over the offending character. The back-space character must not be used.

The statements where erase characters must not appear are

(i) FORMAT statements
(ii) The very first statement (line) of the program following COMPILER FORTRAN (this should be a *RUN directive).

(6) In addition to the above method of deleting mispunched characters, the whole statement may be deleted by punching a ? (query) in the line. This method must be used for errors in the *RUN directive, or in FORMAT statements.

If a ? is punched in a line, then the whole line (from the newline preceding the ? to the newline following the ?) is ignored by the compiler. Any label is also ignored. However, if a source listing is requested such deleted lines are also listed.

Note that if a ? is punched in a primed text constant, the ? is taken as part of the constant, and does not cause the line to be deleted.

(7) Except in FORMAT statements and in text constants, the following characters are always ignored by the compiler:-

space
erase
backspace
spurious upper case (figure shift) i.e. run-out
spurious lower case (letter shift).
inner, or outer set shift characters.

This means that any outer set characters (see Appendix 2) not in primed text constants or in FORMAT statements, will be taken as their inner set equivalents.

Spaces and inner or outer set shift characters are not ignored in primed text constants.

(8) FORMAT statements are subject to more severe rules than other statements, and the following characters must not appear in a FORMAT statement.

Erase
backspace
spurious upper case (figure shift)
spurious lower case (letter shift)

If an error is made when punching a FORMAT statement, then the whole line must be deleted (using a ?) and repunched.

(9) If an error is made when punching a primed text constant, a manual backspace and erase may not be successful. Such errors should be corrected by punching a' (prime-to end the literal) and then a ? (to delete the line), and the whole statement should be re-punched.

(10) In the case of comment (C or π) or directive (*) lines, the C or π or * must appear in column one, i.e. immediately following a newline character. These characters must not be preceded by any erases or redundant upper or lower case characters.

e.g. If a comment line is punched with a G (say) instead of C, then the G cannot be erased: the whole line must be deleted by punching a ?.

(11) The facility for conditional compilation of statements with an X punched in column 1 is also available for paper tape lines. The feature works in the same way as for cards, and is described in detail in section 2.3.

The X character is subject to the conditions described in paragraph (10) above.

APPENDIX 2 THE CHARACTER SET

The full character set of Fortran V is given below. With the exception of 3 characters, only the standard set may be used outside of text constants, of FORMAT statements, or comments. The standard set is given in section 2.2.

The exceptions to the above rule are:

(i) < (less than), which may be used in relational expressions
(ii) > (greater than), which may be used in relational expressions
(iii) ? (query), which may be used to delete mispunched lines on paper tape only (see Appendix 1).

The non standard characters should be used only in text constants for printing etc.

The inner set characters are given in the table at the end of this Appendix. If no paper tape case is specified, then the character is available as both upper and lower case (or figure and letter shift), and in both inner and outer sets.

In the card codes, 10 is the top (+) row of the card, and 11 is the second (-) row.

The card code is the Atlas Extended Hollerith Code. This differs from the EBCDIC code used on System/360, and the ICT 1900 code. The paper tape codes are Atlas codes. Paper tape punches for other computers may have incompatible codes.

The outer set characters are distinguished from the inner set characters by the presence of an outer set shift character in the string. Once an outer set shift has appeared all succeeding characters are taken as outer set, until an inner set shift character appears. Further characters are then taken as inner set, and so on.

If outer set characters are present in a text constant, then, when dealing with the length of the constant, provision must be made for the presence of one extra character for each change of set. The shift characters are not printed, but they are stored as part of the constant.

e.g. PRINT 10, '[A]'

The constant in fact contains 7 characters, and not 3, since [and] are outer set. Thus the format would be

```
10 FORMAT (A7)
```

which would print

```
[A]
```

Note that

```
10 FORMAT (A3)
```

would not print the whole constant, similarly,

A2 3H [A]
cont would not be valid, it must be written as

7H [A]

The outer set characters which are available on the line printers are given below. Non-available characters are printed as a decimal point.

Character	Internal Code	Card Code (rows punched)	7 track tape code (case & binary bits)	5 track tape code (case & binary bits)
£ pounds	03	-	-	-
: colon	17	6, 8	LC0011.111	-
[left bracket	21	11, 7, 8	LC0110.001	-
] right bracket	22	11, 6, 8	LC0110.010	-
- Underline	26	10, 6, 8	LC0100.110	-
| bar	27	10, 7, 8	LC0110.111	-
2 (as in X^2)	30	-	LC0101.010	-
α alpha	32	-	UC0101.010	-
β beta	33	-	UC0111.011	-
$\frac{1}{2}$ half	34	-	UC0101.100	-
10 ten (pence)	35	-	-	-
11 eleven (pence)	36	-	-	-
Erase (prints as .)	77	-	1111.111	11.111

Note: on most Atlas Flexowriter keyboards the characters below may be punched:

§ is acceptable for $\frac{1}{2}$
→ is acceptable for ?
; is acceptable for π
× (not ex) is acceptable for &.

In addition, on some punches, the character '(prime) is given as $ or &.

CHARACTER	Internal Code (octal)	Card Code (rows punched)	7-track tape code (case & binary bits)	5-track tape code (case & binary bits)
0 zero	20	0	UC0100.000	FS00.001
1	21	1	UC0110.001	FS10.000
2	22	2	UC0110.010	FS01.000
3	23	3	UC0100.011	FS11.001
4	24	4	UC0110.100	FS00.100
5	25	5	UC0100.101	FS10.101
6	26	6	UC0100.110	FS01.101
7	27	7	UC0110.111	FS11.100
8	30	8	UC0111.000	FS00.010
9	31	9	UC0101.001	FS10.011
A	41	10,1	UC1010.001	LS10.000
B	42	10,2	UC1010.010	LS01.000
C	43	10,3	UC1000.011	LS11.000
D	44	10,4	UC1010.100	LS00.100
E	45	10,5	UC1000.101	LS10.100
F	46	10,6	UC1000.110	LS01.100
G	47	10,7	UC1010.111	LS11.100
H	50	10,8	UC1011.000	LS00.010
I	51	10,9	UC1001.001	LS10.010
J	52	11,1	UC1001.010	LS01.010
K	53	11,2	UC1011.011	LS11.010
L	54	11,3	UC1001.100	LS00.110
M	55	11,4	UC1011.101	LS10.110
N	56	11,5	UC1011.110	LS01.110
O (owe)	57	11,6	UC1001.111	LS11.110
P	60	11,7	UC1110.000	LS00.001
Q	61	11,8	UC1100.001	LS10.001
R	62	11,9	UC1100.010	LS01.001
S	63	0,2	UC1110.011	LS11.001
T	64	0,3	UC1100.100	LS00.101
U	65	0,4	UC1110.101	LS10.101
V	66	0,5	UC1110.110	LS01.101
W	67	0,6	UC1100.111	LS11.101
X	70	0,7	UC1101.000	LS00.011
Y	71	0,8	UC1111.001	LS10.011
Z	72	0,9	UC1111.010	LS01.011
blank	01	none	0010.000	FS01.110
(	10	0,4,8	LC0111.000	FS10.100
)	11	10,4,8	LC0101.001	FS01.100
, comma	12	0,8,3	LC0101.111	FS11.110
π pi	13	11,8,3	LC0111.011	LS01.111
? query	14	11,8,5	LC0101.100	LS10.111
& ampersand	15	8,5	LC0111.101	
* asterisk	16	11,8,4	LC0111.110	FS11.000
/ slash	17	0,1	UC0011.111	FS11.101
< less than	32	0,8,5	LC0100.011	
> greater	33	10,8,5	LC0110.100	FS10.001
= equals	34	8,3	LC0100.101	FS01.010
+ plus	35	10	UC0111.101	FS01.011
- minus	36	11	UC0111.110	FS11.010
. point	37	10,8,3	UC0101.111	00.111
' prime	40	8,4	LC0100.000	FS10.111
tabulate	02		0000.100	
shift to O. Set	04			
shift to I. Set	05			
shift to LC/LS	06		0010.110	11.011
shift to UC/FS	07		0000.111	00.000

APPENDIX 3 SOURCE STATEMENTS AND SEQUENCING

A complete list of Fortran V statements is given overleaf. In general, the ordering of the statements should be such that all specifications statements appear in the text of the program before any of the specified variables are referenced.

Variables are considered to have been referenced if they have occurred in:

(i) an executable statement, or a statement function.
(ii) a DATA statement, or Type statement in which values are assigned.
(iii) as a dummy argument
(iv) a COMMON or PUBLIC statement.
(v) an adjustable dimensioning statement.
(vi) an EXTERNAL statement, or otherwise used as a routine name.
(vii) or have already occurred in an EQUIVALENCE.

The specification statements are:-

Type statements
COMMON
DIMENSION
PUBLIC
EXTERNAL

If a variable appears in a specification statement and is to appear in an EQUIVALENCE statement, then the EQUIVALENCE statement must come after the specification(s), and before any reference to the variable. Otherwise the order of the specification statements is not significant, except that:

(i) For adjustable dimensions, the array dimension declaration must come after (or in the same statement as) any type declaration. It must also come after any declarations involving the variables used in the dimensions.

(ii) If values are assigned in a Type statement, then any dimension information must come first (or it may be in the same statement). Such variables must not appear in an EQUIVALENCE statement.

A3
cont

Statement	Executable or Not	Position in Program
a = b (arithmetic or logical replacement)	Executable	Anywhere
ASSIGN	Executable	Anywhere
BACKSPACE	Executable	Anywhere
BEGIN	Non executable	at start of a program block
BLOCK DATA	Non executable	First Statement of BLOCK DATA Subprogram
BOOLEAN	Non executable	See above
CALL	Executable	Anywhere
CLEAR	Executable	Anywhere
COMMON	Non executable	See above
COMPLEX	Non executable	See above
CONTINUE	Executable	Anywhere
DATA	Non executable	Anywhere
DIMENSION	Non executable	See above
DO	Executable	Before the statement number it references
DOUBLE LENGTH DOUBLE PRECISION REAL*8	Non executable	See above
END	Non executable	Last statement of program or subprogram, or of a program block
ENDFILE	Executable	Anywhere
EQUIVALENCE	Non executable	See above
EXTERNAL	Non executable	See above
f(x) = b (statement function)	Executed by reference to it in other statements	Before any reference to the function in an executable statement
FORMAT	Non executable	Anywhere
FUNCTION type FUNCTION	Non executable	First statement of FUNCTION subprogram
XXX FUNCTIONS	Non executable	Anywhere (see Chapter 8)
GOTO	Executable	Anywhere
GOTO (computed)	Executable	Anywhere

Statement	Executable or Not	Position in Program
IF (arithmetic)	Executable	Anywhere
IF (Hartran logical and Fortran IV logical)	Executable	Anywhere
IMPLICIT	Non executable	See above
INTEGER	Non executable	See above
LOGICAL	Non executable	See above
Machine code	Executable	Anywhere
OUTPUT	Executable	Anywhere
PAUSE	Executable	Anywhere
PRINT	Executable	Anywhere
PUNCH	Executable	Anywhere
READ READ TAPE READ INPUT TAPE	Executable	Anywhere
REAL	Non executable	See above
RETURN	Executable	Anywhere
REWIND	Executable	Anywhere
STOP	Executable	Anywhere
SUBROUTINE	Non executable	First statement of SUBROUTINE subprogram
TEXT	Non executable	See above
TRACE	Executable	Anywhere (see Chapter 10)
TRACE PATH	Executable	before first label referred to in the TRACE PATH statement
TRUNCATION	Non executable	Anywhere (see Chapter 5)
UNLOAD	Executable	Anywhere
WRITE WRITE TAPE WRITE OUTPUT TAPE	Executable	Anywhere

APPENDIX 4 TABLE OF SYSTEM FUNCTIONS

The tables overleaf give the names and properties of all intrinsic and basic external functions available in Fortran V.

The instrinsic names vary according to whether a FUNCTIONS statement (or *RUN option) has been supplied. The basic external names (if available) are always the same, no matter what set of intrinsic names is in force. Functions are described in detail in section 8.4.

In the tables the following abbreviations are used:

C	Complex
D	Double precision
I	Integer
L	Logical
R	Real
a	argument
a_1	first argument
a_2	second argument

The arguments of the trigonometric functions are expressed in radians.

INTRINSIC NAME							
ASA or F4 FUNCTIONS (STANDARD)	OLD or F2 FUNCTIONS	HARTRAN FUNCTIONS	Basic External Name	Type of Function (result)	Number of Arguments	Type of Arguments	Properties of Function
ABS IABS CABS DABS	ABSF XABSF - -	ABSF IABSF - -	ABS IABS - -	R I R D	1 1 1 1	R I C D	Result is absolute value of a. i.e. ABS(a)=/a/
*AINT * INT * IDINT (see note below)	INTF XINTF -	AINTF INTF -	AINT INT -	R I I	1 1 1	R R D	Result is equal to the largest integer less than or equal to a. (see 5.1.4(2)).
NINT	-	NINTF	NINT	I	1	R	Result is equal to the nearest integer to a. See 5.1.4. (2)
- IFIX	- XFIXF	FIXF IFIXF	FIX IFIX	R I	1 1	R R	Result is equal to the sign of a multiplied by the largest integer which is less than, or equal to, the modulus of a. See 5.1.4 (2).
SIGN ISIGN DSIGN	SIGNF XSIGNF -	SIGNF ISIGNF -	SIGN ISIGN -	R I D	2 2 2	R I D	Result is equal to sign of a_2 multiplied by the modulus of a_1. (transfer of sign).
DIM IDIM	DIMF XDIMF	DIMF IDIMF	DIM IDIM	R I	2 2	R I	Result is equal to a_1 minus the smaller value of a_1 and a_2. (a_1 -Min(a_1, a_2)). i.e.Positive difference
AMOD MOD DMOD	MODF XMODF -	AMODF MODF	AMOD MOD -	R I D	2 2 2	R I D	Remaindering. Result is equal to $a_1-(a_1/a_2)a_2$ Where (a_1/a_2) is the integer whose magnitude does not exceed the value of a_1/a_2, and whose sign is the same as a_1/a_2. e.g. a_1-a_2* IFIX (a_1/a_2), for MOD.

INTRINSIC NAME							
ASA or F4 FUNCTIONS (STANDARD)	OLD or F2 FUNCTIONS	HARTRAN FUNCTIONS	Basic External Name	Type of Function (result)	Number of Arguments	Type of Arguments	Properties of Function
AMAX0 (zero) MAX0 (zero) AMAX1 (one) MAX1 (one) DMAX1 (one)	MAX0F XMAX0F MAX1F XMAX1F -	- MAXF AMAXF - -	AMAX0 MAX0 AMAX1 MAX1 -	R I R I D	≥ 2 ≥ 2 ≥ 2 ≥ 2 ≥ 2	I I R R D	Choosing largest value. Result is equal to the largest argument; or IFIX (largest argument) for MAX1.
AMIN0 (zero) MIN0 (zero) AMIN1 (one) MIN1 (one) DMIN1 (one)	MIN0F XMIN0F MIN1F XMIN1F -	- MINF AMINF - -	AMIN0 MIN0 AMIN1 MIN1 -	R I R I D	≥ 2 ≥ 2 ≥ 2 ≥ 2 ≥ 2	I I R R D	Choosing smallest value. Result is equal to the smallest argument: or IFIX (smallest argument) for MIN1.
FLOAT	FLOATF	FLOATF	FLOAT	R	1	I	Converts a from integer to real.
DBLE	-	-	-	D	1	R	Express single precision (real) argument in double precision form.
SNGL	-	-	-	R	1	D	Express double precision argument in single precision (real) form. i.e. obtain most significant part.
REAL	-	-	-	R	1	C	Obtain real part of complex argument.
CMPLX	-	-	-	C	2	R	Express two real arguments in complex form. Result = $a_1 + ia_2$
AIMAG	-	-	-	R	1	C	Obtain imaginary part of complex argument.
CONJG	-	-	-	C	1	C	Obtain conjugate of complex argument. If $a = x+iy$ then result = $x-iy$, and vice versa.

INTRINSIC NAME							
ASA or F4 FUNCTIONS (STANDARD)	OLD or F2 FUNCTIONS	HARTRAN FUNCTIONS	Basic External Name	Type of Function (result)	Number of Arguments	Type of Arguments	Properties of Function
EXP	EXPF	EXPF	EXP	R	1	R	Exponential. Result = e^a.
CEXP	-	-	-	C	1	C	
DEXP	-	-	-	D	1	D	
ALOG	LOGF	LOGF	ALOG	R	1	R	Natural logarithm. Result = $\log_e a$.
CLOG	-	-	-	C	1	C	
DLOG	-	-	-	D	1	D	
ALOG10	LOG10F	-	-	R	1	R	Common logarithm. Result = $\log_{10} a$.
DLOG10	-	-	-	D	1	D	
SIN	SINF	SINF	SIN	R	1	R	Result is equal to sine of argument.
CSIN	-	-	-	C	1	C	sin (a)
DSIN	-	-	-	D	1	D	
COS	COSF	COSF	COS	R	1	R	Result is equal to cosine of argument
CCOS	-	-	-	C	1	C	cos (a)
DCOS	-	-	-	D	1	D	
TAN	-	TANF	TAN	R	1	R	Result is equal to tangent of argument. tan (a)
COTAN	-	-	-	R	1	R	Result is equal to cotangent of argument. cot (a)
ARSIN	-	ASINF	ASIN	R	1	R	Obtain arcsine of argument $\sin^{-1}$ (a)
ARCOS	-	ACOSF	ACOS	R	1	R	Obtain arccosine of argument $\cos^{-1}$ (a)

INTRINSIC NAME							
ASA or F4 FUNCTIONS (STANDARD)	OLD or F2 FUNCTIONS	HARTRAN FUNCTIONS	Basic External Name	Type of Function (result)	Number of Arguments	Type of Arguments	Properties of Function
ATAN DATAN	ATANF -	ATANF -	ATAN -	R D	1 1	R D	Obtain arctangent of argument. $\tan^{-1}$ (a).
ATAN2 DATAN2	ATAN2F -	- -	- -	R D	2 2	R D	Obtain arctangent of quotient. $\tan^{-1}$ (a_1/a_2). The result lies between + π and - π
TANH	-	TANHF	TANH	R	1	R	Hyperbolic tangent. tanh (a).
SINH	-	-	-	R	1	R	Hyperbolic sine. sinh (a)
COSH	-	-	-	R	1	R	Hyperbolic cosine. cosh (a)
SQRT CSQRT DSQRT	SQRTF - -	SQRTF - -	SQRT - -	R C D	1 1 1	R C D	Obtain square root of argument.
AND OR ER NOT SHIFTR SHIFTL	- - - - - -	- - - - - -	- - - - - -		2 2 2 1 2 2		See Section 5.1.6

* In the A.S.A. specification, both INT and IFIX are defined as giving the sign of a times the largest integer less than or equal to the modulus of a.

APPENDIX 5 NOTES ON EFFICIENCY

The information in this section may serve as a guide to writing programs which are economical in the use of time and store on Atlas. However, efficient programming is an art which is acquired only by experience. Furthermore, the efficiency of a program depends considerably on the machine and compiler on which it is run, and may change if improvements are made in the compiler.

Compilation Efficiency

The Fortran V compiler has been carefully designed for high speed compilation. The compilation speed tends to depend on the number of routines and the overall length of the program, with only minor variations caused by the type of statement used.

Compilation expenses are considerably increased by requesting

Object Listings
BAS Cards
Source Listings

A program punched on paper tape compiles faster than the same program on cards. Compilation of many short routines will be more expensive than an equivalent compilation with fewer but longer routines.

The 'short list' form in DATA, or input-output statements, will compile more efficiently than the 'full list' form.

Execution Efficiency

The major cause of inefficiency in a program usually turns out to be use of an inefficient technique, rather than the efficiency of the machine code produced by the compiler. Some gains in efficiency may be made if the following points are kept in mind.

Statements inside a loop are executed many times. Therefore, the major rule for efficient coding is: Do not put a statement inside the loop if it can be executed outside the loop.

Calls to routines take time. Arguments of routines must be initialised. More efficient coding will result if, instead of arguments, COMMON or PUBLIC or global variables are used.

Multi-dimensional arrays are less efficient than one-dimensional arrays.

Adjustable dimensions or dynamic arrays are somewhat slower in execution than arrays with constant dimensions.

For very large multi-dimensional arrays, efficiency of the Atlas system will be increased if the first subscripts vary most rapidly.

e.g.
```
DO 2 K=1,N
DO 2 J =1,N
```

```
      DO 2 I=1,N
2     A(I,J,K) = 1.0
```

Storage will be saved if, when multi-dimensional arrays are declared, the largest dimension is given first.

e.g. DIMENSION A(250, 10)

is better than

DIMENSION A(10, 250)

Programs compiled in TEST mode are less efficient than programs in PRODUCTION mode. TEST mode should be used only when debugging.

The use of a large number of short labelled COMMON blocks may waste storage.

The INT type of truncation is more efficient in execution than IFIX or NINT.

DATA initialisation and CLEAR are efficient in execution.

APPENDIX 6 SOURCE PROGRAM ERRORS

A list of source program errors detected by the Fortran V compiler is given below. When an error is found, the appropriate message is explicitly printed. If no source listing is specified, then the (possibly) incorrect statement is printed together with its message. The messages always apply to the Fortran statement preceding them, but, if a source listing is obtained, then comment lines and/or FORMAT statements may appear between the incorrect statement and its error message.

In Fortran V there is a distinction between warnings or 'possible errors' and serious or 'grammatical' errors. Each serious error causes the error count (as in *RUN GO n, or *FORTRAN CARDS n) to be increased by one. The warning-only messages do not add to the error count (n).

Note that, once an error has occurred, the compiler may mistake the meaning of later statements, and may accept incorrect statements or reject correct ones. (see 66 for an example).

A list of current errors is given below, further messages may be added from time to time.

(1) SYNTACTICAL ERROR IN ABOVE STATEMENT.

The preceding statement is not a recognisable Fortran V statement, and has probably been mis-punched. e.g. a comma instead of a point, etc.

Such error statements are ignored, but any attached label is accepted. i.e. these statements are treated as CONTINUEs.

(2) LABEL NOT SET -***

Occurs when a label (statement number) is referred to, but is not present in columns 1-5 in the routine.

e.g. LABEL NOT SET - 10

If a FORMAT label is not set, then the message is of the form

LABEL NOT SET - NO.10

(3) LABEL GIVEN TWICE -***

Occurs if two, or more, statements have the same label.

(4) FORMAT HAS NO LABEL

Every FORMAT statement must possess a statement number.

(5) SPECIAL FORMAT 6G20 ACCESSED

A6 cont (Warning only). Occurs if reference to FORMAT omitted from formatted I/O statement (6G20 is accessed).

e.g. PRINT, I

(6) FORMAT GIVEN TWICE: FIRST ACCEPTED

Two FORMAT statements have the same statement number.

(7) STRANGE I/O STATEMENT

Occurs for WRITE INPUT TAPE, or similar invalid statements. The statement is ignored.

(8) EXPRESSION IN INPUT LIST

Occurs for example in

READ 10, X+Y

(9) SYNTAX ERROR IN LIST

Can occur in I/O list or DATA statement. Usually due to excess or missing commas or brackets.

(10) ILLEGAL UNIT NUMBER

Unit number referred to is not a legal expression.

e.g. REWIND//

(11) LONGER /VALUES/ THAN LIST

In DATA or Type statements. There are more constants than list items. The excess constants are ignored.

(12) LONGER LIST THAN /VALUES/

As (11) but too few constants - the excess list items are undefined.

(13) DIFFERENT TYPES FOR /VALUE/ AND LIST ITEM

In DATA or Type statements. Constant and list item should be of same type. Constant is, however, loaded without conversion.

(14) STRANGE CONSTANT IN /VALUES/

In DATA or Type statements

e.g. DATA A,B /2,X/

(15) ARRAY TOO SMALL FOR LIST PARAMETERS

In DATA statements

e.g. DIMENSION A(10)
DATA (A(I), I=1,20)/20*2./

The excess values are not loaded.

(16) /VALUES/ CANNOT BE ASSIGNED TO BLANK COMMON, OR ADJUSTABLE ARRAYS.

Can occur in DATA or Type statements. The message is self explanatory.

(17) /VALUE/ TOO LONG FOR FIELD

(Warning only). Can occur in Type or DATA statements.

e.g. DATA A/10HABCDEFGHIJ/
Where A is not an array.

(18) DATA PUNCH LIST FULL

There are too many DATA values for the compiler. This error is not likely to occur.

(19) BLOCK SIZE LARGER THAN IN EARLIER ROUTINE

An attempt has been made to increase the length of a COMMON block defined in an earlier routine.

(20) STORE AREAS OVERLAP

Can occur if program is too large for machine, or if very large arrays are used.

(21) OPERATOR IGNORED

Arithmetic operator is redundant.

e.g. A=*1, taken as A=1

(22) MISSING OPERATOR IS TAKEN AS *

e.g. A(A+1) is taken as A*(A+1)

Note that AB is not taken as A*B

(23) TOO MANY SUBSCRIPTS

More subscripts present than dimensions declared. The excess subscripts are ignored.

(24) TOO FEW SUBSCRIPTS, OR ARRAY NAME USED AS SCALAR

(Warning only). The missing subscripts are assumed to be one.

(25) INVALID MIXTURE OF TYPES IN EXPRESSION

e.g. A = 3*'XY'

Instructions are compiled but result usually meaningless.

(26) ILLEGAL OPERATION ON NON-NUMERIC TERMS

e.g. X = 'XY'*'XYZ'

Instructions are compiled, but result usually meaningless.

(27) TYPES OF LHS AND RHS ARE NOT COMPATIBLE

e.g. INTEGER A
A='XY'

A6 See Chapter 5 for meaningful combinations.
cont

(28) DO, OR IMPLIED DO: INDEX CHANGED WITHIN LOOP

```
e.g.    PRINT 10 ((A(I), I=1, 5), B(I), I=1, 6)
or
        DO 2I = 1,5
        I=4 etc.
```

Indices must be different

(29) DO ENDS ON CONTROL STATEMENT

```
e.g.    DO 2 I = 1,5
    2   DO 3 J = 1,6
          . .
          . .
          . .
```

(Use CONTINUE)

(30) DO: LABEL HAS ALREADY APPEARED

```
e.g.  2 X=4
        DO 2 I = 1,2
```

(31) ILLEGAL DO NESTING

```
e.g.    DO 2 I = 1,10
        DO 3 J = 1,5
    2   CONTINUE
```

(32) DO LOOP NOT TERMINATED

The label referred to is not set in program.

(33) ARRAY DECLARED TWICE: LAST IGNORED.

Dimension information is repeated. Only first dimensions specified are accepted.

(34) REFERENCED ITEM DECLARED COMMON: COMMON IGNORED.

```
e.g.    A=X
        COMMON X
    or
        REAL A/1./
        COMMON A
    or
        EQUIVALENCE (A,B)
        COMMON B
```

see Appendix 3 for rules.

(35) TWO ITEMS ALREADY REFERENCED: GROUP IGNORED.

Can occur in EQUIVALENCE

```
e.g.    COMMON A
        B=X
        EQUIVALENCE (A,B)
```

or

```
COMMON A (20)
REAL B(10)
EQUIVALENCE (A,B) (A(10),B(10))
```

See Chapter 4 for rules.

(36) ITEM DECLARED AFTER REFERENCE

e.g.
```
X=4
INTEGER X
```

(37) ITEM FORCED BEFORE START OF DATA AREA

Can Occur in EQUIVALENCE

e.g.
```
DIMENSION X (10)
COMMON Y
EQUIVALENCE (X(9), Y)
```

Since Y is referenced, X(1) to X(8) now lie before the start of the COMMON region. This is invalid.

(38) UNDECLARED ARRAY ON LHS OF STATEMENT

e.g.
```
X=4
 .
 .
X(I) = 7
```
Where X has not been dimensioned.

(39) STATEMENT FUNCTION DEFINED

(Warning only). Appears when ever a statement function is defined. If no function was intended, then an array has not been dimensioned.

(40) INCORRECT ARGUMENT IN STATEMENT FUNCTION

Dummy arguments must be simple variable or array names.

(41) WRONG NUMBER OF ARGUMENTS IN SYSTEM FUNCTION

e.g.
```
X=SQRT (Y, Z)
```

Number of arguments must be correct - See Appendix 4.

(42) SYSTEM FUNCTION OVERWRITTEN

(Warning only). Occurs if statement function is given same name as a system function. The statement function takes precedence.

(43) FUNCTION HAS NO ARGUMENTS

(Warning only). Occurs in FUNCTION statement when no dummy argument is present.

(44) FUNCTION NAME NOT USED IN ROUTINE

(Warning only). Occurs in FUNCTION subprogram, when the function name has not been assigned a value.

(45) DUMMY ARGUMENT REPEATED

A6 cont Can occur in SUBROUTINE or FUNCTION statements when a dummy argument is referred to more than once.

e.g. SUBROUTINE X(A,B,A)

(46) VARIABLE NAME WAS PREVIOUSLY USED AS SUBPROGRAM NAME

e.g.
```
CALL X
  .
  .
  .
X = 4
```

Names should be distinct.

(47) SUBPROGRAM NAME WAS PREVIOUSLY USED AS VARIABLE

Similar to (46).

(48) EQUIVALENCE ON ADJUSTABLE ARRAY

e.g.
```
DIMENSION A(N)
EQUIVALENCE (A,B)
```

Illegal equivalence - see Chapter 4. Also applies to dynamic arrays.

(49) EXTERNAL: NAME PREVIOUSLY USED

Name previously used as a variable is now declared EXTERNAL.

(50) DYNAMIC ARRAY DECLARED

(Warning only). Occurs when any dynamic array is dimensioned. This is not an error, but may be due to name left out of argument list.

(51) ADJUSTABLE ARRAY IN PUBLIC OR COMMON

Not allowed. Also applies to dynamic arrays.

(52) LABEL ON CONTINUATION CARD: LABEL IGNORED

(Warning only). May not be an error, but is often due to punching statements from cc 1, rather than cc 7.

(53) TOO MANY CONTINUATION CARDS: STATEMENT IGNORED.

The statement covers more than 35 cards, and is ignored.

(54) ILLEGAL CHARACTER: STATEMENT IGNORED

A character has been punched (on a card), which is not in the Atlas card set. See Appendix 2.

(55) OUTER SET CHARACTER IN LITERAL

(Warning only). This is not an error, but see Appendix 2 for use of O/S characters.

(56) TEXT CONSTANT EXTENDS BEYOND END OF STATEMENT

(Warning only). Occurs if second prime missing or if n H too long.

e.g. X = 80 H

with no continuation card.

(57) CONSTANT OUT OF RANGE

Printed if a real or D. P. constant in the program is larger than 10**110.

(58) CONSTANT HAS MORE THAN 22 DIGITS

(Warning only). Constant cannot be stored to accuracy given, but excess digits are treated as zero, and are not ignored, so that the number is stored accurate to 22 digits.

(59) ILLEGAL OPERAND IN M/C CODE

Message is self explanatory. See Chapter 11 for legal forms.

(60) STATEMENT CANNOT BE REACHED

(Warning only). An unlabelled statement follows an unconditional transfer, and thus can never be accessed.

(61) UNRECOGNISED OPTION ON DIRECTIVE

e.g. *FORTRAN GO

The illegal option is ignored.

(62) ROUTINE ALREADY LOADED

(Warning only). A routine with the same name has appeared before. This is not an error. Only the first routine is used.

(63) CHECKSUM ERROR

Occurs if a mis-punched, object (binary) card is loaded.

(64) P.U.T OR CARD MIXUP

Occurs if binary cards are missing or in wrong order, or if a binary deck which uses the Hartran PARAMETER facility is loaded.

(65) BINARY MIXUP

Occurs when the loader expects a binary card, but actually reads BCD card. Due to cards in wrong order or missing.

(66) REQUIRED ROUTINES,

followed by a list of routine names which have been referred to, but which are not present. Note that this could occur by forgetting to dimension an array.

e.g. if A is meant to be an array, but has not been dimensioned, and if the following statement appears:

X=A(I),

then A appears to be a function, and is treated by the compiler as such - thus the above message could appear.

(67) NO MAIN

No main program is present. The job is not executed.

APPENDIX 7 LIBRARY SUBPROGRAMS

The following standard constants are held in the library, and may be accessed by putting their names in an EXTERNAL statement. If the names are also declared to be DOUBLE PRECISION, then double precision (22 digits) values will be accessed.

e.g.
```
EXTERNAL PI
AREA = PI*R**2
```

Name	Single length Value
PI	3.1415926536
RECIPI ($1/\pi$)	0.31830988618
DEG ($180/\pi$)	57.295779513
RAD ($\pi/180$)	0.017453292520
E (e)	2.7182818284
LOGEPI ($\log_e \pi$)	1.1447298858
LOGE10 ($\log_e 10$)	2.3025850930
LOGE2 ($\log_e 2$)	0.69314718056
LOG2E ($\log_2 e$)	1.4426950409
LOG10E ($\log_{10} e$)	0.43429448190
GAMMA	0.57721566490
GAUSS	0.47693627620

In addition to the list below many mathematical (e.g. matrix) subroutines are available; these are not in the system library, but are available as object (BAS) decks from Atlas Computing Service.

Other system library routines are described in reference 6. These include many routines for specialised input/output operations.

The following library subroutines can be accessed by CALL statements from any Fortran V program, or subprogram.

Name, and form of arguments	Properties
EXIT	Prints 'END OF JOB' on output stream zero, and terminates execution (normal exit).
EEXIT	Prints 'JOB TERMINATED' on output stream zero, and terminates execution (error exit).
OUTBRK (N)	Causes a break of output to appear on output stream N. This should be used to divide large amounts of output into manageable segments (of (say) 3000 lines).

(continued)

A7
cont

Name, and form of arguments	Properties
OUTSEL (N)	Select output stream N.
INPSEL (N)	Select input stream N.
OUTREC	Print the output buffer.
INPREC	Read a record to the input buffer.
IOBUFF (A)	Initialise the I/O buffer to contain the character in the left-most position of A. (Standard setting is to blanks). e.g. CALL IOBUFF ('/')
IOI	Set I/O buffer to blanks.
SETBFR (N)	This causes a special set of fixed block length transfer routines to be used instead of variable length transfers when writing to magnetic tape. It is only useful when a known maximum number of words is being transferred at any one time, and can be inefficient if there is a wide variation in the number of words being transferred at any one time. SETBFR sets the buffer length to N.
OUTDEL (N)	Delete (i.e. destroy) the current output on output stream N.
DUMP (X, Y)	Dumps storage from location of X to location of Y.
PDUMP (A, B, N)	Dumps storage from address of A to address of B (or B to A if B has the lower address). If N = 0, format is as machine instructions, with the address in octal. If N = 1 words are printed all in octal (i.e. as data words) A and B are simple, or subscripted variable names, or routine names. e.g. CALL PDUMP (A(1), A(500), 1) CALL PDUMP (THETA, PI, 0)
LOC(N) (Function)	M = LOC (N), the location (address) of N is placed in M. Alternative names are LOCF and XLOC.
RMCTR(A)	This places, in A, the number of instruction interrupts left for this job.
TIME(A)	Places the time since the start of the job in A (as a real number of seconds).
OVFLOW (O) (Function)	Can be used as a logical function to test the overflow indicator and set it to FALSE.

Name, and form of arguments	Properties
ELAPSE (A)	Gives the time since the last call of ELAPSE, (or from the start of the job). The result is in A as a real number of seconds.
TXCLOK(A) or TXTIME(A)	Gives, in A, a text version of the time of day, suitable for output on A8 conversion. e.g. 09.47.33
TXDATE(A)	As TXCLOK, but gives date in A8 format. e.g. 11/07/67
TAPE5(I)	For reading five track paper tape. Sets I (as a Fortran integer) to the next character on the currently selected input stream (INSPEL (above) should normally be used to do this). There are two modes: - (i) If read as a binary document then the bits (i.e. the holes are 1 bits) of the character form the value of I, so that I is within the range 0 to 31. For an end of record, I is set to 32. For the physical end of tape, I is 33. The least significant punch is the one at the edge of the tape, on the side which is two holes from the sprocket holes. (ii) If read as a B.C.D. document, then I is set to minus the internal code number of the character read. (See Appendix 2.) e.g. If the character A is read I is set to - 33.
TAPE7 (I)	As TAPE5, but for reading seven track paper tape. If read as a binary document $0 \leq I \leq 127$ I is 128 for end of record, and 129 for physical end of tape. The least significant punch is the one at the edge of the tape, on the side which is three holes from the sprocket holes.
TAPE6 (I) TAPE8 (I) (continued)	For reading six or eight track paper tape (on currently selected input stream) which must be presented as a binary document. I is set to a Fortran integer formed of the bits (holes) of the number punched. For 6 track $0 \leq I \leq 63$ For 8 track $0 \leq I \leq 255$ For end of record or I = 64 (6 track) I = 256 (8 track)

Name, and form of arguments	Properties
TAPE6 (I) TAPE8 (I) (continued)	Note that because of spurious newlines at the start of the tape, end of record characters may appear before the tape proper is read. For eight track tape, the least significant punch is the one, on the edge of the tape, on the side which is three holes from the sprocket holes.

APPENDIX 8 THE JOB DESCRIPTION

Note: the material in this Appendix is dependent on the Atlas Supervisor program, and is subject to change from time to time. Up to date information is available from Atlas Computing Service.

Every Fortran V job presented to Atlas must be preceded by a Job Description, which states, among other things, the output devices to be used, the amount of output to be produced, and the amount of time to be allowed for compilation and execution.

A brief explanation of Job Descriptions is given below. This explanation will be found sufficient for most purposes, but further details are given in Reference 8.

Each Job is presented to the computer as one, or more, documents. These are self contained blocks of information presented to the machine through one input device without a break. Each document is preceded by a heading, followed by a document title; and is ended by a card punched with 7 and 8 in column one, or by a paper tape line beginning with *** (this line is usually ***Z). The majority of jobs involve only one input document.

A8.1 THE DOCUMENT TITLE

A title is punched on one card or one paper tape line. It must be different from the title of any other document present in Atlas at the same time.

It must not start with a blank, comma, point or the word END. The first characters of the title should be the user's job number.

A8.2 THE DOCUMENT HEADING

This is one of: -

JOB (preceding a job description document).

or

COMPILER FORTRAN (or USE FORTRAN) (Preceding a program document).

or

DATA (preceding a data document).

A8.3 THE JOB DESCRIPTION

Where appropriate, the Job Description must contain the information dealt with below. Each line starts in column one, and may be punched on cards or paper tape. The various sections which may be contained in a Job Description are given below. Note: - a maximum of seven input and/or output streams may be used by one job.

A8.3.1 The OUTPUT section

This section specifies the kind of output devices to be used, and the maximum amount of output to be produced on each output device (or stream). (See also section 7.2.1).

If a device is used which is not specified, or if the specified quantity of output is exceeded, then the job is terminated with execution error 9: OUTPUT NOT DEFINED or OUTPUT EXCEEDED (see section 10.3).

The output section of the Job Description consists of the word OUTPUT followed by a list of the logical numbers of all output documents (i.e. the numbers by which they are referred to in the program), followed by the output device name, followed by the maximum quantity of output to be produced by that device. The logical numbers used for output streams must be within the range

$$0 \leq \text{number} \leq 15.$$

Examples:

a) The presence of a statement

WRITE (6, fmt) list

in the program, where this statement is required to produce output on a line printer, would require a Job Description specification of:

```
OUTPUT
6 LINEPRINTER n LINES
```

where n is the maximum number of lines to be produced on output stream 6. This could be written on one line as:

```
OUTPUT 6 LINEPRINTER n LINES
```

b)
```
OUTPUT
 0 LINEPRINTER 500 LINES
 6 LINEPRINTER 1000 LINES
15 CARDS 120 LINES (i.e. 120 cards)
 8 CARDS 2000 LINES
 9 SEVEN HOLE PUNCH 2 BLOCKS
10 FIVE HOLE PUNCH 80 LINES
```

The term "LINES" means records (see section 7.3). One "BLOCK" is equal to 512 words each of 8 characters (this is usually equivalent to about two printed pages).

Note: Output streams 0 and 15 are used by the Fortran V compiler. Source (and object) listings are produced on output stream 0, and object cards are punched on output stream 15.

Output stream 0 is usually a line printer (but it need not be), output stream 15 should always be specified as a card punch if object cards are to be punched (see section 12.2).

In addition PRINT statements access output stream 0, and PUNCH statements access output stream 15.

In order to write the Job Description, it is necessary to know how much output is likely to be produced by the compiler. If a routine contains P lines (statements and comments and continuation lines), then the following output is likely to be produced on stream zero: -

(i) 100 lines; plus

(ii) if SOURCE is specified: 2P lines; plus
(iii) if LIST is specified: 4P lines; plus
(iv) if MAP is specified: 50 lines

In addition, on output stream 15:-

(v) if CARDS (or CARDSn) is specified: 2P/3 lines. (This is approximate).

Thus, for a program containing 300 statements for which it is required to produce SOURCE listings, and object CARDS for each routine, and a loading MAP, a minimum specification of:

```
OUTPUT
0 LINEPRINTER 700 LINES
15 CARDS 200 LINES
```

would be required.

Note that any cards punched on stream 15 by the program (by PUNCH, or WRITE (15, fmt)) would have to be added to the above estimate; similarly with any program output on stream zero (by PRINT, TRACE, OUTPUT or WRITE (0, fmt), statements).

The above proportions of output produced to number of source statements are approximate, and may be exceeded under some circumstances: e.g. if a job consists of a large number of short routines. However, a job is not likely to exceed the above estimates by more than 50 per cent.

A8.3.2 The INPUT section

A description of the way in which input streams are specified is given in section 7.2.2.

Normally, the program itself is read on input stream zero, and this input stream is not usually specified in the Job Description.

If input statements in the program refer to input streams other than zero, then these streams must be specified in the Job Description.

e.g. READ (5, fmt) list

This refers to input stream 5, which could be specified in the Job Description as:

```
INPUT
5 LXP932XY, BOND NUMBERS
```

There would then exist a separate document with the heading and title:-

```
DATA
LXP932XY, BOND NUMBERS
```

The title following the stream number in the Job Description must match exactly with the title following the DATA line.

The job cannot be run until all input documents specified in the Job Description are present in the machine.

The logical numbers used to refer to input devices (streams) must be within the range:-

$$0 \leq \text{number} \leq 15$$

and this number may be the same as a logical number used for an output stream.

Data read by READ statements on input stream zero may follow the *ENTER directive and do not need an INPUT specification in the Job Description.

Example:
```
INPUT
 5 LSR584X3, SMITH SUEZ DATA
 7 LSR584X3, SMITH HUNGARY VALUES
15 LSR584X3, SMITH VIETNAM STATISTICS
```

This would require that the following documents be provided

```
DATA
LSR584X3, SMITH SUEZ DATA
    .
    .
    .
and DATA
    LSR584X3, SMITH HUNGARY VALUES
    .
    .
and DATA
    LSR584X3, SMITH VIETNAM STATISTICS
    .
    .
```

Note that the input stream number is not given in the data document itself (although it could be mentioned in the title if desired), and that the type of input device is nowhere mentioned.

The title(s) used for data documents must be different from the title of any other document present in the machine at the same time, hence data document titles must be different from job titles. In order to avoid confusion with other jobs, the job number and user's name should be present in all data document titles.

A8.3.3 Magnetic tapes

(1) One Inch (Ampex) Tapes

As described in section 7.2.3, when a magnetic tape is used for input or output (or both), the word TAPE must appear in the Job Description.

Example: READ (10) list

Would require a specification such as:-

```
TAPE
10 LS010*PERMIT
```

Where LS010 is the title (or reel number) of the magnetic tape (this is physically present at the start of the tape); and the * means that what follows is a comment to the machine operators. The usual comments are

```
    PERMIT
or
    INHIBIT
```

INHIBIT means that the tape is to be file protected; i.e. the tape can only be read, and will not be written to. If an attempt is made to write on an inhibited tape, then the job will be terminated on execution error 9: WRONG TAPE MODE.

PERMIT means that the tape can be written as well as read.

A8.3.3 The logical number used for a one-inch magnetic tape must be within the range
cont

$$0 \leq \text{number} \leq 99$$

The number should not be the same as the logical number of any input or output stream.

The titled magnetic tapes described above are hired by the user and are not used by other programs. Sometimes however, it may be desired to use a (one inch) magnetic tape for intermediate storage while the job is being run, and it may not be necessary to retain the tape after the job is finished. In such cases a COMMON (scratch) tape is specified in the Job Description.

e.g. TAPE
COMMON 10

Common tapes are mounted in "PERMIT" mode (i.e. they are not file protected).

A common tape cannot be retained by the user once the job is finished and it may be overwritten by other jobs.

(2) Half Inch (I.B.M.) Tapes

Half-inch tapes, which are compatible with I.B.M. 7 track tapes, may be used for formatted (B.C.D.) I/O operations on Atlas. These tapes cannot be used for unformatted (binary) input or output.

The logical number used for a half-inch magnetic tape must be within the range

$$0 \leq \text{number} \leq 15.$$

Not more than two half-inch tapes may be used in any one job.

Example: READ (3, 10) list

when used to access a half-inch tape, would require a specification such as:

TAPE IBM (to indicate a half-inch tape)
3 LP008 SMITH DATA 556 DENSITY INHIBIT

In this example, LP008 is the reel number of the tape, and SMITH DATA is the title of the tape. These are not normally written on the tape itself, and hence cannot be checked by the Supervisor, but they must appear in the Job Description so that the correct reel can be mounted. If desired, a title record could be written on to the tape, and checked by each program that uses it, but there is no automatic facility for doing this.

The words 556 DENSITY indicate the density at which the tape is to be read, or written. All operations on the tape must be performed in the same density.

There are three densities: 200, 556, and 800 six-bit characters to an inch. On the London Atlas, these densities are referred to as low, medium, and high density respectively; but at some installations high density refers to 556.

INHIBIT has the meaning described in (1) above.

(3) Lengths of magnetic tapes.

A new one-inch magnetic tape contains about 5000 blocks each of 512 forty-eight bit words.

A new half-inch tape can contain the equivalent of:

at 200 density: 1300 blocks
at 556 density: 3500 blocks
at 800 density: 4900 blocks.

On half-inch tapes, each record is terminated by a $\frac{3}{4}$ inch inter-record gap; so that if short records are written the amount of information which can be contained in the tape is considerably reduced.

A8.3.4 COMPUTING time

The amount of time (or instructions) to be allowed for the compilation of a job must also be specified. The time includes time spent in execution, and time taken to compile and load the program. If the specified time is exceeded the job is terminated with the message C TIME EXCEEDED.

Examples: COMPUTING 1 MINUTE
COMPUTING 1.2 MINUTES
COMPUTING 30 SECONDS
COMPUTING 20000 INSTRUCTIONS

One INSTRUCTION is in fact, equivalent to 2048 basic machine instructions. About 10000 "INSTRUCTIONS" occur in one minute, so that

COMPUTING 1 MINUTE

is roughly equivalent to

COMPUTING 10000 INSTRUCTIONS

The time spent in execution must be estimated by the user, but a guide to the time likely to be spent in compilation of a source program containing P lines (statements) is:-

150 instructions
plus If no listing or object cards are produced: 5P instructions
plus If a SOURCE listing is produced: P/2 instructions
plus If object CARDS are produced: P/2 instructions
plus If an object LIST is produced: 5P instructions
plus $\frac{1}{2}$ instruction for each object card loaded
plus If execution is required: 100 instructions.

Thus, for a job containing 800 source statements, requiring a source listing and object cards, the number of instructions spent in compilation and loading would be about

150 + 4000 + 400 + 400 + 100

= 5050 INSTRUCTIONS

i.e. about 30 SECONDS

The figures given above are only approximate and may be exceeded under certain circumstances: e.g. where a job consists of a large number of short routines, or where there are a large number of complicated statements. However, no job is likely to be as much as twice as slow as the above estimates.

A8.3.5 EXECUTION time

When a job uses magnetic tapes, a certain amount of time is spent in manipulating (e.g. REWINDing) these tapes. This tape manipulation time is specified separately from the COMPUTING time, and is known as EXECUTION time. The difference

between EXECUTION and COMPUTING times is usually small, but will vary according to the number and type of the jobs in the machine.

A reasonable guide is to specify 50 per cent more execution time than computing time.

Examples: EXECUTION 5 MINUTES
EXECUTION 2.4 MINUTES
EXECUTION 100000 INSTRUCTIONS

Note that the Fortran V compiler does not itself use magnetic tapes, and therefore execution time need only be specified when the program itself uses them. If the execution time is exceeded, then the job is terminated with the message

E TIME EXCEEDED

A8.3.6 STORE requirements

This specification gives the maximum number of storage blocks (each of 512 words) which are to be used:

(i) in execution (n_1)

and (ii) in compilation (n_2)

The form is

STORE n_1/n_2 BLOCKS

If n_1 is not specified (i.e. if the STORE line is omitted) then 20 blocks are allowed in execution. If n_2 is not specified, the line is written as

STORE n_1 BLOCKS

and 80 blocks are allowed in compilation. Note this standard allocation is subject to any changes made in the Supervisor. 80 blocks is sufficient for programs up to about 400 statements, if the program is longer than this then the allowance (n_2) should be increased at the rate of 3 blocks for every 100 statements above 400. Again, this is an approximation, and may not always be sufficient. However, programs are not likely to exceed the above estimate by more than 10 blocks.

Compile store is dependent on the size of the whole program (plus object routines), and not merely on the size of the largest routine. There is thus no limit on the amount of storage which could be required for a compilation.

Savings in compilation storage may be effected by using the COMPILE option on the *RUN directive (see section 12.1); if this is done the compile store used will depend only on the length of the largest routine.

When calculating execution storage, 6 blocks should be allowed for library routines, and one block for each input and output stream.

If either of the storage specifications is exceeded, then the job is terminated with the message EXCESS BLOCKS.

A8.3.7 A complete job description

Examples:

(a) A simple Job Description is:

```
JOB
LXS99PQR, WELLER BUBBLE CHAMBER
OUTPUT
0 LINE PRINTER 1000 LINES
15 CARDS 200 LINES
COMPUTING 1 MINUTE
COMPILER FORTRAN
*RUN...etc.
```

(b) A more complex job:

```
JOB
LXS99PQR, WELLER ROTATE AXIS
OUTPUT 0 LINEPRINTER 2000 LINES
6 LINEPRINTER 1000 LINES
15 CARDS 2600 LINES
14 CARDS 200 LINES
3 SEVEN HOLE PUNCH 5 BLOCKS
INPUT
5 LXS99PQR, WELLER AXIS DATA
3 LXS99PQR, WELLER STREAM 3DATA
COMPUTING 70000 INSTRUCTIONS
EXECUTION 100000 INSTRUCTIONS
TAPE
4 LS101*INHIBIT
TAPE IBM
1 LS002 WELLER IBM VALUES 556 DENSITY PERMIT
STORE 65/100 BLOCKS
COMPILER FORTRAN
* RUN etc.
```

The input documents for streams 5 and 3 must be provided.

The order of the lines in the Job Description is not significant provided that all OUTPUT, all INPUT, and all TAPE specifications are kept together.

REFERENCES

1 American Standard FORTRAN U.S.A. Standards Institute. X.3.9 March 7 1966.

2 Atlas Fortran Manual (Part 1) by E. J. York. HMSO AERE R 4599 (1964).

3 IBM 7090/7094 IBSYS Operating System. Version 13. Fortran IV language. July 1965. Form No. C28-6390-1 File No. 7090-25.

4 IBM 7090/7094 Programming Systems. Fortran II Programming. August 1963. Form No. C28-6054-4. File No. 7090-25.

5 IBM System/360. Fortran IV language. 1966. Form No. C28-6515-4. File No. S360-25.

6 Science Research Council. Atlas Computer Lab. Hartran System Note No. 4. May 1965.

7 I.C.T. Atlas 1 Computer. Programming manual for A.B.L. List CS 348A. January 1965.

8 I.C.T. Ltd., "Preparing a Complete Program for Atlas I" TL 1254. List CS460 March 1966.

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

Designed and produced by A. B. Saunders Associates Ltd
Printed in Great Britain by Megaron Press Ltd