FP6000
computer system
FORTRAN

Ferranti ELECTRONICS

A DIVISION OF FERRANTI-PACKARD ELECTRIC LIMITED

INDUSTRY STREET, TORONTO 15 - ONTARIO - CANADA

St. John's
Halifax
Saint John
Sherbrooke
Trois-Rivières
Chicoutimi
Montreal
St. Catharines
Fort William
Winnipeg
Regina
Calgary
Vancouver

FORTRAN Memorandum #1

Additions and Errata

This document is provided to explain a number of changes to FP6000 FORTRAN which have been made since the manual was published and also to list certain errors which have been discovered in the manual.

1. Section 1.3

- The range of Real numbers is approximately -5.6×10^{76} to 5.6×10^{76} and not as stated.

2. Section 1.6

- rule 5 is not correct. Meaning will be assigned to a sequence of multiplications and/or divisions, or additions and/or subtractions as though they were to be performed from left to right. The expression however, may be evaluated in any sequence which does not alter this meaning. Therefore, the expression A * B / C might be evaluated as A multiplied by B and then divided by C or as B divided by C and then multiplied by A. The choice which will be made by FORTRAN depends on the context of the expression.

3. Section 1.6

- rules 6 and 7 are now relaxed. Integer and real quantities may be mixed in any way in an expression. In such circumstances the integer quantities will be converted to real and hence the whole expression will have a real result. If more than one integer quantity appears then they may be operated on as integers or may be separately converted to real. Thus the statement A = B + I + J is preferably written A = B + (I + J) to ensure that I and J are added as integers. The form A = (B + I) + J would separately convert I and then J to real and would be slower.

4. Section 2.2.

If more than one variable appears on the left side of an arithmetic statement they must be of the same mode. That is, the extension allowed in paragraph 3 above does not apply here. For example, A, I = X + Y is illegal.

5. Section 2.3.

The explanation of integer division is incorrect in view of paragraph 2 above. Expressions involving integer divisions should always include parentheses to resolve any doubt about the sequence of operation. Note particularly the dangers of including division with mixed mode such as A = B * I / J. It should also be noted that in statements such as A = I / J the division is done in integer mode and hence is truncated even though the result is then converted to real mode.

6. Section 5.2.

In addition to an END statement at the end of each segment, the statement "FINISH" must follow the END statement of the last segment. If the program is prepared on paper tape at least 3 "newline" characters must follow the FINISH statement.

7. Section 7.3.

The overflow indicator is "set" by any arithmetic operation which produces a number outside of the permitted range or by standard functions for undefined ranges of the argument (e.g., square root of a negative argument) and remains set until tested by the IF OVERFLOW statement. The overflow indicator is cleared (and preserved) by entry to a subroutine or function segment. After the operation of the subroutine or function segment, the overflow indicator is set if: -

1) it was set prior to entering the segment or
2) if it was set (and not subsequently cleared) within the segment.

8. Section 9.1.

The value i given in a PERIPHERAL statement may be either numeric or alphanumeric as desired. In fact, it may be any string of at most 40 letters and digits of which the first need not necessarily be a letter.

Additions and Errata:

9. Chapter 9.

An additional facility is provided for numeric input by the I, E or F specifications. If w = 0 in any of these specifications then the field width to be used is determined from the input data as follows:-

1) any initial spaces or spaces between the "E" (if any) and the following digits or sign are ignored.
2) any other space or "tab"or "newline" is taken as the end of the field.

This allows greater freedom in preparing data for input since exact spacing is not required. It is very dangerous to mix w = 0 and w ≠ 0 specifications in the same line.

10. Chapter 9.

The "newline" character (applicable only to paper tape) will serve to terminate any numeric field whether w = 0 or not. Then:-

1) If some digits have been found, they are taken as the current number. If a further numeric field is read from the same record then 2) below applies to this next field.

2) If no digits have yet been found, a new record is read and scanning of the new record is commenced at the beginning.

Difficulties can arise because either a newline or the closing bracket (or /) of the FORMAT statement can cause the reading of a new record. For example, if with a statement such as:- FORMAT (10I5), only 8 numbers are given in the first record, followed by a "newline", then at most two fields can be read from the following line. A useful format statement for numeric input only is:-
FORMAT (1000I0) in which the repetition number is sufficiently large not to be exhausted.

11. Chapter 9

- An additional format specification is provided to control vertical paper motion. This is applicable to paper tape (applies when the tape is printed) and to the line printer.

The V format specification, written nV, where n is an integer ($0 \leqslant n \leqslant 10$) controls the paper motion to precede printing of the line being formed when the V format spec. is encountered. Note that the V spec. does not cause a new record to be begun as does / or the final parenthesis. The action for various values of n is as follows:-

n	Line Printer	Paper Tape
0	Single feed	Single feed
1	Skip to Chan. #1	Form feed
2	Skip to Chan. #2	Ver. Tab
3	Skip to Chan. #3	Single feed
4-8	Skip to Chan. #n	Single feed
9	Double feed	Double feed (2 newlines)
10	No feed	No feed (car. return)

In the absence of a V spec., 0V is assumed. For example the statement:-

FORMAT (1VI5 / 3F16.12)

causes the first number to be printed on a line at the top of the next page and the next 3 numbers to be printed on the following line. The V spec. is ignored on input.

If the V spec. is written -1V then the specification must correspond to an integer variable in the I/O list which will be used in place of n. For positive n, no I/O list variable is used.

12. Chapter 9

- If the letter Z is encountered in a numeric field, than all input caused by the current READ statement (including the field containing the Z) is terminated (as though there were no further variables in the I/O list) and all variables not yet read are unaltered.

13. Chapter 9

- The N format specification, written simply as N, must correspond to an integer variable in the I/O list.

Additions and Errata:

13. continued

For input, the variable is set equal to the number of fields read after the N spec. was encountered.

For output, the maximum number of fields to be written, following the N spec., is specified by the variable. If the I/O list is longer than this, output ceases after the specified number.

The N specification should be encountered only once in scanning the FORMAT statement.

As an example of this consider the following program which will read a group of (at most 100) numbers, and then print them followed by their sum.

```
      MASTER SUMMATION
      PERIPHERAL (IN, TR0), (OUT, LP0)
      DIMENSION A (100)
  2   READ (IN, 100) N, A

100   FORMAT (N, 100 I 0)
      SUM = 0.0
      DO 1 I = 1, N
  1   SUM = SUM + A(I)

      WRITE (OUT, 101) N, A
101   FORMAT (N IV (F20.10 ) )
      WRITE (OUT, 102) SUM
102   FORMAT (9VF20.10, 3 X 5HTOTAL)

      PAUSE
      GO TO 2
      END
      FINISH
```

14. Table 9.1

has been corrected and is re-issued herewith.

15. Chapter 9

- The new statement:-

RELEASE i

(where i is a reference to any peripheral device) may be used if the peripheral is no longer required and will make it available to another program.

16. Chapter 9

- The new statement:-

DISENGAGE i

(where i is a reference to a paper tape reader) will switch the reader "off-line". The next attempt to use the reader will stop the program until the operator has manually switched the reader back on-line. This can be useful in changing from one reel of paper tape to another.

17. Section 9.7

- Binary READ and WRITE is not yet available.

18. Chapter 11.

More extensive monitoring facilities are in preparation but cannot be described in detail at this time.

TABLE 9.1

INPUT		
FIELD SPECIFICATION	EXTERNAL DATA	INTERNAL NUMBER
E15.6	+12345678	+ 12.345678
	1234.5678E-2	+ 12.345678
	-1.2345678E5	- 123456.78
	.012345678	+.012345678
	-1.2	-1.2
	+1234-3	+.000001234
F 15.6	+12345678	+ 12.345678
	1234.5678	+ 1234.5678
	-1.2345678	- 1.2345678
	.012345678	+.012345678
	-1.2	-1.2
	+1234	+.001234
3PF 15.6	+12345678	+.012345678
	-1.2	-.0012
	1234	+.000001234
OUTPUT		
	INTERNAL DATA	EXTERNAL NUMBER
E15.6	+12345678.	.123457E 8
	+1234.5678	.123457E 4
	-1.23	-.123000E 1
	+.000123	.123000E -3
3P E15.6	+1234.5678	123.456780E 1
	-1.23	-123.000000E -2
	+.000123	123.000000E -6
F 15.6	+12.345678	12.345678
	-1.2345678	-1.234568
	+.012345678	.012346
	-1.2	-1.200000
3PF 15.6	+12.345678	12345.678000
	+.012345678	12.345678
	-1.2	-1200.000000

FORTRAN Memorandum #2

FORTRAN Standard Functions

1. The number of standard functions can be expected to increase as others are added in the future. Those currently provided are:-

SQRTF
ALOGF
EXPF
SINF

COSF
ATANF
ABSF

IABSF
INTF
NINTF
REALF

In all cases, except as noted below, an accuracy of better than eleven decimal places of eleven figures, (whichever is least accurate) is retained.

2. SQRTF
- this provides the real, positive square root of a real, positive argument. If the argument is negative, the overflow indicator will be set and the square root of the negative of the argument will be given. The accuracy is better than eleven significant figures (not decimal places) over the whole range.

3. ALOGF
- this provides the real natural logarith of a real argument. If the argument is negative, the overflow indicator will be set, and the logarith of the negative of the argument will be given. If the argument is zero, the overflow indicator will be set and the result is not defined.

4. EXPF
- this provides the real exponential function of a real argument. An accuracy of 11 + significant figures is given. For large arguments, the overflow indicator is set and the result is not defined. For large negative arguments the result is given as zero.

5. SINF
- this provides the real sine function of a real argument given in radians. The accuracy of 11+ significant figures in the vicinity of the root at argument = 0 and 11 + decimal places elsewhere. For extremely large arguments, accuracy will decrease since the accuracy of the useful portion of the argument falls off.

6. COSF
- this provides the real cosine function of a real argument given in radians. See SINF above for accuracy with large arguments.

7. ATANF
- this provides the real inverse tangent function in radians of a real argument. The accuracy is 11 + significant figures throughout. The result given is the principal in the range $-\pi/2$ to $\pi/2$.

8. ABSF
- this provides the real absolute value of a real argument. Accuracy is exact.

9. IABSF
- this provides the integer absolute value of an integer argument. Accuracy is exact.

10. INTF
- this provides the integer, integral part of a real argument, that is the integer equal to or next smaller than the argument. The overflow indicator will be set if the result is out of range. This function may be used to convert a real variable or expression to integer where only an integer expression is allowed. (e.g. as a subscript)

11. NINTF
- this is similar to INTF except that the nearest integer is provided.

12. REALF
- this provides the real representation of an integral argument. This function may be used to convert an integer variable or expression to real where only real expressions are allowed, e.g. X = SINF (REALF (I)).

FORTRAN Memo # 3

March 5, 1964

FORTRAN Description of Operation

1. Introduction:

1.1 The process of preparing a FORTRAN source program for operation is divided into three steps:- Compilation, Consolidation and Loading. At present these three steps are distinct and occur sequentially. It is expected that the first two will be combined in the near future (by May 1964).

1.2 This description also applies to AUTOCODER operation which differs only within the first step (Compilation).

1.3 Only paper tape operation is described although punched card operation differs only in minor respects.

2. Storage Assignment

2.1 All FORTRAN programs are divided into segments. Even though the source program may consist of only one segment, other segments are added to provide for arithmetic and input/output operations and for standard functions. The compilation of each segment is done independantly.

2.2 Each segment will refer to working space locations, variables, constants, arrays etc. which are associated with the segment but which are stored separately. i. e. not necessarily adjacent to the program part of the segment. In addition, segments may refer to storage areas which are shared by other segments. Since segments are compiled separately it is not possible, at the time of compilation, to assign final addresses to instructions. The address assignment made at the time of compilation is therefore in two parts:
1) The address relative to the beginning of an area of storage and
2) An identification of the area. Final addresses are not produced until the program is loaded into storage for execution at which time the starting addresses of areas are known and can be added to the addresses of the appropriate instructions.

2.3 Storage areas may be classified in the following three ways resulting in eight possible combinations.

1) Private or Shared - Private areas may be referred to by only one segment. Shared areas are available to all segments.

2) Upper or Lower - Upper areas may lie anyplace in storage and are referred to only by branch instructions or by indexing. Lower areas are located below location 4096 so that they may be referred to by 12 bit addresses.

3) Preset or Non-preset - Preset areas contain data or program which must be defined prior to execution (e. g. program or constants). Non-preset areas need not contain defined data (e. g. working space or variables) and are cleared to zeros prior to execution.

2. 4 Private storage areas are limited to 10 per segment and are as follows.

#	Symbol	Class	Use
1	W	Lower-Nonpreset	Working space
2	V	Lower-Nonpreset	Variables
3		Lower-Nonpreset	Spare - not used
4	C	Lower-Preset	Constants
5	L	Lower-Preset	Literals (not used by FORTRAN)
6		Lower-Preset	Spare - not used
7	UC	Upper-Preset	Upper constants and miscellaneous
8		Upper-Preset	Spare - not used
9	UV	Upper-Nonpreset	Upper Variables (Arrays)
10		Upper-Nonpreset	Spare - not used

The size of each of the above areas is determined by the requirements of the segment. One more area (Symbol P) is provided per segment to store the program for that segment but is essentially a shared area because it must be referred to by other segments.

2. 5 Shared storage areas are not limited in number and are always associated with a name for identification. This name is the name of the program segment or of the common area or public variable concerned (blank common is named %).

3. Format of binary tapes

3. 1 All paper tapes involved in a program, with the exception of the original source tape, are in binary form. They are divided into "blocks" each consisting of at most 80 characters punched or read in binary mode. Each block is terminated by a "newline" character.

3. 2 The first character of each block determines the type of block as follows:-

Char.	Octal	Type
3	03	Object program title block
0	00	Object program data block
4	04	Object program terminating block
+	33	Leader title block
/	37	Leader cue block
(	30	Priority and name block
,	34	Leader terminating block
=	15	Consolidated leader relativizer block
>	16	Consolidated leader parameter block
?	17	Consolidated leader cue block
<	14	Consolidated leader terminating block
;	13	Pause block
[	73	Executive block

3.3 The second character of each block indicates the number of words of the block to be checksummed. That is - one of the first n words (normally the last) of the block is such that the sum of the first n words is zero. The value of n is given in the second character (n may be zero).

3.4 Executive blocks are used to record binary information to be read by EXECUTIVE. Programs produced by the "DUMP" directive in EXECUTIVE and to be read by the "LOAD" or "DO" directive are recorded as a sequence of blocks in this form. The request slip is a special case of this. Further details of executive blocks is not given here.

3.5 Object program data blocks contain the instructions and constants etc. of a semi-compiled program. Further details of their structure is not given here.

3.6 In all other block types, the portion between the second character and the check sum is divided into "fields" which may be of two types:- 1) Octal numbers (Base 8) or 2) Alphanumeric field which may consist of any character except space. An octal field may have leading spaces followed by up to 8 octal digits and is terminated by any character which is not an octal digit. An alphanumeric field may have leading spaces followed by up to 40 characters and is terminated by space. Thus these types of block are reasonably easy to interpret if printed.

4. Compilation

4.1 The output of the compiler, for each segment of source program, consists of a semi-compiled program followed by a "leader" (The term leader is retained from a previous scheme although it now follows the segment) and is made up of the following blocks:-

1) A length of blank tape
2) An object program title block
3) A number of object program data blocks
4) An object program terminating block
5) A length of blank tape
6) A leader title block
7) A number of leader cue blocks
8) A priority and name block (Master segment only)
9) A leader terminating block

4.2 The object program title block consists of:-

1) An octal field denoting "type"
2) An alphanumeric field giving the name of the segment.

The type is octal 400 for master segments and zero otherwise.

4. 3 An object program terminating block consists of ten octal fields giving the sizes of storage areas required in the ten private categories described in section 2. 4

4. 4 The leader title block is identical to the object program title block with the exception of the first (type of block) character.

4. 5 A leader cue block consists of:-

1) An octal field giving "type and value"
2) An alphanumeric field giving "name".

The leader will contain one cue for each shared area and each peripheral device referred to by the segment. The "type and value" field is an eight octal digit field (leading zeros may be omitted) of which the first 3 octal digits define "type" and the last 5 digits = value. The possible types are as follows:-

Type 000 = blank cue - value = 00000
This indicates that the segment refers to an area segment or peripheral without defining anything about it.

Type 410 = program cue - value = size of program in words.
In this case "name" is the name of the current segment and the cue is used to define the length of the program storage required.

Type 020 = entry point cue - not used by FORTRAN (AUTOCODER only).

Type 030 = area cue - value = size of area in words.
This indicates that the segment refers to a common area or public variable of the given name and defines the size of the storage area required.

Types n30 where $n \neq 0$. These are other types of carea cues not used by FORTRAN.

Type 040 = peripheral cue - In this type, the second last digit of value defines the type of peripheral (0 = Tape reader, 1 = Tape punch etc.) and the last digit denotes the unit number. One peripheral cue is produced for each peripheral definition made in a PERIPHERAL statement.

4. 6 A priority and name block consists of

1) An octal field giving the priority of the program. In the case of FORTRAN this is always given as zero (later steps assume zero priority to be undefined and assign priority = 50)
2) An alphanumeric field giving the name of the program to be used by EXECUTIVE.

4.7 A leader terminating block is identical to an object program terminating block (see 4.3) with the exception of the first (type of block) character.

4.8 When the FINISH statement is encountered, A length of blank tape is produced followed by a pause block.

5. Consolidation

5.1 The consolidation process is performed by a program called CNSI. This program determines the final storage allocations using the information provided by the leaders of all the segments involved. CNSI therefore reads the semi-compiled tapes for all the segments required to form a complete program.

5.2 As the title block (either object program or leader) of a segment is read, the segment is either accepted or rejected. If the segment is accepted it is copied to the output punch and the leader is used in determining the final storage. If the segment is rejected it is merely skipped.

5.3 A segment will be accepted if:-

1) It is the first segment read by CNSI
or 2) If one or more blank cues have been found in the leaders of previously accepted segment with the name of this segment and no non blank cues with this name have already been found.
or 3) If the segment is a master segment and no non blank cue of the same name has already been found.

5.4 If a pause block is read by CNSI it types a message on the console typewriter giving the name of one blank cue i.e. of a segment it has not yet read.

5.5 When no blank cues remain, all necessary segments etc have been found. CNSI then punches:-

1) A length of blank tape
2) A pause block
3) A length of blank tape
4) A request slip
5) A copy of GPI
6) A length of blank tape
7) A consolidated leader
8) A length of blank tape
9) A pause block

The portion of the output tape following 2) above is then removed and spliced, omitting 9) above, to the front of the tape to form the final program tape.

5.6 The request slip is an executive block which specifies to executive the name and priority of the program and the peripheral and core storage requirements

5.7 GPI is a program (General Purpose Loader) which is read by executive and which in turn reads the rest of the tape.

5.8 The consolidated leader contains all the information required by GPI to enable it to load the program. The consolidated leader consists of:-

1) A consolidated leader relativizer block
2) A consolidated leader parameter block
3) A number of consolidated leader cue blocks
4) A consolidated leader terminating block

5.9 The consolidated leader relativizer block consists of ten octal fields which give the starting addresses of the storage areas allotted to each of the ten private storage catagories. As each segment is read by GPI these addresses are incremented by the amount used by that segment (derived from the object program terminating block) in readiness for the next segment.

5.10 The consolidated leader parameter block consists of three octal fields which represent:-

1) Offset - the amount by which the program must be moved as it is loaded and later moved into correct position.
2) An instruction to be obeyed by GPI after loading is complete - this is either zero (which has no effect) or is an instruction to release the tape reader if it is not used by the program after loading.
3) The name of the program (4 characters) expressed as 8 octal digits.

5.11 A consolidated leader cue block is similar to a leader cue block except that the "value" is now the starting address of the segment or area.

6. BINT

6.1 BINT is a program which will accept, as input, a semi-compiled program and which will produce an interpretation of all object program data blocks. All other types of blocks will merely be copied. The form of the interpretation is as follows:-

Col 1. Location of constant or instruction relative to start of area.

Col 2. Identity of area to which col 1 refers.

Cols 3-6 Contents of above location printed as an instruction address is relative to start of an area.

Col 7. Identity of area to which col. 6 refers.

Col 8. Same data as cols 3-6 but printed in octal.

7. Example

7.1 The following pages give an example of the stages in preparing a complete program.

7.2 The first page shows the source program which consists of two segments.

7.3 The next four pages show the output produced by BINT. Note that title blocks and leaders etc. are directly reproduced.

7.4 The last page shows the request slip and consolidated leader produced by CNSL.

```
C     THIS PROGRAM READS A LIST OF AT MOST 1000 NUMBERS AND
C          PRINTS THEM FOLLOWED BY THEIR AVERAGE AND RMS


      MASTER PROGRAM EXAMPLE
      PERIPHERAL (IN,TR0),(OUT,TP0)
      DIMENSION A(1000)
      PUBLIC AVGE,RMS
      READ (IN,100) LENGTH,A
100   FORMAT (N,1000F0)
      CALL SUMMATION (LENGTH,A)
      WRITE (OUT,101) LENGTH,A
      WRITE (OUT,102) AVGE,RMS
101   FORMAT (1V(F10.4))
102   FORMAT (9VF10.4,3X4HAVGE/F10.4,3X3HRMS)
      END

      SUBROUTINE SUMMATION (N,A)
      DIMENSION A
      PUBLIC AVGE,RMS
      SUM,SUMSQ = 0.0
      DO 1 I=1,N
      EL = A(I)
      SUM = SUM + EL
1     SUMSQ = SUMSQ + EL * EL
      AVGE = SUM / N
      RMS = SQRTF (SUMSQ / N)
      RETURN
      END
      FINISH
```

```
380400 PROGRAMEXAMPLE    4    -< &          Title Block
   20        074 0        0 P     03600000  Entry branch in L20
    0 C      000 0        1       00000001  Header for array A - redefined later
    1 C      000 0        0 AVGE  00000000  } Indirect reference to
    2 C      000 0        0 RMS   00000000  } AVGE & RMS
    0 P      000 3        3 C     30000003  }
    1 P      070 1        6 FIOF  13400006  } Enter FIOF to initiate READ
    2 P      000 0        0 IN    00000000  }
    3 P      100 3        0 V     34000000  } Enter FIOF with address
    4 P      070 1        8 FIOF  13400010  } of LENGTH
    0 C      003 4 3 4094 UV      40177776  Header for A
    4 C      000 0        0 UV    00000000  } Limits of A
    5 C      000 0     2000 UV    00003720  } for READ loop
    5 P      000 3        4 C     30000004  X3 = start of array
    6 P      070 1       10 FIOF  13400012  }
    7 P      101 3        2       34040002  } Read A
    8 P      026 3        5 C     31300005  }
    9 P      074 5        6 P     53600006  }
   10 P      070 1       12 FIOF  13400014  Terminate READ
    3 C      000 0        0 UC    00000000  }
    0 UC     147 5       64       56340100  } FORMAT
    1 UC     000 0     2432       00004600  }
    2 UC     020 3     2432       31004600  }
   11 P      070 1        0 SUMM  13400000  }
   12 P      100 3        0 V     34000000  } CALL SUMMATION
   13 P      100 3        0 C     34000000  }
   14 P      000 3        6 C     30000006  }
   15 P      070 1        4 FIOF  13400004  } Enter FIOF to initiate write
   16 P      000 0        0 OUT   00000000  }
   17 P      100 3        0 V     34000000  } Address of LENGTH
   18 P      070 1        8 FIOF  13400010  }
   19 P      000 3        4 C     30000004  }
   20 P      070 1       10 FIOF  13400012  }
   21 P      101 3        2       34040002  } Write A
   22 P      026 3        5 C     31300005  }
   23 P      074 5       20 P     53600024  }
   24 P      070 1       12 FIOF  13400014  Terminate WRITE
   25 P      000 3        7 C     30000007  }
   26 P      070 1        4 FIOF  13400004  } Initiate WRITE
   27 P      000 0        0 OUT   00000000  }
   28 P      000 3        1 C     30000001  } AVGE
   29 P      070 1       10 FIOF  13400012  }
   30 P      000 3        2 C     30000002  } RMS
   31 P      070 1       10 FIOF  13400012  }
   32 P      070 1       12 FIOF  13400014  Terminate WRITE
```

```
     6 C     000 0       3 UC     00000003
     3 UC    035 0 2 1574         01663046
     4 UC    020 0   1924         01003604
     5 UC    026 3 1 1924         31313604
     7 C     000 0       6 UC     00000006
     6 UC    035 1 2 2433         11664601
     7 UC    007 0 2  284         00360434
     8 UC    076 0    296         03700450
     9 UC    035 4 2 2533         41664745
    10 UC    171 3 2   64         37460100
    11 UC    141 3   1795         36043403
    12 UC    000 7 3 2610         70035062
    13 UC    134 5 3 1650         55633162
    33 P     156 0       0        06700000
480 1 0 10 0 0 16 0 3720 0 4  =;
+80400 PROGRAMEXAMPLE   4   5< &
/941000042 PROGRAMEXAMPLE   4   )9<)
/40 FAPF 4  JZ↑Z
/54000000 IN   4 *2>
/54000010 OUT  4L)\6
/53000002 AVGE 4W5;6
/53000002 RMS  4K)Z?
/40 FIOF 4  J=←Z
/60 SUMMATION    4  <[7T
(40 PROG 4  G4←Y
,80 1 0 10 0 0 16 0 3720 0 4X =;
```

FORMAT

FORMAT

END

Terminating Block

Leader title Block

Cue defining Master seg = 34 words

Cue requesting FAPF

Cues defining peripherals

Cues defing AVGE & RMS

Cue requesting FIOF

Cue requesting SUMMATION

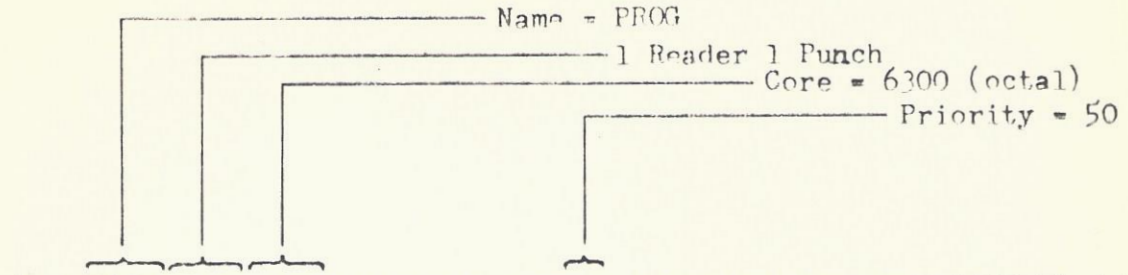
Executive name = PROG

Leader terminating block

```
370000 SUMMATION     4    ↑0Q.
     0 P     124 2          2          25200002 }
     1 P     010 1          0 W        10400000 |
     2 P     023 0 1        0          01150000 |  Prologue to obtain
     3 P     010 3 2        1 W        30420001 |  addresses of arguments
     4 P     000 1          0 W        10000000 |
     5 P     101 1          1          14040001 |
     6 P     060 2          1 P        23000001 }
     0 C     000 0          0 AVGE     00000000 }  indirect references to
     1 C     000 0          0 RMS      00000000 }  AVGE & RMS
     2 C     000 0          0          00000000 }  0.0
     3 C     000 0          0          00000000 }
     7 P     000 6          2 C        60000002 }
     8 P     000 7          3 C        70000003 |
     9 P     010 6          2 V        60400002 |  SUM,SUMSQ = 0.0
    10 P     010 7          3 V        70400003 |
    11 P     010 6          0 V        60400000 |
    12 P     010 7          1 V        70400001 }
    13 P     100 3          1          34000001 }  I = 1
    14 P     010 3          4 V        30400004 }
    15 P     000 3          1 W        30000001 }
    16 P     000 3 3        0          30030000 }  move N to 5V
    17 P     010 3          5 V        30400005 }
    18 P     000 3          2 W        30000002 }
    19 P     070 1          6 FAPF     13400006 }  calc. address of A(I)
    20 P     100 3          4 V        34000004 }
    21 P     000 6 3        0          60030000 }
    22 P     000 7 3        1          70030001 |  EL = A(I)
    23 P     010 6          6 V        60400006 |
    24 P     010 7          7 V        70400007 }
    25 P     000 6          0 V        60000000 }  X6,7 = SUM
    26 P     000 7          1 V        70000001 }
    27 P     100 3          6 V        34000006 }  Add EL
    28 P     070 1          0 FAPF     13400000 }
    29 P     010 6          0 V        60400000 }  SUM = X6,7
    30 P     010 7          1 V        70400001 }
    31 P     000 6          6 V        60000006 }  X6,7 = EL
    32 P     000 7          7 V        70000007 }
    33 P     100 3          6 V        34000006 }  Mult by EL
    34 P     070 1          1 FAPF     13400001 }
    35 P     100 3          2 V        34000002 }  Add SUMSQ
    36 P     070 1          0 FAPF     13400000 }
    37 P     010 6          2 V        60400002 }  SUMSQ = X6,7
    38 P     010 7          3 V        70400003 }
    39 P     002 6          4 V        60100004 }
    40 P     103 6          1          64140001 |
    41 P     012 6          4 V        60500004 |  End of DO loop
    42 P     001 6          5 V        60040005 |
    43 P     054 6         18 P        62600022 }
```

```
   44 P     000 6       0 V     60000000
   45 P     000 7       1 V     70000001
   46 P     000 3       1 W     30000001
   47 P     070 1      16 FAPF  13400020
   48 P     100 3      10       34000012
   49 P     070 1       5 FAPF  13400005
   50 P     000 3       0 C     30000000
   51 P     010 6 3     0       60430000
   52 P     010 7 3     1       70430001
   53 P     000 6       2 V     60000002
   54 P     000 7       3 V     70000003
   55 P     000 3       1 W     30000001
   56 P     070 1      16 FAPF  13400020
   57 P     100 3      10       34000012
   58 P     070 1       5 FAPF  13400005
   59 P     010 6       3 W     60400003
   60 P     010 7       4 W     70400004
   61 P     070 1       0 SQRT  13400000
   62 P     100 3       3 W     34000003
   63 P     000 3       1 C     30000001
   64 P     010 6 3     0       60430000
   65 P     010 7 3     1       70430001
   66 P     000 1       0 W     10000000
   67 P     072 1       1       13500001
   68 P     156 0       0       06700000
475 10 0 4 0 0 0 0 0 0 4ZSJL
+70000 SUMMATION    4   FØQ.
/841000105 SUMMATION    4   AZM,
/40 FAPF 4  J%↑Z
/53000002 AVGE 4W5;6
/53000002 RMS  4K)Z?
/50 SQRTF    4  VTL,
,75 10 0 4 0 0 0 0 0 0 4BSJL
;24 T]IP
```

- X6,7 = SUM
- L10,11 = N(real)
- Div. by N
- Store in AVGE
- X6,7 = SUMSQ
- L10,11 = N(real)
- Div. by N
- Store in 3W
- X6,7 = $\sqrt{3W}$
- Store in RMS
- Return




```
=043 50 114 114 160 160 2056 2202 2202 6224
>0564 0 60625747
?041000160 PROGRAMEXAMPLE
?041000327 FAPF
?04000000 IN
?04000010 OUT
?03006224 AVGE
?03006226 RMS
?041000576 FIOF
?041000222 SUMMATION
?041002005 SQRTF
<0
```

Consolidated Leader

FORTRAN Memo #4

March 17, 1964

Additional Standard Functions

1. The following additional standard functions are now provided.

TANF	-	tangent
COTF	-	cotangent
ASINF	-	inverse sine
ACOSF	-	inverse cosine
SINHF	-	hyperbolic sine
COSHF	-	hyperbolic cosine
TANHF	-	hyperbolic tangent
ASINHF	-	inverse hyperbolic sine
ACOSHF	-	inverse hyperbolic cosine
ATANHF	-	inverse hyperbolic tangent
DLOGF	-	logarithm-base 10
ANTILOGF	-	antilogarithm-base 10 (i.e. 10^X)
PLOTF	-	graph plotting subroutine

2. The first twelve of the above are written in terms of the previous standard functions (e.g. $\operatorname{asinh}(x) = \log(x + \sqrt{x^2 + 1})$).

3. PLOTF is described in FORTRAN Memo #5.

FORTRAN Memo #5

March 17. 1964

PLOTF

1. PLOTF is a FORTRAN Standard Subroutine designed to provide a crude graph plotting facility on a line printer. It could also be used with paper tape output.

2. A sequence of characters forming part or all of a line of print can be stored in a succession of array elements. Each element of a real array can store 8 characters and each element of an integer array can store 4 characters. Information held in character form may be printed by means of the "A" format specification. PLOTF is designed to produce information in this form.

3. The FORTRAN statement:-

CALL PLOTF (64, J, K, A(I))

will set K characters commencing with the first character of array element A(I) to spaces except that the first and every Jth character will be set to "period". If J = 0, all characters are set to spaces.

4. The FORTRAN statement:-

CALL PLOTF (I, J, K, A(I))

where $0 \leq I \leq 63$, will set the Jth character beyond the first character of A(I) to a value determined by I. (J = 0 = first character of A(I) etc.). If J is negative, its absolute value will be used. If J is then greater than K, its residue module K will be used.

5. In both cases above, the first three arguments are integer constants, variables or expressions and the fourth argument is an array element which may be either real or integer.

6. Some of the characters determined by I are as follows:-

I	character (s)
0-9	0-9
26	*
30	.
33-58	A-Z

7. The following example plots the sine and cosine functions in steps of three degrees in a space of 101 characters.

```
      MASTER PLOTEXAMPLE
      PERIPHERAL (OUT, LP0)
      DIMENSION  A(13)
      F = 3.1415926536/180.
      DO  1  I = 0, 360, 3
      X = I * F
      KS = 50 + NINTF (SINF(X) * 50.)
      KC = 50 + NINTF (COSF (X) * 50.)
      CALL PLOTF (64, 10, 101, A (1) )
      CALL PLOTF (51, KS, 101, A (1) )
      CALL PLOTF (35, KC, 101, A (1) )
1     WRITE (OUT, 2) I,  A (1)
2     FORMAT ( I4, 3XA101)
      END
      FINISH
```

FORTRAN Memo #6

March 18, 1964

Intersegment Communication

1. This memo describes certain details of FORTRAN which must be known in order to write programs partly in FORTRAN and partly in AUTOCODER.

2. Storage of Arrays

2.1 Arrays are stored in consecutive FP6000 words - one word per element for integer arrays and two words per element for real arrays. Associated with each array is an "array header" which is stored separately from the array and consists of n+1 words for an n dimensional array as follows:-

1st word:-	sign bit = 1 for integer array, 0 for real array next 8 bits = number of dimensions last 15 bits = base address of array (see below)
2nd word:-	first dimension (for $\geq$ 2 dimensions only)
3rd word:-	product of first two dimensions (for $\geq$ 3 dimensions)
"	
,	
,	
nth word:-	product of first n-1 dimensions
n+1 th word:-	total number of elements

2.2 The first element of an array is that with subscripts 1, 1, -----, the next is that with subscripts 2, 1,----. The base address given in the header is that of the non-existant element 0,0,---. For example, if a real, two dimensional array of dimensions 3,4 starts (has element 1,1) at location "A", the base address is given as A-8. It is conceivable, although unlikely, that the base address of an array is negative in which case it is given modulo 2^{15}.

3. Common Storage Areas

3.1 The FORTRAN statement-

COMMON/POOL/I, X, A(4), LIST(5,2)

establishes a common area named "POOL" of 21 words as follows:-

integer variable	I	-	1 word
real variable	X	-	2 words
real array	A	-	8 words
integer array	LIST	-	10 words

The AUTOCODER directive:-

#UPPER COMMON/POOL/ITEM, RES(10), K1(5), K2(5), Y

If an argument is a function or subroutine name, the corresponding argument word must be an instruction which will place in X3 a branch instruction (BRN) to the function or subroutine.

4.3 The prologue is produced by FORTRAN at the beginning of all function and subroutine segments. It is designed to operate with the calling sequence described in 4.2 and stores the link and the addresses of the arguments in working space. The prologue is as follows:-

1) For a subroutine segment of zero arguments:-

SBN (103)	1	1
STO (010)	1	WS0

2) For a subroutine or function of one argument:

STO (010)	1	WS0
OBEY (023)		0(1)
STO (010)	3	WS1

3) For a subroutine or function of two or more arguments:

LDCT (124)	2	Number of arguments
STO(010)	1	WS0
OBEY(023)		0(1)
STO(010)	3	WS1(2)
LDX(000)	1	WS0
ADN(101)	1	1
BUX(060)	2	*-5

4.4 The return from a function segment consists of:-

LDX(000)	6	Result
LDX(000)	7	Result+1 - Omitted if result is integer
LDX(000)	1	WS0
EXIT(072)	1	1

The return from a subroutine segment consists of:-

LDX(000)	1	WS0
EXIT(072)	1	1

5. Special Segments

5.1 Five special segments are provided by the FORTRAN system and are automatically included as required by FORTRAN programs. These are:-

FAPF	-	FORTRAN arithmetic package
FIOF	-	FORTRAN input/output package
FEXF	-	FORTRAN extension arithmetic package
FTAF	-	FORTRAN total array input/output auxillary
FMNF	-	FORTRAN monitor system

establishes a common area named "POOI" of 22 words as follows:-

ITEM	-	1 word
RES	-	10 words
K1	-	5 words
K2	-	5 words
Y	-	1 word

3.2 During consolidation (See FORTRAN Memo #3) the two cues will be combined into one storage area named "POOI" of 22 words (the largest of the sizes) of which the FORTRAN may use only the first 21 words. Thus I will correspond to ITEM, X and A together will correspond to RES and LIST will correspond to K1 and K2.

3.3 The FORTRAN statement PUBLIC is similar to COMMON except that it establishes an area containing only one variable or array and gives the area the same name as the variable or array.

3.4 The AUTOCODER directive #LOWER could be used in place of #UPPER and would ensure that the area was placed in lower storage although the FORTRAN segment(s) would not take advantage of this.

4. The Standard FORTRAN Linkage

4.1 The standard linkage may be divided into three parts.

1) The calling sequence - used in the calling segment to enter the called segment.
2) The prologue - used at the beginning of all function and subroutine segments.
3) The return - used by the called segment to return to the calling segment.

4.2 The calling sequence is produced by FORTRAN by the use of the CALL statement or by the use of a function name within an expression and is as follows.

```
CALL (070)     1     first location of called segment
LDN(100)       3     first argument
LDN(100)       3     second argument
    '
    '
    '
LDN(100)       3     last argument
```

If any of the arguments require indirect addressing, the appropriate LDN instruction is replaced by an LDX (000) instruction. The effect of each argument instruction must be to place in X3 the address of the argument variable.

If an argument is an array, the corresponding argument word must be an instruction which will place the address of the header in X3.

The linkage to these is not in the standard form described above.

5.2 FAPF provides floating point arithmetic and miscellaneous operations. Any of 19 operations are performed depending on the entry point which may be to any of the locations FAPF to FAPF+18 inclusive: If:-

X is a floating point number in X6, 7
I is an integer in X6
Y is a floating point number whose address is given in X3
J is an integer whose address is given in X3
N is an integer in X3
Z is a floating point number in L10,11

Then the following operations are performed.

Entry	Operation
0	X = X + Y
1	X = X * Y
2	X = X - Y
3	X = Y - X
4	X = -X
5	X = X/Y
6	X3 = address of array element (see below)
7	X = Y/X
8	I = I ** J
9	I = I ** N
10	X = X ** J
11	X = X ** N
12	I = J ** I
13	I = N ** I
14	X = Y ** I
15	X = I
16	Z = J, X Unchanged
17	Z = N, X Unchanged
18	I = X

In all cases entry is made by the instruction CALL(070) 1 FAPF+n and except for entry 6 return is made to the following instruction.

Entry 6 is entered with the address of an array header in X3 and with successive words after the CALL(070) instruction as instructions to place the addresses of the subscripts in X3. Return is made to the instruction following the argument list with the address of the desired element in X3.

Entry 1 has the additional property that if X3 contains the integer 6 (i.e. X=Y) then the square of X is calculated.

5.3 FIOF provides input output operations. Nine different entries are provided as follows:

FIOF	Runout	Apply only to paper tape punch
FIOF+2	Endfile	
FIOF+14	Release	
FIOF+16	Disengage	

In the above 4 cases the word following the call must identify the peripheral - The last six bits = peripheral type and unit in the same form as a peripheral cue. Return is made to the instruction following the peripheral identity.

FIOF+4	Initiate write
FIOF+6	Initiate read
FIOF+8	Integer variable
FIOF+10	Real variable
FIOF+12	Terminate

These 5 cases provide data input/output. Entry 4 or 6 must first be used with the address of the format data in X3 and the peripheral identity in the word following the call. Return is made to the instruction following the peripheral identity.

Subsequent entries 8 and 10 mixed in any sequence may now be used each supplying, in X3, the address of a variable. Return is made to the instruction following the call. X3 is preserved by these entries.

Finally entry 12 must be used to terminate the operation.

The format data consists of a string of characters similar to those in a FORMAT statement starting with the character following the first left parenthesis to and including the final right parenthesis.

5.4 FEXF provides two additional floating point operations as follows:

Entry	FEXF	X=X**Y
	FEXF+1	X=Y**X

X and Y are as described in 5.2

5.5 FTAF may be used between the initial and final entries to FIOF to read or write an entire array. entry is made to FTAF with the address of the array header in X3. Return is to the instruction following the CALL instruction.

5.6 FMNF provides extended monitoring facilities. Further detail will be provided at a later date.

FP6000 COMPUTER SYSTEM

FORTRAN

Ferranti ELECTRONICS
A DIVISION OF FERRANTI-PACKARD ELECTRIC LIMITED
Industry Street, Toronto 15, Ontario, Canada

ACKNOWLEDGEMENT

Mr. Daniel D. McCracken's book "A Guide to Fortran Programming" contains sections which are reproduced in this manual in whole or in part. Mr. McCracken's book was published in 1960 by John Wiley and Sons, Inc. of New York and Ferranti-Packard Electric Limited is grateful for the permission of both the author and the publisher to use their copyright material.

"A Guide to Fortran Programming" is recommended as a worthwhile investment for anyone connected with the writing of Fortran programs and particularly as a supplement to this manual, since it is not oriented to any specific computer but contains valuable information on the differences in the various Fortran "dialects".

TABLE OF CONTENTS

Table of Contents (continued)

CHAPTER 1

CONSTANTS, VARIABLES & EXPRESSIONS

1.1 What is FORTRAN?

For an electronic computer to be effective in the solution of an engineering or scientific problem, the problem-solving procedure must be presented to the computer in a language that it can "understand". The computer's basic language consists of elementary instructions, such as add, subtract or shift a number. The problem-solving procedure must therefore be translated into simple instructions that the computer is capable of obeying. This translation, which is called programming or coding, can be carried out entirely by a human being, or the computer may assist in the process by use of a compiler.

A compiler is a set of computer instructions which can accept a problem-solving procedure, written in a form resembling the language of the procedure, and produce from it the proper elementary machine instructions that will solve the problem. A compiler that is designed primarily for use in the solution of engineering and scientific problems is generally spoken of as an algebraic compiler. FORTRAN, a shortened form of FORmula TRANslation, is such a compiler. The user of an algebraic compiler is not required to know the detailed operation of the computer. He is therefore able to concentrate on the problems to be solved rather than on how the computer operates.

1.2 A FORTRAN Program

A procedure to be followed in solving a problem with FORTRAN is specified by a series of statements. These statements are of several types. One type specifies the arithmetic operations that are the heart of the procedure. A second calls for input or output, such as reading a data tape, printing a line of results, or punching a card of results. A third specifies the flow of control though the set of statements, that is, the sequence in which the statements are to be executed. The fourth consists of statements that provide certain information about the procedure without themselves causing any action.

Taken together, all of the statements that specify the manner in which a problem is to be solved constitute the "source program". When a source program has been written and transferred to the input medium appropriate to the computer installation, it is translated by FORTRAN into an "object program". This is a set of elementary instructions that the computer can understand. The object program is then ready for execution by the computer.

1.3 Real and Integer Numbers

There are two different number representation systems, or modes, used in FORTRAN: Real and Integer. The two modes are not interchangeable because numbers in different modes are stored and processed within the computer in entirely different ways.

An "Integer number" is any integer in the range

$$-8,388,608 \text{ to } 8,388,607$$

A "Real number" is any number in the range

$$-10^{77} \text{ to } 10^{77}$$

and is represented internally with a precision of eleven significant digits. Numbers in the range

$$-10^{-77} \text{ to } 10^{-77}$$

are considered, and represented as, zero.

Note that an "Integer number" is always an integer, whereas a "Real number" may be an integer or have a fractional part. Furthermore, FORTRAN carries out computations with Real numbers in such a way that we do not have to be concerned with the location of decimal points. All questions of lining up decimal points before addition and subtraction are automatically taken care of by the computer. For this reason Real numbers are sometimes termed "floating-point" numbers.

1.4 Constants

Any number that appears in literal explicit form in a statement is called a constant, whereas a quantity that is given a name is called a variable. For instance, we shall see a little later that the following are FORTRAN statements:

```
I = - 2

X = A + 12.7
```

Here 2 and 12.7 are constants; I, X and A are variables.

An "Integer constant" is an unsigned Integer number, assumed positive and written without a decimal point.

A "Real constant" is an unsigned Real number, assumed positive and written in one of the two following ways:

(a) With a decimal point, or

(b) With or without a decimal point, and followed by an exponent Ey where "y" is a one or two digit integer, assumed positive if written without a sign.

The following are acceptable Integer constants:

```
0
5
123
8388607
```

The following are not acceptable Integer constants:

1.2	(no decimal allowed)
1,124	(no comma allowed)
8388608	(too large)
53E + 5	(no exponent allowed)

The following are acceptable Real constants:

0.0	
6.	
.9	
12300.	
0.0012	
12.345	
5.0E + 2	(5.0×10^2)
3.E - 2	(3×10^{-2})
19E5	(19×10^5)
.713E23	($.713 \times 10^{23}$)

The following are not acceptable Real constants:

6	(no decimal point)
1,234.	(no comma allowed)
2E77	(2×10^{77} - too large)
4.5 + 2	(E missing)

Note that the characteristic decimal point may appear at the beginning of the number, at the end, or between two digits. A Real constant may be written with any number of digits, but only eleven digits of significance will be retained by the computer.

1.5 Variables

The term "variable" is used in FORTRAN to denote any quantity that is referred to by name rather than by explicit appearance. A variable name, which may be freely invented by the programmer, consists of from one to forty alphanumeric characters (ie. letters or digits) of which the first must be a letter.

The first letter of the name of an Integer variable must be I, J, K, L, M or N. Examples of acceptable names of Integer variables are:

```
I
KLM
MATRIX
L1123
I6M2K
```

Examples of unacceptable names of Integer variables are:

ABC	(incorrect first letter)
5M	(begins with a number)
J34.5	(contains a non-alphanumeric character)

Generally, most computation will be done in Real mode because of the convenience provided by the automatic handling of all decimal points. The first letter of a name of a Real variable is any letter other than I, J, K, L, M or N. Examples of acceptable names of Real variables are:

```
AVAR
R51TX
G
SVECT
```

1.5 continued

Examples of unacceptable names of Real variables are:

8CAR	(begins with a number)
KJL1	(incorrect first letter)
B9/35	(contains a non-alphanumeric character)

Care must be taken to observe the rule for distinguishing between Integer and Real variables. If the rule is violated, FORTRAN will in some cases reject the program and in other cases give results that are not what the programmer intended. It should be noted that the compiler places no significance on names; it merely inspects the first letter to determine whether the variable is Real or Integer. A name such as B7 specifically does not mean B times 7, B to the seventh power, or B_7. If the programmer chooses to assign names that simplify recall of the meaning of the variable, this is perfectly acceptable, but no such meaning is attached to the symbols by the compiler. It should also be noted that every combination of letters and digits constitutes a separate name. Thus the name ABC is not the same as the name BAC, and the names A, AB and AB7 are all distinct. However, blanks are not considered in variable names. Therefore, ABC and A B C will be considered to be the same name.

1.6 Operations and Expressions

FORTRAN provides for five basic operations: addition, subtraction, multiplication, division, and exponentiation. Each of these operations is represented by a distinct symbol:

Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	**

Note that the combination ** is considered to be one symbol; there is no confusion between ** and *, since, as we shall see, it is never permissible to write two operation symbols side by side. These are the only operations allowed: any other mathematical operations must either be built up from the basic five or computed by using the functions discussed later.

The term "expression" is used in its precise mathematical sense in FORTRAN. An expression is defined as a constant, variable, or function, or any mathematically meaningful combination of these separated by operation symbols, commas, and parentheses. Some examples of expressions and their meanings are shown in Table 1.1 below.

TABLE 1.1

Expression:	Meaning:
K	The value of the Integer variable K
3.14159	The value of the Real constant 3.14159
A + 2.1828	The sum of the value of A and 2.1828
RHO - SIGMA	The difference between the values of RHO and SIGMA
X*Y	The product of the values of X and Y
OMEGA/6.2832	The value of OMEGA divided by 6.2832
C**2	The value of C raised to the second power
(A+F)/(X+2.)	The sum of the values of A and F divided by the sum of the value of X and 2.
1./(X**2+Y**3)	The reciprocal of $(X^2 + Y^3)$

In writing expressions, the programmer must observe certain rules in order to convey his intentions correctly.

1. Two operation symbols must not appear next to each other. Thus A* - B is not a valid expression but A*(-B) is.

2. Parentheses should be used to indicate groupings just as in ordinary mathematical notation. Thus $(X+Y)^3$ should be written (X+Y)**3 to convey the correct meaning. Again, A-B+C and A-(B+C) are both legitimate expressions but they do not mean the same thing.

1.6 continued

3. The ambiguous expression A^{B^C} should be written as A**(B**C) or as (A**B)**C, whichever is intended. It should not be written as A**B**C.

4. When the hierarchy of operations in an expression is not completely specified by the use of parentheses, all exponentiations are performed first, then all multiplications and divisions, and finally all additions and subtractions. Thus, the following two expressions are equivalent.

(a) A*B+C/D-E**F

(b) (A*B) + (C/D) - (E**F).

As another example, X*Y**3 means $X . Y^3$ not $(X. Y)^3$. Note that this rule applies only in the absence of parentheses. Thus, the expression (X*Y)**3 means $(X. Y)^3$, since Rule 2 takes precedence.

5. Within a sequence of consecutive multiplications and/or divisions, or additions and/or subtractions, in which the order of the operations to be performed is not completely specified by the use of parentheses, the operations are performed from left to right. Thus the expression A/B*C would be taken to mean $\frac{A}{B} . C$, not $\frac{A}{B. C}$,

and I - J + K would mean (I-J)+K, not I-(J+K).

6. Although any expression may be raised to a power that is a positive or negative Integer quantity, only Real expressions may be raised to a Real power. An exponent may itself be any expression Thus the expression X**(I+2) is perfectly acceptable.

7. Integer and Real quantities must not be "mixed" in the same expression; however, Integer quantities may appear in Real expressions as exponents and as subscripts. Subscripts will be discussed in Chapter 3.

8. Parentheses indicate grouping with the exception of their application to subscripted variables and functions (see Chapters 3 and 4). Specifically they never imply multiplication. Thus, the expression (A+B) (C+D) is incorrect and should be written (A+B)*(C+D).

TABLE 1.2

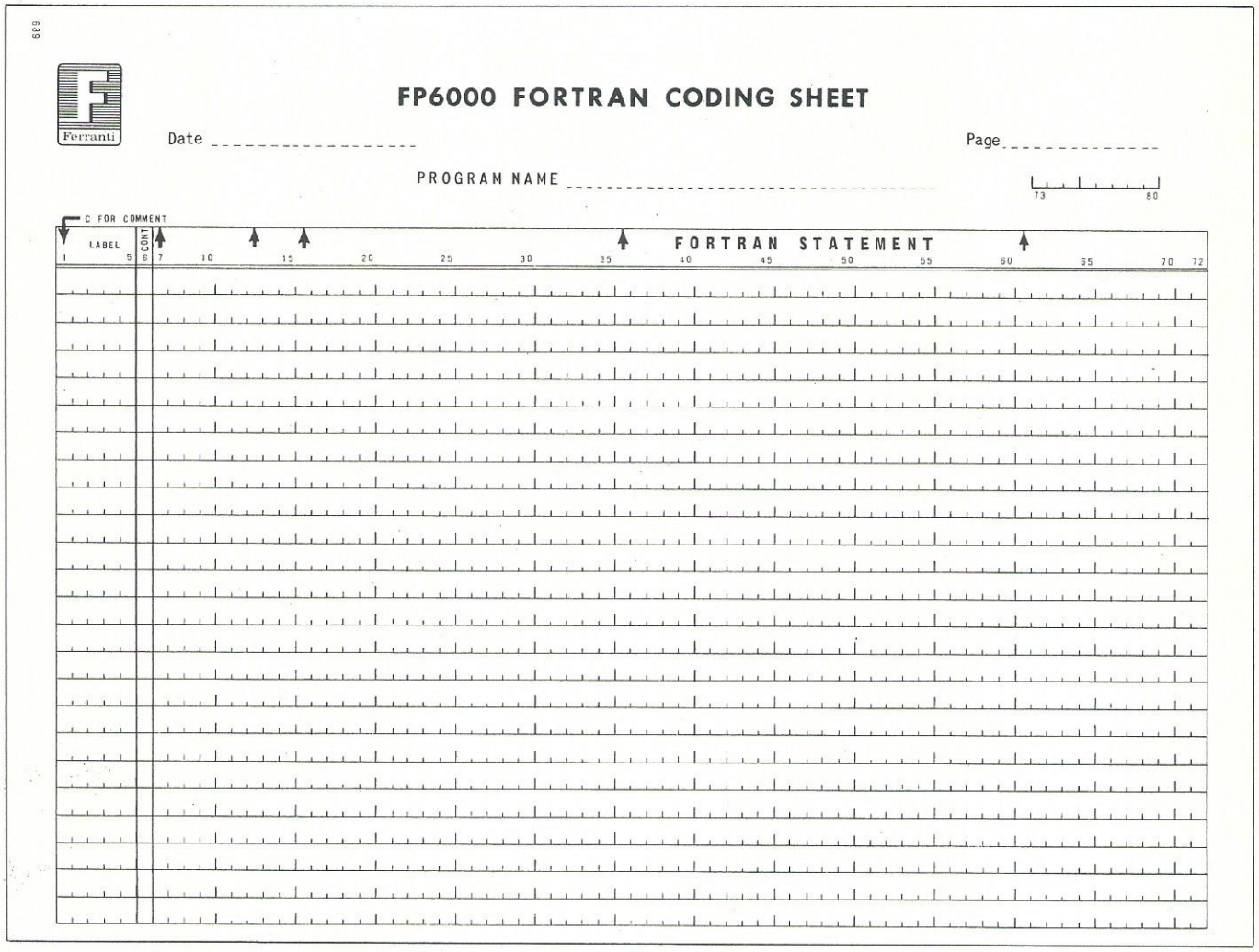
Mathematical Notation	Correct Expression	Incorrect Expression
$A.B$	A*B	AB (no operation symbol)
$A. (-B)$	A*(-B) or -A*B	A*-B (two operation symbols side by side)
$A + 2$	A + 2.	A + 2 (mixed modes)
$-(A + B)$	-(A + B)	- A + B or - + A + B
A^{I+2}	A**(I+2)	A**I + 2 ($= A^I + 2$, mixed)
$A^{B+2}.C$	A**(B+2.)*C	A**B + 2.*C ($= A^B + 2.C$)
$\frac{A.B}{C.D}$	A*B/(C*D) or A/C*B/D	A*B/C*D $\left(= \frac{ABD}{C}\right)$
$\left(\frac{A+B}{C}\right)^{2.5}$	((A + B)/C)**2.5	(A + B)/C**2.5 $\left(= \frac{A+B}{C^{2.5}}\right)$
$A\left[X + B(X + C)\right]$	A*(X + B*(X + C))	A(X + B(X + C))

CHAPTER 2

ARITHMETIC STATEMENTS AND STANDARD FUNCTIONS

2.1 Writing FORTRAN Statements

A FORTRAN program, as we have noted, is made up of a series of statements, each of which is either an instruction to carry out some operation or a source of information to the compiler about the program. The program will be coded on the form shown in Figure 2.1 below.



2.1 continued

Positions 1 - 5 contain the statement number, if any. The use of statement numbers is described in Chapter 7. Position 1 may also be used to indicate a comment line. If position 1 contains a "C", then the FORTRAN compiler recognizes that the line is a comment line, and does not process it. However, the line will be printed on any listing made of the program. Comment lines may be used freely to give information about the program to anyone who may have to read it. Liberal use of comments can be valuable to the original programmer, as well, if the program has to be modified after a long period. FORTRAN programs are a great deal easier to read than programs written in actual machine instructions, but it can still be difficult to understand the purpose of a complex program if there are no comments.

If a statement is written on one line, position 6 must be left blank. If more than one line is required to write a statement, position 6 must be blank or zero in the first line and contain some non-zero, non-blank character in subsequent lines. The statement itself is written in columns 7 to 72. Blanks are ignored by FORTRAN and may be used freely to improve readability. The statement need not begin in column 7; some programmers, for instance, indent the continuations of a long statement to make it a little clearer that continuation is involved. Some like to leave a space on both sides of each operation system for readability. All such conventions are at the discretion of the programmer. Positions 73 to 80 are not processed by FORTRAN and may be used for any desired line or program identification.

It is essential that the coding forms be filled out with great care and attention to detail. The statements must always be written in exactly the format specified; if a comma is misplaced or omitted the program may not be compiled or may be compiled incorrectly. It is strongly recommended that only capital letters be used and that great care be taken to write certain easily confused characters in a distinctive manner. Various conventions are available for distinguishing between characters such as the letter "O" and the digit "zero". One acceptable way to write these characters is shown in Table 2.1. Greek letters are not permitted.

TABLE 2.1

The digit one;	1	The letter I;	I
The digit zero;	0	The letter O;	Ø
The digit two;	2	The letter Z;	Z
The digit five;	5	The letter S;	S
The letter V;	V	The letter U;	U

2.2 Arithmetic Statements

The most common statement is the arithmetic statement, which is an instruction to FORTRAN to perform a computation. Its general format is

a = b

in which "a" is a list of one or more variables of the same mode, written with or without subscripts and separated by commas. Each variable will be set equal to the expression "b", which is any expression as defined in Section 1.6. As will be stated later, the mode of "a" need not be the same as that of "b". The equals sign in an arithmetic statement is not used in the same way as it is in ordinary mathematical notation. For example, it is not permissible to write statements such as A+B=C+D, in which A is unknown and the others are known. The precise meaning of the equals sign is then: replace the value of the variable(s) named on the left by the value of the expression on the right. Thus the statement A=B+C is an instruction to form the sum of the values of the variables B and C and to replace the value of the variable A with that sum. The previous value of the variable A is lost. The statement GAMMA = 1.67 is an instruction to replace the value of the variable GAMMA by 1.67.

Another example of an arithmetic statement brings out very forcefully the special meaning of the equals sign. A statement such as N = N + 1 has the meaning: replace the value of the variable N by its old value plus 1. This sort of statement, which is clearly not an equation, finds frequent use.

TABLE 2.2

Arithmetic Statement:	Original Formula:
R = (A + B*X)/(C + D*X)	$R = \frac{A + BX}{C + DX}$
BETA = -1./(2.*X) + A**2/(4.*X**2)	$\beta = \frac{-1}{2X} + \frac{A^2}{4X^2}$
FY = X*(X**2 - Y**2)/(X**2 + Y**2)	$Fy = X. \frac{X^2 - Y^2}{X^2 + Y^2}$
C = 1.112*AK*R1*R2/(R1 - R2)	$C = 1.112K \frac{r_1 r_2}{r_1 - r_2}$
Y = (1E-6 + A*X**3)**1.5	$Y = (10^{-6} + AX^3)^{3/2}$
A = 4*K - 6*K1*K2	$A = 4K - 6k_1k_2$
I = I + 1	$I_{new} = I_{old} + 1$
K = 12	$K = 12$
PI = 3.1415927	$\pi = 3.1415927$
M = 2*M + 10*J	$M_{new} = 2M_{old} + 10J$
ALPHA, BETA = 0	$\alpha = \beta = 0$

The examples of Table 2.2 above show acceptable arithmetic statements with their equivalent normal mathematical forms. Variable names have been chosen arbitrarily; any other legitimate names would have been equally valid. It is also assumed, of course, that previous statements have established values of the variables on the right-hand sides.

The examples in Table 2.3 below are presented to emphasize once again the importance of writing expressions and statements in the prescribed format. All of the examples in Table 2.3 contain at least one error.

TABLE 2.3

Incorrect Statement:	Error:
Y = 2.X + A	* missing
3.14 = X - A	Left side must be a variable name
A = ((X + Y)A**2 + (R - S)**2/16.7	* missing Not the same number of left and right parentheses;
X = 1,624,009.*DELTA	Commas not allowed in constants
-J = I**2.	Variable on left must not be written with a sign; Integer quantities may not be raised to Real powers.
BX6 = 1./ -2.*A**6	Two operation symbols side-by-side not permitted, even though the minus is not intended as indicating subtraction.
DERIV = N*X**(N - 1)	"Mixing" Integer and Real this way not permitted.
A,I = X + Y	"Mixing" Integer and Real this way not permitted.

2.3 FORTRAN Arithmetic

Most arithmetic operations should be programmed in Real mode. This will avoid decimal-point location problems and allow convenient handling of fractions. However, cases will arise occasionally in which it is necessary to use Integer mode. Since pitfalls await the unwary, it is necessary to describe precisely how Integer arithmetic is done.

The fundamental consideration is that if an Integer division gives a result that is not an integer the result is truncated to a whole number rather than rounded; that is, any fractional part is simply discarded. For instance, the result of the Integer division 5/3 is 1, not 2. This applies only to division. Additions, subtractions, and multiplications on Integer quantities give correct integral results.

Expressions that involve sequences of operations which include division require caution in the grouping of the operations. This is the only occasion in which Rule 5 of Section 1.6 is of practical importance. For instance, if we carry out the calculation 5/3*6, the result will be 6, not 10 or 12. The rule states that in a sequence of multiplications and divisions in which no parentheses appear the calculations are performed from left to right. Thus 5 would be divided by 3 and the result truncated to 1, which would be multiplied by 6 to give the result of 6. On the other hand, if the expression is written as 5*6/3, 6*5/3, 5*(6/3), or (6/3)*5, the result will be 10. This somewhat bothersome fact is ordinarily only an academic consideration because most programs will operate on Real numbers, in which case the problem does not arise. The calculation in the example would give a Real result of 10.0 regardless of the way in which the expression is written (but note the next paragraph).

Even with Real mode arithmetic, there is a precaution to be observed. Because of rounding errors and the fact that most fractions have only approximate decimal representations, results that should be integers may not come out as integers. For instance, suppose we were to perform the calculation 1./3.*3., which, we recall, means (1./3.)*3. The result clearly should be 1.0, but it will not be. The result of the division, to eleven decimal places, is 0.33333333333. When this is multiplied by 3, we get 0.99999999999, not 1.0. The unsuspecting programmer may be understandably surprised by such a result. This difficulty is unavoidable in the use of any digital computer. One must simply be aware that such things can happen and take appropriate steps to handle properly any situation in which problems could arise.

In most cases the variables in an arithmetic statement will be either all Integer or all Real, with a few exceptions as noted. However, it is permissible to have an Integer variable on the left of an arithmetic statement with a Real expression on the right and vice versa. If the variable on the left is Integer and the expression on the right is Real, then the entire computation will be done in Real mode and the final result truncated to an integer before being stored as the new value of the variable on the left. If the variable on the left is Real and the expression is Integer, then the entire computation will be done in Integer mode, with truncation of any nonintegral results after each operation, and the final result converted to Real form. These situations will have occasional usefulness.

A final word of caution. Although the allowable range of numbers will be entirely adequate for most problems, there is the possibility that a result will exceed the limits. If the result of a computation exceeds these limits, overflow will be set. The programmer may test for overflow with the IF OVERFLOW statement (see Chapter 7) and act accordingly.

2.4 Standard Functions

FORTRAN provides for the use of certain common mathematical functions. Every function has a pre-assigned name. Some of these functions and their names are shown in Table 2.4.

TABLE 2.4

Mathematical Function	FORTRAN Name
Square Root	SQRTF
Sine of an angle in radians	SINF
Cosine of an angle in radians	COSF
Arctangent; angle given in radians	ATANF
Natural logarithm	ALOGF
Exponential	EXPF
Absolute Value	ABSF

The names of all standard functions end in "F". As will be seen later, other types of functions may be defined by the programmer, and their names must not conflict with those of standard functions. It is suggested, therefore, that their names do not end in "F".

In order to make use of a mathematical function, it is only necessary to write the name of the function and to follow it with an expression enclosed in parentheses. As an example of the use of functions, suppose it is necessary to compute the cosine of an angle named X. This angle must be expressed in radians. Writing COSF(X) in a statement will result in the computation of the cosine of the angle. In this example the argument of the function is the single variable X. This is by no means a necessity; the argument may be any expression, with the one restriction that in all of the mathematical functions mentioned above the argument must be a Real quantity and the function value is computed in Real mode. If, for example, we wanted the square root of $B^2 - 4AC$, we would simply write

```
SQRTF (B**2 - 4.*A*C)
```

A function may be written in a statement wherever the function value is desired. If, for instance, we are using the square root function in computing one of the two roots of a quadratic equation, we could write a statement

```
ROOT1 = (-B+SQRTF(B**2 - 4.*A*C))/(2.*A)
```

It may be well to review the purpose of the parentheses here. Those enclosing B**2 - 4.*A*C are required to enclose the argument of the square root function. The parentheses around the numerator in the formula indicate that everything before the slash is to be divided by what follows. The parentheses enclosing the 2.*A indicate that the A is in the denominator; without this final set, the result would be to divide the numerator by 2 and then multiply the entire fraction by A.

It is perfectly permissible for the argument of one function to involve another function. This is illustrated by a statement which we may write to carry out the following computation, which arises in the calculation of a certain integral:

$$VAL = \frac{1}{\cos X} + \log\left|\tan\frac{X}{2}\right|$$

Since we have no function for taking the tangent of an angle, we will compute the tangent from the formula:

$$\tan x = \frac{\sin x}{\cos x}$$

The statement to compute this value could be as follows:

```
VAL = 1./COSF(X) + ALOGF(ABSF (SINF (X/2.)
                    /COSF (X/2.)))
```

That part of the statement following the plus sign calls for the computation of the logarithm of the absolute value of the sine over the cosine. All of the parentheses here are essential; in each case they enclose the argument of a function.

This statement gives the same result as the following set of statements, where the statements would be executed in sequence.

```
Y = X/2.
TAN = SINF(Y)/COSF(Y)
AB = ABSF(TAN)
VAL = 1./COSF(X) + ALOGF(AB)
```

Now, each function has an argument consisting of only one variable. This may always be done if desired to reduce the complexity of a statement. In fact, such simplifying substitutions are often a good idea even in computations not involving functions, just to keep the statements simple and easy to work with.

CHAPTER 3

SUBSCRIPTED VARIABLES

3.1 Definitions

Subscripted variables permit us to represent many quantities with one variable name. A particular quantity is indicated by writing a subscript (or subscripts) in parentheses following the variable name. The complete set of quantities is called an "array", and the individual quantities are called "elements". An array may have one or more subscripts, where the number of subscripts represents the dimensionality of the array.

The first element of a one-dimensional array is number 1, the second is number 2, etc., up to the number of elements in the array. In mathematical notation we might write X_1, X_2, X_3, ..., X_{19}, X_{20}; in FORTRAN subscript notation we would write X(1), X(2), X(3),...., X(19), X(20). When used in this connection, "one-dimensional" refers to the number of subscripts, not to the number of elements.

A two-dimensional array may be thought of as being composed of horizontal rows and vertical columns. The first of the two subscripts then refers to the row number, running from 1 up to the number of rows, and the second to the column number, running from 1 up to the number of columns. For instance, an array of two rows and three columns might be shown in mathematical notation as

$$\begin{matrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \end{matrix}$$

In FORTRAN subscript notation the elements would be written A(1,1), A(2,1), A(1,2), A(2,2), A(1,3), A(2,3). Note that the subscripts are separated by commas.

The naming of an array is the same as for non-subscripted variables. In particular, an array may consist of either Integer or Real elements, as defined by the first character of the name. The elements of any one array are, however, all Integer or all Real.

3.2 Examples of the Subscript Notation

Suppose we have two points in space, represented in co-ordinate form by X_1, X_2, X_3 and Y_1, Y_2, Y_3. We are required to compute the distance between them, which is given by

$$D = \sqrt{(X_1 - Y_1)^2 + (X_2 - Y_2)^2 + (X_3 - Y_3)^2}$$

Now suppose that we have set up an array called X, the three elements of which are the Real co-ordinates of the point X, and another similarly for Y. The computation of the distance between the points can be called for by the statement shown in Figure 3.1.

3.3 Motivations For The Use Of Subscripted Variables

The foregoing example shows the fundamental idea of the subscript notation, but it does not really indicate the power of the technique.

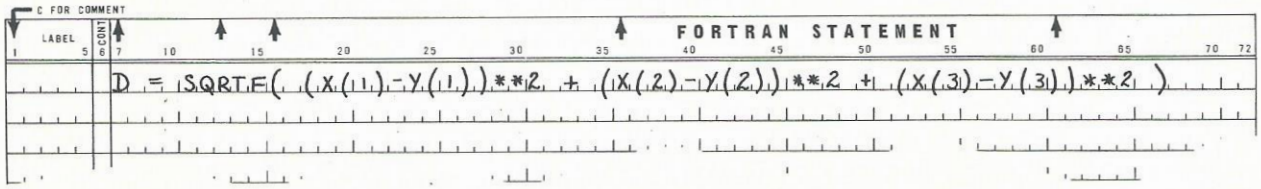


Figure 3-1

3.3 continued

After all, there is nothing in the example that could not be done just as conveniently by giving each variable a separate name. Why then are subscripted variables such an important feature of FORTRAN?

The reason is that the subscripts may themselves be Integer variables or Integer expressions. This means that a program can be set up to perform a basic computation and then make the same computation on many different values simply by changing the value of the subscript.

Suppose, for instance, that it is required to compute the sum of the squares of 20 numbers, X_1 to X_{20}, that are stored in the computer. They could, of course, be given 20 different names and a long arithmetic statement written to compute the sum of their squares, but this would be tedious, cumbersome, and inflexible. Instead the 20 numbers could be considered to be the elements of a one-dimensional array called X. Now, any of the 20 can be referenced by the name X(I) where I takes on the appropriate value from 1 to 20.

The usual mathematical notation for this operation is

$$SUMSQ = \sum_{i=1}^{20} X_i^2$$

The sum could be computed as follows:

(1) Set the result field to zero and "i" equal to 1

(2) Add $(X_i)^2$ to the result field

(3) Add 1 to "i"

(4) Test whether "i" $<$ 21. If so, go back to step (2). When "i" becomes equal to 21, the answer has been formed in the result field and the routine could continue to perform further computations.

This could be coded as shown in Figure 3.2 below. The effect of the IF statement, which will be discussed in Chapter 7, is to cause the program to loop through the add routine twenty times and then to continue with the rest of the program.

```
      SUMSQ = 0
      I = 1
123   SUMSQ = SUMSQ + X(I)*X(I)
      I = I + 1
      IF (I - 21) 123,9
```

Figure 3-2

3.4 The DIMENSION Statement

When arrays are used in a program, certain information about them must be supplied to the FORTRAN compiler. This indicates:

(1) which names refer to arrays,

(2) how many subscripts there are for each array, and

(3) the maximum size of each subscript.

This information may be supplied by the DIMENSION statement, although there are other means, as will be seen later, for supplying this information. The dimensions of each array must be given prior to its use in the program. A common practice is to give the dimension information for all arrays in a DIMENSION statement at the start of the program. One DIMENSION statement may mention any number of arrays, and there may be any number of DIMENSION statements.

The DIMENSION statement is of the form

DIMENSION a_1, a_2, a_3, ...

where each a_i stands for an array name followed by parentheses enclosing one or more unsigned Integer constants which give the maximum size of each subscript. When FORTRAN processes a DIMENSION statement, it sets aside enough storage locations to contain arrays of the sizes defined by the information in the statement. Thus, if a program contains the statement

DIMENSION X(20), A(3,10), K(2,2,5)

FORTRAN will assign 20 locations to the one-dimensional array named X; 30 (3 x 10) to the two-dimensional array, A; and 20 (2 x 2 x 5) to the three-dimensional array, K.

It is the programmer's responsibility to write the program so that no subscript is ever larger than the maximum size specified in the DIMENSION statement. Furthermore, subscripts must never be smaller than 1; zero and negative subscripts are not permitted. If these restrictions are violated, the source program will be compiled, but the object program will in all probability give incorrect results.

The DIMENSION statement is said to be "non-executable", that is, it only provides information to the FORTRAN processor and does not result in the creation of any instructions in the object program. It may therefore appear anywhere in the source program, even between two arithmetic statements. As previously noted, however, the dimension information for each array must be given before the first use of that array.

3.5 Storage of Arrays

The elements of an array are stored in successive storage locations.

The elements of a one-dimensional array are stored in sequence starting with the element corresponding to the subscript 1 and proceeding to the largest subscript as defined by the DIMENSION statement.

The elements of a two-dimensional array are stored in such a manner that the first subscript varies more rapidly. Thus the elements of array R,

DIMENSION R(2,3)

are stored in the sequence R(1,1), R(2,1), R(1,2), R(2,2), R(1,3), R(2,3). This can be summarized by saying that the elements of a two-dimensional array are stored in column order.

In general, the elements of arrays are stored in such a manner that the first subscript varies most rapidly and successive subscripts vary less rapidly.

The order of storage of array elements are of concern only when considering the READ and WRITE statements (see Chapter 9), which can transmit entire arrays, and the COMMON, PUBLIC and EQUIVALENCE statements (see Chapter 6) which can overlay the storage of arrays.

CHAPTER 4

ARITHMETIC STATEMENT FUNCTIONS

4.1 Introduction

The various functions of the system greatly increase the power and flexibility of FORTRAN. We have so far discussed only those functions supplied with the system, which require no effort from the programmer beyond writing the correct name where the function value is desired and entering the desired expression for the argument. The names of these functions are all established in advance, and the programmer must write them exactly as specified.

4.2 Arithmetic Statement Functions

It often happens that a programmer will find some relatively simple computation recurring throughout his program, making it desirable to be able to set up a function to carry out the computation. This function would be needed only in the one program, so that there would be no point to setting up a new, standard function for the purpose. Instead, a function can be defined within the one program and then used in it wherever desired.

An arithmetic statement function is defined by writing a statement of the form:

a (argl, arg2, ---, arg n) = b

where "a" is the name of the function and "b" is an expression. The name, which is invented by the programmer, must conform to the same rules as variable names. It must not be the same as the name of any supplied function or of any variable used in the program. The name of the function is followed by parentheses enclosing the arguments, which are separated by commas if there is more than one. The arguments in the definition are variables which must not be subscripted. The mode of the function will depend, as for variables, on the first letter of the name.

The right-hand side of the definition statement may be any expression. Any variables which are included in the expression and are not arguments are the parameters of the function; they are just ordinary variables. The expression may use other functions. Each arithmetic statement function definition must appear before the function is used.

This is only the definition of the function; it does not cause computation to take place. The variable names used as arguments are only dummies; they may be the same as variable names appearing elsewhere in the program. The argument names in the definition are unimportant, except as they specify Real or Integer mode.

An arithmetic statement function is used by writing its name wherever the function value is desired and adding the appropriate expressions for arguments. The values of these expressions will be substituted into the routine established by the definition and the value of the function computed. These actual arguments may be subscripted if desired.

As an illustration, suppose that in a certain program it is frequently necessary to compute one of the roots of the quadratic equation $ax^2 + bx + c = 0$, given values of a, b, and c. An arithmetic statement function can be defined to carry out this computation by writing

```
ROOT (A,B,C) = (SQRTF (B*B-4. *A*C)-B)/(2. *A)
```

The foregoing equation defines the function; the compiler will produce a sequence of instructions in the object program to compute the value of the function, given three values to use in the computation.

Suppose now that it is desired to use this function, with 16.9 for a, R-S for b, and T+6.9 for c; the value of the function ROOT is to be added to the cosine of X and the sum stored as the new value of ANS. All this can be done with the statement.

```
ANS = ROOT (16.9,R-S,T+6.9) + COSF (X)
```

Suppose that later in the program it is necessary to compute the function with DATA(I) for a, DATA (I + 1) for b, and 0.087 for c; the function value is to be cubed and stored as the value of TEMP:

```
TEMP = ROOT (DATA(I), DATA(I+1), 0.087)**3
```

4.2 continued

It must be emphasized that the variables A, B, and C in the function definition have no relation to any variables of the same name elsewhere in the program. To illustrate, suppose that the value of the root is needed for an equation

$$22.97X^2 + AX + B = 0$$

where A and B are simply variables in the program. The root may be found by writing

```
VAL = ROOT (22.97, A,B)
```

The A and B that appear here in the use of the function are completely unrelated to the A and B in the definition of the function. In summary, the variables in the definition are simply dummies that establish how the expression values in the use of a function should be substituted into the object program routine set up from the definition.

CHAPTER 5

PROGRAM SEGMENTS

5.1 Reasons for Segmenting a Program

A program is developed in a number of stages. First the problem to be solved must be generally defined, taking into consideration the form of input and output and general method of solution. Next, flow charts showing the general sequence of operations should be drawn up, with detailed charts made for any more complicated portions of the program. Following the preparation of flow charts the actual writing, or coding, of the program will be done.

If the problem is large, it may be desirable to divide it into segments, treating each segment as an almost separate entity. The program may be divided among various programmers either to facilitate early development, or to allow those who are most familiar with a particular phase of the program to do the developing of that part. Another reason for building a program in parts is to allow separate testing of each part. Then, if one portion of the program is not correctly written it is only necessary to rewrite, compile and test that segment on its own. The development of the other segments can go on independently. A further reason for segmented programs is that often a routine may be used in a number of different programs. This routine may be written as a separate entity and incorporated into any program which requires it.

It may be noted here that variables are defined separately within each segment and thus have no relation to any other segment, (see, however, the effect of the COMMON and PUBLIC statements in Chapter 6). Also, arithmetic statement functions apply only to the segment in which they are defined. Thus a name may be used in one segment to refer to a variable, and the same name used in another segment to refer to a completely different and independent variable, array or arithmetic statement function.

5.2 The MASTER, FUNCTION, SUBROUTINE and END Statements

Each program must have one (and only one) basic, MASTER segment. This segment will be introduced by a MASTER statement, written in the form

MASTER name

where "name" is the name of the total program, and is subject to the same rules as variable names, although no distinction between Real and Integer modes is applicable. The first four characters of the name, or the full name if fewer than four characters are involved, are used as the program name during the running of the compiled program.

This name will be used by the console operator in communicating with the program during its execution.

All other segments of the program will be FUNCTION or SUBROUTINE segments, each of which must be introduced by either a FUNCTION or a SUBROUTINE statement of the form

FUNCTION name (arg 1, arg 2, ...)

SUBROUTINE name (arg 1, arg 2, ...)

where "name" is the name of the program segment, and is subject to the same rules as variable names. The distinction between Real and Integer mode is made in the case of a FUNCTION segment, but does not apply to a SUBROUTINE segment. The purpose of the statements' arguments will be covered in the following sections.

Each segment of a program, including the MASTER segment, must terminate with the statement

END

This is physically the last statement of every segment and indicates to the compiler that it has reached the end of the segment.

5.3 The FUNCTION Segment

Useful as an arithmetic statement function often is, it does have three rather serious restrictions: the definition may only be used in the segment in which it is written, it is limited to one statement and it can compute only one value. The use of the FUNCTION segment removes the first two restrictions and of the SUBROUTINE segment removes all of them. Both types of segment may be written and compiled as entities separate from and without any reference to or interference with other segments, yet it is quite easy to set up communication between segments. Whether these segments are viewed as powerful extensions of the arithmetic statement function idea or as a way to divide a program, they are a most valuable part of the language.

5.3 continued

As with the arithmetic statement function, we must distinguish carefully between the definition and use. A FUNCTION segment is defined by writing a FUNCTION statement, followed by a series of statements forming a routine to carry out the function's computation, and terminating with an END statement. The arguments in the function definition are only dummy variables of the same mode as the actual arguments that will be processed by the routine.

The function's name must appear at least once in its routine as a variable on the left side of an arithmetic statement or in the list of an input statement.

Each FUNCTION or SUBROUTINE segment must contain at least one RETURN statement of the form

RETURN

This causes the segment to return control to the segment from which it was entered. In a FUNCTION segment, the value of the function is the value of the variable "name" at the time the RETURN statement in segment "name" is executed.

To make use of the FUNCTION segment, it is only necessary to write the name of the function with suitable expressions for arguments as part of an arithmetic statement. The mechanics of the operation of the object program are as follows. The FUNCTION segment is compiled as a set of machine instructions in one area of storage. Wherever the name of the segment appears in an executable statement, a transfer to the segment is set up in the object program. When the computations have been completed, a transfer is made back to the section of the program which caused transfer to the segment. This return is effected through the instructions generated by the RETURN statement.

As an example of the use of a FUNCTION segment let us consider the calculation of one of the roots of a quadratic equation. As shown in Section 4.2, coding to solve this problem may be written in the form of an arithmetic statement function

```
ROOT (A,B,C) = (SQRTF (B*B-4. *A*C)-B)/(2. *A)
```

However this method of calculating the root would necessitate writing the statement in each segment which required the root and, if more than one segment using this arithmetic statement function were combined into one program, space would be wasted. If it were written as a FUNCTION segment, it could be used by any other segment of the program and also by any program that required it.

The coding for this segment might then be:

```
FUNCTION ROOT (A,B,C)

ROOT = SQRTF(B*B - 4.*A*C) - B)/(2.*A)

RETURN

END
```

The segment would be called into use in exactly the same way (from a programming standpoint) as the arithmetic statement function.

It can be seen then that a FUNCTION routine is quite similar to an arithmetic statement function, except that it can comprise many statements instead of only one. Furthermore, it is not restricted to arithmetic statements.

5.4 The SUBROUTINE Segment

The SUBROUTINE segment is quite similar to the FUNCTION segment, with three differences:

(1) A SUBROUTINE segment may have many results instead of the one to which a FUNCTION segment is restricted.

(2) A SUBROUTINE segment cannot be used simply by writing its name in an expression. Instead, we write a CALL statement to specify the arguments and to bring it into action. The arguments may be used to transmit results produced in the segment, as well as to supply data to it.

(3) Since the results of a SUBROUTINE may be transmitted by the arguments and may be a combination of Integer and Real numbers, the first letter of the SUBROUTINE name is not used to designate mode.

5.4 continued

In all other respects the two segment types are entirely analogous. Let us consider again the problem of finding the roots of a quadratic equation. However, to illustrate multiple results, both roots will be calculated in the following example. The coding necessary to carry out the calculation is as follows:

```
SUBROUTINE ROOT (A,B,C,R1,R2)
RT = SQRTF (B*B - 4.*A*C)
DEN = 2.*A
R1 = (RT - B)/DEN
R2 = (-RT - B)/DEN
RETURN
END
```

This and similar preceding examples assume that the roots of the equation will not be complex.

5.5 The CALL Statement

The above routine would be entered by means of a CALL statement. This is a statement of the form

CALL name (arg 1, arg 2, ...)

where "name" is the name of a SUBROUTINE segment. If there are any arguments they must be enclosed in brackets, separated by commas and must agree in number, sequence, and mode with the arguments in the corresponding SUBROUTINE statement.

A CALL statement which could have been used to call the above SUBROUTINE example might have been

CALL ROOT (R,S,T,X,Y)

The effect of this statement is to set variables X and Y equal to the roots of the equation

$$Rx^2 + Sx + T = 0$$

5.6 Dummy Arrays

The name of an array, written without subscripts, may be used as an argument. In this case the entire array is supplied to the FUNCTION or SUBROUTINE segment. Dimensions of such arrays are automatically provided and need not be specified in the segment. However, the name of the array must appear in a DIMENSION statement within the segment. Only the array name, without subscripts appended, need be given in the DIMENSION statement, (eg., DIMENSION A,B where A and B are dummy arrays). If any dimensions are given they will be ignored. The use of the DIMENSION statement is necessary to indicate to the compiler that the name(s) concerned applies to an array.

5.7 Dummy Functions and the EXTERNAL Statement

The name of a FUNCTION or SUBROUTINE segment or standard function, written without arguments, may be used as an argument. Thus if in one case a particular FUNCTION segment will require a sine routine and in another case, perhaps at a different point in the program, a cosine routine is required to carry out calculations, the name of the desired routine may be included as an argument of the function. The FUNCTION routine may then examine this argument to determine whether it will calculate a sine or cosine at a particular stage in its computations.

If a FUNCTION, SUBROUTINE or standard function name appears in a segment, without arguments, as an argument of a function or of a CALL statement, then it must have previously appeared in the same segment either as a function or a CALLed SUBROUTINE segment, or else in an EXTERNAL statement. This is necessary so that the compiler may recognize the names as those of functions rather than variables.

The EXTERNAL statement is written in the form

EXTERNAL n_1, n_2, n_3, ...

where n_1, n_2, n_3, ... is a list of function names.

5.7 continued

To clarify the use of this statement, consider the following. If a SUBROUTINE segment has been created and is required is some instances to calculate the sine of a variable and in other cases the cosine of the variable, there must be some means of informing the routine which function it must use. This may be done by naming the function required as an argument in the CALL statement. The SUBROUTINE statement must name a dummy function as one of its arguments, and may then use the actual function which is given as the corresponding argument of a CALL statement. For example:

```
SUBROUTINE JOE (FUNCT,A,B,C)

A = FUNCT (B/C)

RETURN

END
```

Whatever function is named as the first argument in the segment's CALL statement will operate on the result of (B/C). Therefore the statements

```
EXTERNAL SINF
CALL JOE (SINF, X, Y, Z)
```

would cause X to be set equal to the sine of Y/Z. The EXTERNAL statement in the above example could have been omitted if SINF had been used previously in the segment as a function.

An EXTERNAL statement is not required for the following statement:

```
CALL JACK (SINF(A), X, Y, Z)
```

Here SINF(A) is an expression, whereas in the previous example, SINF was a function required by the SUBROUTINE segment JOE.

5.8 Summary of Functions

FORTRAN allows the use of four types of functions: standard functions supplied with the system, arithmetic statement functions, FUNCTION segments and SUBROUTINE segments.

The differences in these function types are summarized in Table 5.1 below.

The arguments appearing in the definition may be the names of variables, arrays or functions (although only the names of variables may be used as arithmetic statement function arguments). Arguments appearing in the use of functions may be any expression, array name or function name, except that arguments used to transmit results from SUBROUTINE segments may only be variables or array names. In the corresponding sequences of arguments (use and definition), expressions must correspond to dummy variable names, array names to dummy array names, and function names to dummy function names.

TABLE 5.1

Type:	Name:	How Defined:	How Used:	Number of Arguments:	Number of Results:
standard	Defined by system	Defined by system	By use in an expression	1 or more	1
a.s.f.	1 - 40 alphanumeric characters	Defined by a single arithmetic type statement within a segment	By use in an expression	1 or more	1
FUNCTION	1 - 40 alphanumeric characters	Established by a FUNCTION segment	By use in an expression	1 or more	1
SUBROUTINE	1 - 40 alphanumeric characters	Established by a SUBROUTINE segment	By use in a CALL statement	0 or more	0 or more

CHAPTER 6

COMMON, PUBLIC AND EQUIVALENCE STATEMENTS

6.1 Introduction

The statements COMMON, PUBLIC and EQUIVALENCE have been provided as a means of conserving storage and of communication between program segments. In a large program it is often desirable to be as economical as possible with storage space. Also due to the fact that programs are frequently written in segments, it is desirable to have a means of relating areas and names in one segment to those of another.

6.2 The COMMON Statement

The COMMON statement is written in the form

COMMON list

where "list" is a list of one or more sections each indicating the name of a storage area and the names of the elements which make up that area. Thus "list" is of the form

/BLOCK NAME 1/a,b, ..., c /BLOCK NAME 2/d,e, ..., f /BLOCK NAME 3/...etc.

where /BLOCK NAME 1/ a,b,...,c is one section and a,b,...,c are the elements of that section. An element may be a variable or an array name.

Storage is assigned to the elements in a section in the order in which they appear, within an area assigned to the block name. Although any number of COMMON statements are allowed within a segment, any one block name may occur only once in a segment.

The block name of a section, comprising from one to forty alphanumeric characters of which the first is alphabetic, is preserved for the whole program. The block name may be "blank" in which case the section starts with two consecutive slashes (//). These slashes may be entirely omitted if the section is the first in a COMMON statement.

When the compiler encounters a block name which has been previously mentioned in a COMMON statement of a different program segment, space will be allotted to its elements in the same area previously assigned to that block name. Elements of blocks of the same name must agree as follows. Elements in corresponding positions in the sequence of elements following the block name must be of the same mode, either both arrays or both variables, and if arrays, of the same size. However, these corresponding elements need not have the same name.

No variable or array in a COMMON statement may have previously occurred in any statement other than DIMENSION, INTEGER or REAL (for INTEGER and REAL, see Chapter 10).

Consider as an example the following:

```
COMMON   /AREA/X,Y,I     (in one segment)

COMMON   /AREA/A,B,J     (in another segment)
```

As a result of these statements, X and A will be assigned the same storage location, as will Y and B, and I and J. Note that variables assigned the same storage location must be of the same mode. Therefore, the two statements could not have been written

```
        COMMON        /AREA/X,Y,I

and     COMMON        /AREA/A,J,B
```

as Y and J would have been illegally matched (as also would I and B).

6.3 The PUBLIC Statement

The PUBLIC statement takes the form

PUBLIC V_1, V_2, V_3, ...

where V_i is the name of a variable or array. The PUBLIC statement enables variables of the same name in different segments to occupy the same storage locations. If a variable is to be public to two routines, it must appear in a PUBLIC statement in both routines.

The purpose of the PUBLIC statement is the same as that of the COMMON statement. However, the way in which the areas are equated (PUBLIC by name and COMMON by order) may make one more desirable than another in particular situations.

6.4 The EQUIVALENCE Statement

The EQUIVALENCE statement takes the form

EQUIVALENCE (list), (list), (list)...

where "list" is a list of variables and array names separated by commas. The variables may not be subscripted.

The EQUIVALENCE statement causes two or more variables to be assigned the same storage location, which is useful in two rather different ways.

In one usage the EQUIVALENCE statement allows the programmer to define two or more variable names as meaning the same thing. It might be that after writing a long program the programmer will realize that he has inadvertently changed variable names and that X1 and RST6 should refer to the same variable. Rather than going back and changing the variable names in the program, a time-consuming and error-prone process, he can write

EQUIVALENCE (X1,RST6)

and the mistake is corrected.

The other application is in making use of the same storage location to contain two or more variables that are never needed at the same time.

These two applications of EQUIVALENCE differ only in viewpoint; the statement and its treatment by the compiler are the same in either case.

One EQUIVALENCE statement can establish equivalence between any number of sets of variables. For instance, if A and B are to be made equivalent, as are X and Y, we can write

EQUIVALENCE (A,B), (X,Y)

All names in a list must refer to arrays or variables of the same mode. No more than one name in the list may have previously occurred in a statement other than DIMENSION, INTEGER or REAL (for INTEGER AND REAL, see Chapter 10). If no names have so occurred, there is no restriction to the mixture of variables and arrays in the list. If one name has so occurred, and refers to:

(1) An array, no other name may be that of an array with a larger number of elements.

(2) A variable, all other names must be those of variables.

CHAPTER 7

TRANSFER OF CONTROL

7.1 Statement Numbers

In most of the examples presented so far it has been assumed that statements are executed sequentially. In many cases, however, we may wish to execute statements in some order other than the one-after-the-other sequence in which they are ordinarily taken. When this is desired, it is necessary to have some means by which a particular statement may be identified. This identification is provided by permitting each statement to have a "statement number" (or "label").

A statement number, which is written in positions 1 to 5 of the statement line, may be any positive number from 1 to 99999. No two statements in the same segment may have the same statement number, but there is no requirement that every statement be labelled. Furthermore, there is no sequencing implied by the statement numbers. The first numbered statement need not bear the number 1, and statement numbers need not be assigned in an unbroken sequence or even in an ascending sequence. To emphasize this point: an acceptable sequence of statement numbers is 500, 7, 9936, 9935, 9937, 200, and 259. Interspersed between the statements having these numbers could be statements having no statement numbers. In short, statement numbers provide the means by which one statement can refer to another within the same segment.

7.2 The GO TO Statements

The GO TO statements provide a means of transferring control to some statement other than the next one in sequence. There are two different GO TO statements:

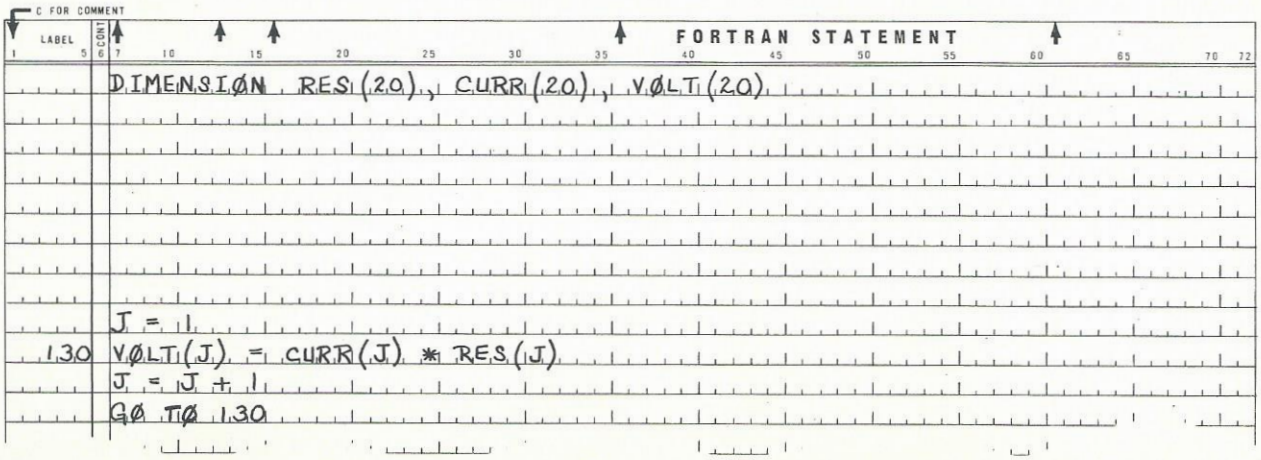
(1) GO TO n

and (2) GO TO $(n_1, ---, n_m)$,i

The latter will be referred to as the "Computed" GO TO statement and is described in Section 7.4. In the former statement, "n" is an integer which is interpreted as a label. Program control is transferred to the statement labelled "n". This statement may be any executable statement in the same program segment.

As an example of the use of the GO TO statement, suppose that it is required to calculate the voltage drop across a rheostat at various resistances and with various currents flowing through it. Three tables, each large enough to contain 20 values, will be represented as arrays: RES_J (resistance), $CURR_J$ (current), and $VOLT_J$ (voltage). Values in array RES_J represent various settings of the rheostat and values in the corresponding positions of $CURR_J$ represent the current flowing through it at each setting. The results would be stored in array $VOLT_J$. A set of statements to compute these results could be as shown in Figure 7.1 below.

A sequence of statements executed repeatedly as are the three beginning with statement 130 are often referred to as a "loop". At the end of twenty loops, the resultant voltages would be in array $VOLT_J$. Note however, that this routine provides no means of stopping after twenty. One way that the Loop could be terminated at that time is by use of an IF statement. This statement will be discussed in the following section.



7.3 The IF Statements

The GO TO statement provides a way to alter the sequence of statement execution unconditionally. Besides this, however, it is often necessary to change the sequence of statement execution on the basis of what happens during execution of the program. In other words, it is necessary to have a way of making a conditional transfer of control based on data or computed results.

The two IF statements provide this capability. They are of the form

(1) IF (b) n_1, n_2, n_3
and (2) IF OVERFLOW n_1, n_2

where "b" may be any expression of either mode. The "successors" n_1, n_2 and n_3 are either:

(1) A number which appears elsewhere as the label of an executable statement,
or (2) Blank or zero.

An IF statement of the first form will cause program control to be transferred to n_1 if the expression is negative, to n_2 if zero, or to n_3 if positive. An IF statement of the second form will cause program control to be transferred to the statement labelled n_1 if the overflow register is set (see the last paragraph of Section 2.3), or to the statement labelled n_2 if the overflow register is clear. The register is left clear after the execution of an IF OVERFLOW statement. If, in either statement, a successor is blank or zero, it is taken to refer to the next sequential executable statement. The commas separating successors must always appear even though the successor may be blank. For example

```
IF(A-B),,112
```

would transfer control to statement 112 if (A-B) was positive. Otherwise the next sequential statement would be executed.

As an illustration of these two statements, consider the example in Section 7.2 showing the use of the GO TO statement. The routine could be expanded to test for overflow and to test for the end of twenty loops as shown in Figure 7.2.

Statement number 131 would be the beginning of a routine handling an overflow condition. The second IF statement would loop the program through the multiplication routine twenty times and then allow the program to continue with further processing.

```
C FOR COMMENT
LABEL  CONT  FORTRAN STATEMENT
1      5     6  7    10   15   20   25   30   35   40   45   50   55   60   65   70  72
      DIMENSION RES(20), CURR(20), VOLT(20)
      J = 1
  130 VOLT(J) = CURR(J) * RES(J)
      J = J + 1
      IF OVERFLOW 131,
      IF (121 - J),,130
```

FIGURE 7.2

7.4 The Computed GO TO Statement

The computed GO TO statement is useful when it is desired to transfer control to one of a number of statements, depending upon the present value of some Integer variable. The statement has the general form

GO TO $(n_1, n_2, ---, n_m)$, i

In this statement "i" must be an Integer variable and the successors $n_1, n_2, ---, n_m$ must be statement numbers. If the value of "i" is "t" (where $1 \leqslant t \leqslant m$ and "m" is the number of successors) then control is transferred to the statement number written as successor n_t. If, for a particular value, transfer to the next sequential statement is desired, the successor should be blank or zero. For example, consider the statement

GO TO (4, 600, ,13),IVAR

If the value of the variable IVAR is 1, then control will go to statement 4; if IVAR is 2, then control will go to statement 600; if IVAR is 3, then the next statement in sequence will be selected, and if IVAR is 4, control will go to statement 13. If the value of IVAR were greater than 4, the results could not be predicted.

As an example of one kind of calculation that can be done with the computed GO TO statement, consider another problem. It is required, at a certain point in a program, to compute one of the first five Legendre polynomials according to the value of an integer variable LEG. These are defined as follows:

If LEG = 0, $P_0(X) = 1$

= 1, $P_1(X) = X$

= 2, $P_2(X) = 3/2\ X^2 - 1/2$

= 3, $P_3(X) = 5/2\ X^3 - 3/2\ X$

= 4, $P_4(X) = 35/8\ X^4 - 15/4\ X^2 + 3/8$

Assume that X has been previously computed and that it has also been determined which of these five functions of X is to be computed, that is, that the value of LEG has been established. The computed GO TO statement can not be used directly because of the restriction that the value of the Integer variable must not be less than 1. Therefore, 1 shall be added to LEG first to make it fall in the range of 1 to 5 instead of 0 to 4. A program for carrying out this computation is shown in Figure 7.3. P will contain the value of whichever Legendre polynomial is computed.

C FOR COMMENT

LABEL	CONT	FORTRAN STATEMENT
		LEG1 = LEG + 1
		GO TO (,62,,63,,64,,65), LEG1
		P = 1.0
		GO TO 70
62		P = X
		GO TO 70
63		P = 1.5 * X ** 2 - .5
		GO TO 70
64		P = 2.5 * X ** 3 - 1.5 * X
		GO TO 70
65		P = 4.375 * X ** 4 - 3.75 * X ** 2 + .375
70		

FIGURE 7.3

CHAPTER 8

THE DO STATEMENT

8.1 Introduction

One of FORTRAN's most powerful statements is the DO statement. This statement makes it possible to carry out a section of a program repeatedly, with changes in the value of an Integer variable between repetitions. Coupled with subscripted variables, the DO statement provides a simple way to make calculations that are quite complicated when done with actual machine instructions. In view of the importance of this topic, its application shall be shown in several examples.

The DO statement may be written in either of the forms:

DO n i = m_1, m_2

or DO n i = m_1, m_2, m_3

where "n" is an integer interpreted as the label of the last statement to be obeyed in the loop. "i" is a non-subscripted Integer variable which may not be a dummy variable nor appear in a COMMON, PUBLIC or EQUIVALENCE statement. Each of m_1, m_2 and m_3 is either an unsigned integer or a non-subscripted Integer variable and m_3 must always be positive. If m_3 is not stated it is taken to be 1. The statements immediately following the DO statement up to and including the statement labelled "n" are executed repeatedly, with "i" having the values m_1, $m_1 + m_3$, $m_1 + 2m_3$, ---etc. The last execution of the loop occurs when "i" has the greatest value less than or equal to m_2. The DO loop is executed at least once every time it is entered.

The effect of the DO statement is identical to that of the statement

$$I = M_1$$

written in place of the DO statement together with the statements

$$I = I + M_3$$

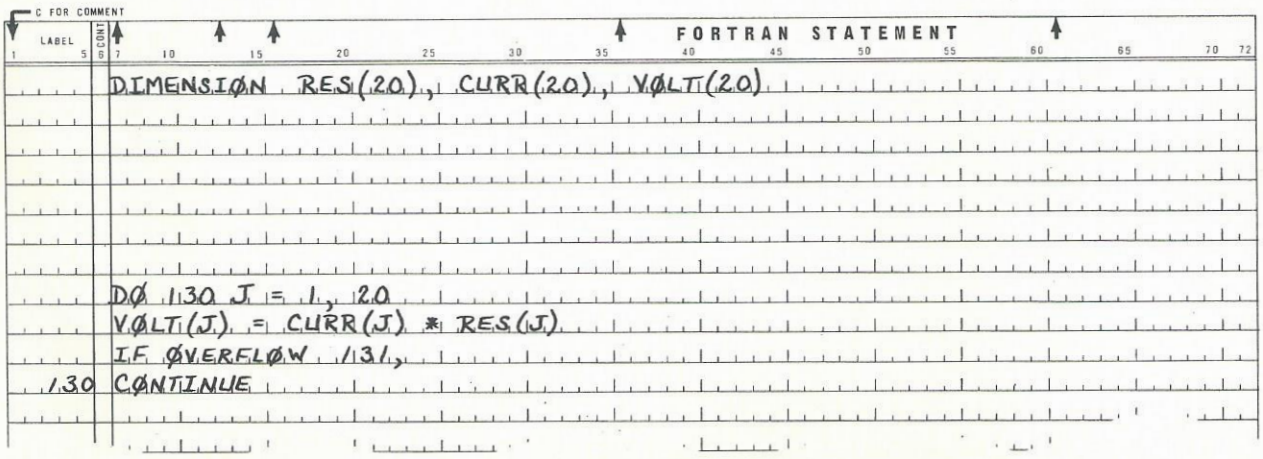
IF (I - M_2) RETURN,RETURN,

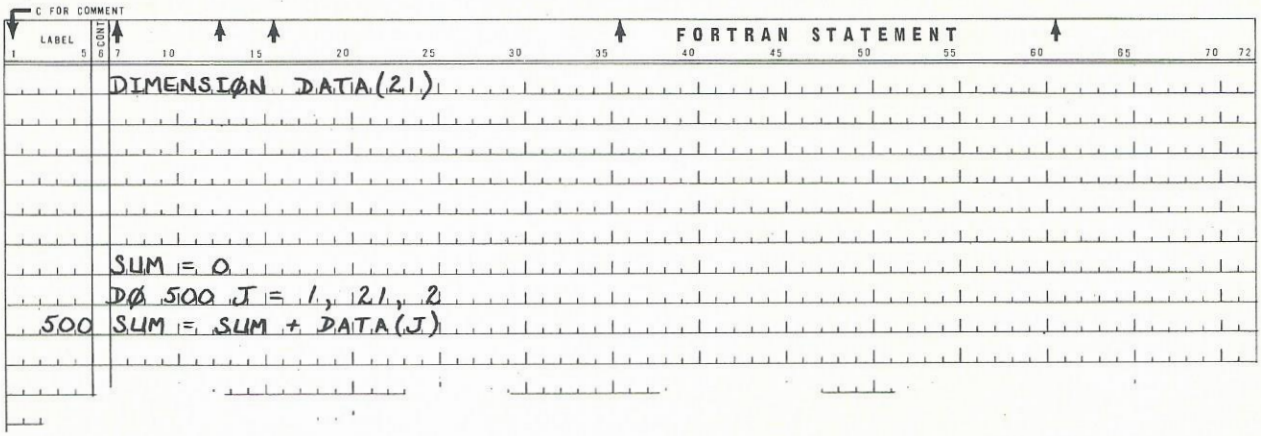
inserted after the statement labelled "n" where RETURN implies a return to the statement following $I = M_1$. Note that the IF statement would not return once 'I' became greater than M_2, as the third successor has been left blank.

To illustrate how the DO statement works and how it may be used, consider again the example used in connection with the IF and GO TO statements. A simpler way of writing this routine would be as shown in Figure 8.1 below.

As before, the loop would be executed twenty times before continuing with the next statements. If overflow occurred, transfer would be made to an overflow routine starting with the statement labelled 131. The CONTINUE statement, described in Section 8.4, is used because the IF OVERFLOW statement must not be the last of a DO loop.

For another example of a DO statement, suppose that a one-dimensional array called DATA contains 21 elements. The program of Figure 8.2 will form the sum of all of the odd-numbered elements and place this sum in a location called SUM.





8.1 continued

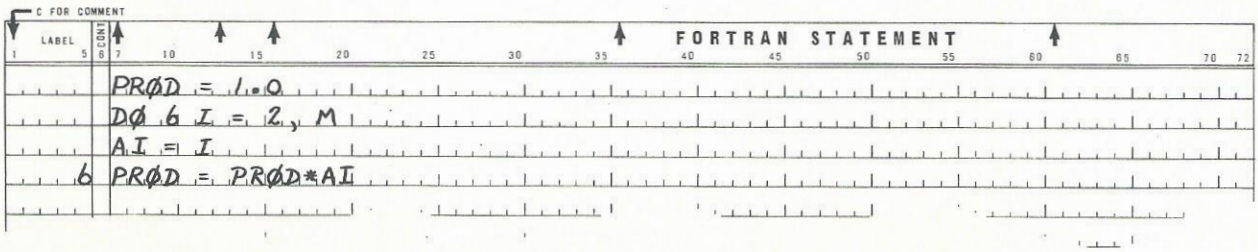
The DO statement, in this case, causes the statement numbered 500 to be executed with J equal to 1, 3, 5, ..., 17, 19, 21. The first time statement 500 is executed the value of the variable SUM is zero, so that the net effect is to move the first element of DATA to SUM. The second time it is executed the net effect is to add to this the value of the third element of DATA, etc. When it has been executed with J equal to 21, which adds the last element, then control passes to the statement following statement 500.

8.2 Further Definitions

Before proceeding to more examples, a few definitions will be given to clarify the use of the DO statement. The "range" of the DO statement is the set of repeatedly executed statements. In short, it consists of the statements beginning with the one immediately following the DO and continuing up to and including the statement named in the DO. The Integer variable "i" in the general form of the DO statement is called its index. Throughout the execution of the range, "i" is available for any purpose permitted for an Integer variable, providing its value is not altered.

There are two general ways in which control is transferred outside the range of a DO. The normal exit occurs when the DO is satisfied, that is, at the completion of the number of executions of the range as specified by the indexing parameters m_1, m_2, and m_3. When this happens, control passes to the statement following the one named in the DO. The second way in which control is transferred outside the range of a DO is through the use of statements such as GO TO or IF statements within the range of the DO. When control passes out of the DO loop, whether as a normal exit after the index exceeds "m_2" or before the loop is satisfied, the index "i" is available for any purpose permitted for an Integer variable.

To examine another way in which the index of a DO may be used within the range, consider the problem of forming the product of all integers from 1 to M, wherein the value of M has been determined earlier in the program. Note that the variable M is an Integer variable; it is desired however, to obtain the product in a Real form. In order to avoid the complications of computing too large an Integer number within the DO loop, each of the factors in the product must be converted from Integer to Real mode. The statements of Figure 8.3 below do all this and place the product in a location called PROD.



8.2 continued

PROD is set equal to 1.0 before entering the DO loop, which is executed with the index I equal to all values from 2 to M. In order to use the index I in a Real mode calculation, the statement AI = I is included as the first of the DO loop. This statement causes the value of I to be stored in AI. However, since AI is a Real variable, the value will be stored in Real form. The first time statement 6 is executed, the effect is to multiply 1.0 by 2.0 (since PROD has been started at 1.0) and to store the product in PROD. The next time, this product is multiplied by 3.0 and the new product is stored back in PROD. The process continues until the two statements in the range have been executed with I equal to M. Implicit in this program is the assumption that M is at least 2.

It is important to be sure that the range of a DO is executed exactly the right number of times. Experience shows that it is all too easy to make mistakes on this point. A good way to check is to ask, "What would the parameters have to be if the range were to be executed only once?" Based on this, it is usually not too difficult to decide if the actual situation is properly handled.

8.3 Rules Governing the Use of the DO Statement

A great deal of flexibility is permitted in the use of the DO statement as long as certain rules are observed. These are:

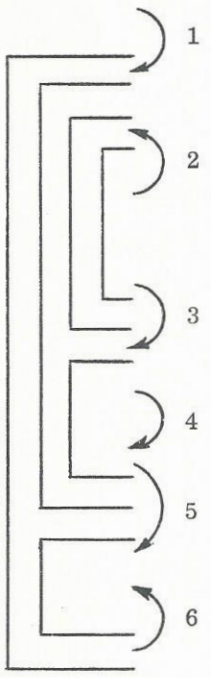
(1) It is permissible for the range of one DO (which may be called the "outer" DO) to contain another DO (which may be called the "inner" DO). This is termed "nesting". When this occurs, it is required that all statements in the range of the inner DO also be in the range of the outer DO. This does not prohibit the ranges of two or more DO's ending with the same statement, but it does prohibit a situation in which the range of an inner DO extends past the end of the range of an outer DO.

(2) The last statement in the range of a DO must not be one that can cause a transfer of control. This excludes the GO TO, the computed GO TO, the IF, the IF OVERFLOW, and the DO statements, which may, however, be used freely anywhere else in the range. (The CONTINUE statement described in Section 8.4 is provided for situations that would otherwise violate this rule).

(3) No statement within the range of a DO may alter any of the indexing parameters of that DO; that is, it is not permitted within the range of a DO to change the values of i, m_1, m_2, or m_3. These parameters may be used in any way that does not alter their values.

(4) Control must not transfer into the range of a DO from any statement outside its range, with one exception. Thus it is expressly prohibited to use a GO TO or an IF statement to transfer into the range of a DO without first executing the DO itself.

The one exception to the rule prohibiting transfers into the range of a DO from outside its range is this; it is permissible to transfer control completely outside the range of a DO, perform a series of calculations which make no change in any of the indexing parameters of the DO, and then transfer back to its range. The rule does not prohibit a transfer from the range of an inner DO into the range of an outer DO. This transfer is permissible because, from the standpoint of the outer DO, the transfer is executed entirely within its range. Some illustrations of this rule are provided in Figure 8.4 below.



8.3 continued

The brackets here represent the ranges of DO's and the arrows represent transfers of control: Transfers 2, 3, and 4 are acceptable, since 2 and 3 are transfers from the range of an inner DO to the range of an outer DO, and 4 is a transfer entirely within the range of a single DO. Transfers 1, 5, and 6 all represent transfers into the range of a DO from outside its range.

8.4 The CONTINUE Statement

CONTINUE is a dummy statement that causes no action when the object program is executed. It enables Rule 2 of Section 8.3 to be satisfied. It has another use as well. If it is determined during a pass that all calculations necessary to that pass have been completed, it may be desirable to continue with the next pass. However, it may not be desired to execute the last statement in the loop, and a return to the DO statement would reinitiate the whole routine. Instead, the CONTINUE statement may be written as the last statement of the loop, and a branch made to it, causing a return for the next pass. An example of the use of the CONTINUE statement in this way appears in Section 8.5.

8.5 A Further Example of the Use of the DO Statement

Since the DO statement is so powerful and since it is so heavily used in most FORTRAN applications, we shall give an additional example of its use.

Suppose that the input to a program consists of a series of experimentally measured values. Each point in the experiment involves an X value and a Y value, corresponding to the abscissa and ordinate on a graph. The data points were gathered and entered into the computer in random order; that is, the first X value goes with the first Y value and the second X value goes with the second Y value, etc., but the first X value is not necessarily the smallest of all of the X values. For the purposes of calculations that are to be done later in the program, it is necessary to rearrange the data points in storage so that the first X value is the smallest and that the second X value is the next larger, and so on. In other words, the data points must be put into ascending sequence on the X values.

It is assumed that the X values as they were originally read (ie., in scrambled order) are the elements of an array named X and that there are 25 of them. The Y values are the elements of another array called Y which also contains 25 values.

A FORTRAN program to rearrange these data points into ascending sequence on the X values involves a nest of two DO loops. The development of the program will be shown, however, by displaying a simplified version of the inner loop before writing the entire program. This simplified loop will place the smallest X value in the first position of the X array. This can be done by the following process. First compare the first and second X values in the original array. If the first X is smaller than or equal to the second, leave them alone; if the first X is larger than the second, interchange these two values within the array. Having inspected the first and second elements and interchanged them if necessary, inspect the first and third elements and either leave them alone or interchange them if the first element is the larger. This "first" element may very well be one one that was originally in second position, but this does not matter. Similarly, compare the first and fourth, first and fifth, etc., until the element in the first position has been compared with all others, interchanging at each step if necessary. This process guarantees that the smallest X will end up in the first position of the X array. As for each X there corresponds a Y, the same interchange operations will be carried out on the Y array as were carried out on the X array, but without testing of the Y values.

In order to interchange two values from the array in storage, a three-step process is followed:

(1) Move the first value to a temporary storage location, TEMP.

(2) Move the second value to the location originally occupied by the first.

(3) Move the first value, which is in TEMP, to the location originally occupied by the second.

A routine to carry out all of this is shown in Figure 8.5. It is assumed that the data values have been read in by an earlier part of the program, and the statements that complete the rearrangement or use the data values are not shown.

8.5 continued

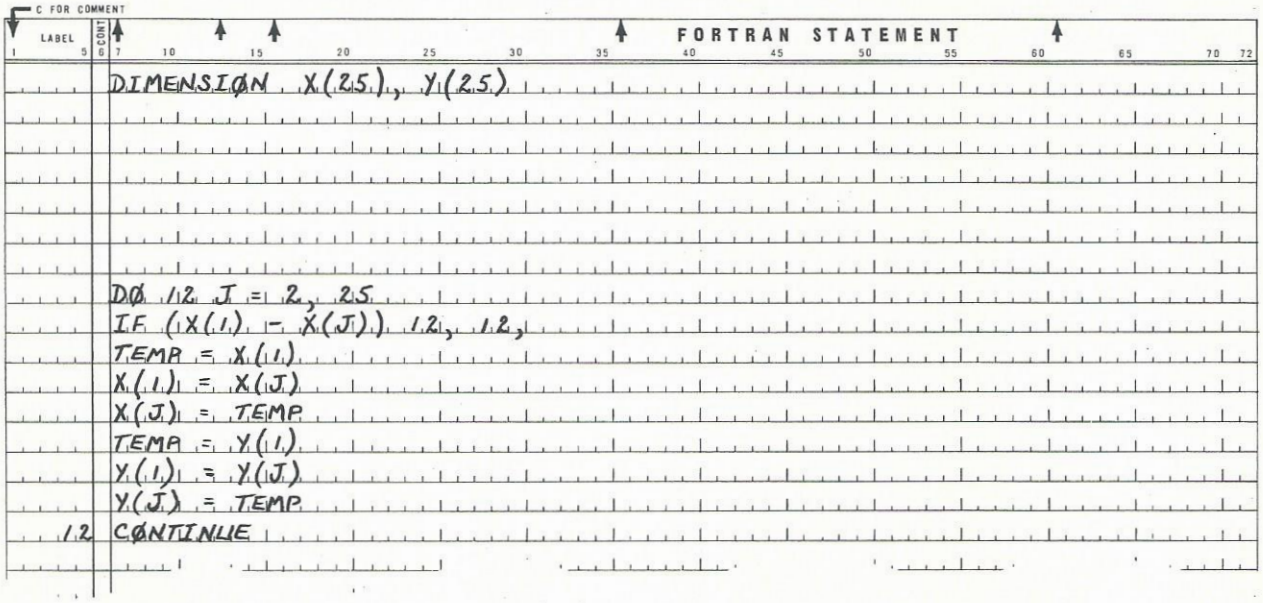


FIGURE 8.5

This illustration shows another example of a DO loop in which the index does not start with 1, and also shows an example of the use of the CONTINUE statement. The CONTINUE statement is required because if the IF statement shows that the first X value is already smaller than the other X value with which it is compared, then a transfer of control is required in order to skip around the six statements that interchange the X and Y values involved. The requirement in this case is to repeat the whole process again with the index J increased by 1. As has already been noted, however, a transfer back to the DO statement would result in starting the DO loop again with J equal to 2. Therefore, control is transferred to the CONTINUE statement which has been identified in the DO statement at the end of the range.

In reading this program, it is well to recall the meaning of an arithmetic statement; the value of the variable on the left side of the equals sign is replaced by the value of the expression on the right. Thus a statement such as

X(1) = X(J)

means that the number identified by the variable name X(J) is to be moved to the location for the number identified by the name X(1). The value in the location X(J) is unchanged.

In the problem as stated there may or may not be two equal X values. As the program has been written, however, it does not matter; if they are equal, then there is no point in exchanging them and control is transferred to the statement CONTINUE to repeat the loop.

When all loops have been finished (when the DO has been "satisfied"), the data points will have been rearranged so that the smallest X is in the first position in the X array. What is required next is to get the next larger X in the second position of the array. This can be done by comparing the second element with the third, and all following, and interchanging whenever necessary. After that the next largest X values would be arranged in the third, fourth, fifth positions, etc., until finally all of the X's had been placed in the X array in order of increasing size.

8.5 continued

It appears then that what is necessary for the complete program is to make variables of all subscripts that appear as 1's. The subscript will then select the element to be compared with all following elements (interchanging if necessary). This subscript, which shall be called I, will start at 1 and run through 24. The subscript which appears as J in the present program will still appear as J, but it will have to start at one more than whatever the I subscript is and run to 25. All of this is easily done by another DO statement that controls the I subscript. This is the outer DO. There is one complication, however. The inner DO must specify that the J subscript will start at one more than whatever I is. Looking back at the definition of the DO statement, it can be seen that each indexing parameter must either be an Integer constant or a single Integer variable. It is not permissible to write a statement such as

DO 12 J = I + 1,25

To avoid this restriction, a statement that computes the value of a variable IP1, which is always one more than whatever I is, may be inserted. The complete program to sort the data points is shown in Figure 8.6.

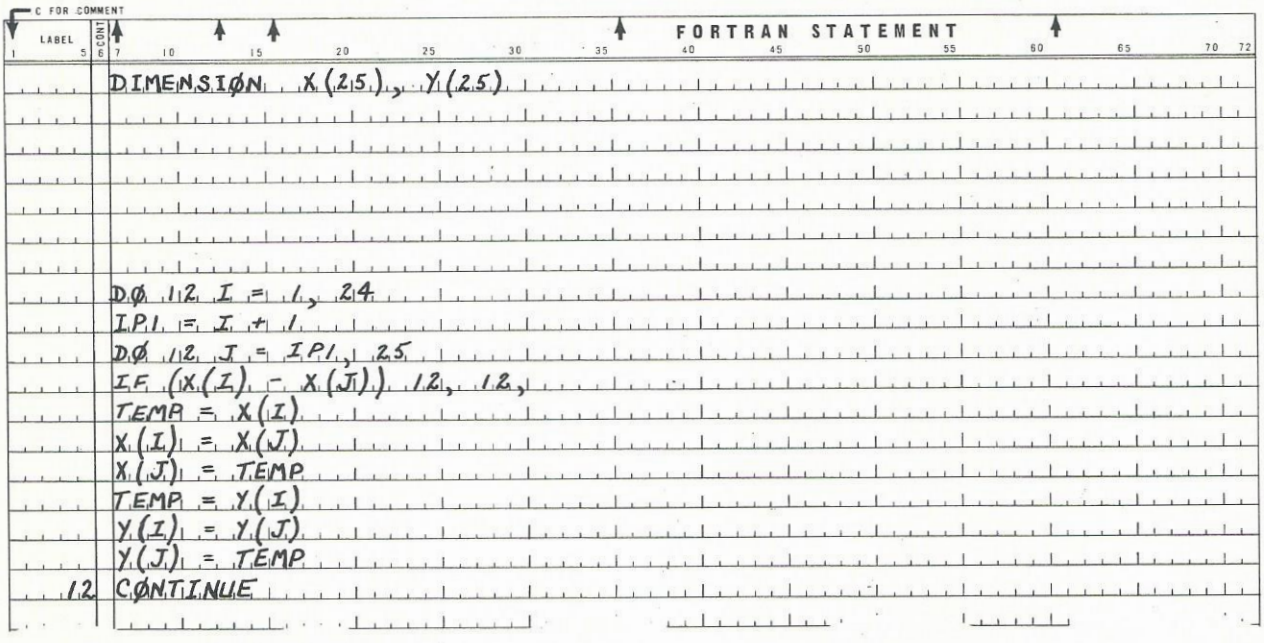


FIGURE 8.6

CHAPTER 9

INPUT-OUTPUT STATEMENTS

9.1 The PERIPHERAL Statement

If a problem is to be done only once, its data can be entered with the program in the form of constants in statements. This is ordinarily not the case, however; programs are usually set up to read in data at the time the program is executed. The same program can then be used to solve the same problem with as many sets of data as desired. In order that the compiler may generate read and write routines for a program, it is necessary that the programmer give the compiler information about the peripherals (input-output devices) which it requires. This is done by means of the PERIPHERAL statement.

This statement is written in the form

PERIPHERAL $(i_1, p_1n_1),(i_2, p_2n_2),\ldots$

where "i" is an unsigned, non-zero, Integer number, "p" is the alphabetic codeword (see below) for the type of device required, and "n" represents a particular device of type "p". Thus TR0 and TR2 would represent two different paper tape readers. "n" also has another meaning to the compiler. The object program will be assigned one more of a device than the largest "n" in a PERIPHERAL statement. If TR0 and TR2 were the only paper tape readers mentioned in a statement, then three paper tape readers would be assigned to the program. Therefore the integers "n" for each type of peripheral device used should belong to a dense set starting at zero (0,1,...).

The codewords "p" are as follows:

Paper Tape Reader	TR
Paper Tape Punch	TP
Line Printer	LP
Card Reader	CR
Card Punch	CP
Magnetic Tape Unit	MT
Peripheral Typewriter	TY

The PERIPHERAL statement is used to equate the programmer's reference number "i" to a specific device "pn". I/O (input-output) statements will refer to peripheral devices by means of the programmer's number "i". More than one "i" may be related to a particular "pn" device. However, the converse is not true.

By convention, the PERIPHERAL statement is written in the MASTER segment and incorporates all the peripheral devices required by the complete object program.

As an example, the statement

PERIPHERAL (3, TR0), (19, TR0), (4, TP0)
(5, TP1), (7, LP0)

would assign one paper tape reader, two paper tape punches and one line printer to the program.

9.2 The READ and WRITE Statements

In order that variable information may be entered into a program, and results may be printed or punched out, FORTRAN includes the READ and WRITE statements. They are of the following form:

READ (i,n) list and WRITE (i,n) list

Here "list" is a list of elements separated by commas, and an element is one of the following:

(1) An unsubscripted variable, eg., B or I.

(2) A subscripted variable, eg., B(3*I+J) or I(L+5, M, 2*N).

(3) An array name, eg., B or I.

In this case the whole array is transmitted (see Section 3.5 for transmission sequence).

(4) A "DO-loop": this is a list of elements of the form (1), (2), or (3) followed by a comma, followed by indexing of the form $I = m_1,m_2,m_3$, or $I = m_1,m_2$ the whole enclosed in brackets, where the rules for $I = m_1,m_2,m_3$ are identical with those DO statements. For example, (E(I), A(I,J), I = 1,10).

This list would cause data to be transmitted in the sequence E(1), A(1,J), E(2), A(2,J),... E(10), A(10,J).

Elements of this form may be nested. For example

(A(I),J,(B(K,J),K=L,M,N), I = 1,10)

9.2 continued

Elements or lists of elements may be enclosed in brackets. For input statements, all elements to the left of the brackets have their new value before the brackets are dealt with.

For example, K,(A(I), I = 1,K) reads one number into K and then, using this new value of K, reads A(1) to A(K).

The READ statement, where "i" is a reference to a peripheral device and "n" is an integer representing a FORMAT statement label (see Section 9.3), causes data to be read from the peripheral device to the locations specified in its list according to the format specified by the list of the FORMAT statement.

The WRITE statement, where "i" is a reference to an output device and "n" is an integer representing a FORMAT statement label, causes data to be transmitted to the peripheral device from the locations specified in its list according to the format specified by the list of the FORMAT statement.

For both statements "i" is an integer which indicates a particular peripheral device specified by a PERIPHERAL statement.

9.3 The FORMAT Statement

The FORMAT statement supplies information to the compiler about the format of data to be read or written by I/O statements which reference it. This statement may appear anywhere in the segment in which it is used. Data as it appears in the external medium is divided into "records" which are further divided into "fields". Each record corresponds to a major division in the external medium (eg., one line of type or one punched card). Each field contains either one number or a group of characters. For each field read or written there must correspond a field specification which lists the kind of information the field contains, and what its layout is. The statement takes the form

FORMAT (S_1, S_2, ..., S_n)

where each S_i is a format specification as described below. Four types of specifications (types I, F, E and A) describe the form taken in the external medium by an element from an I/O list. Two other types (types H and X) cause input or output without reference to an I/O list. In the following descriptions of the specifications, "w" is an integer that gives the total field width allowed in the external medium for a particular item and "d" is an integer that indicates the number of digits in the external medium to the right of the decimal point.

9.4 Numeric Input and Output

Three specifications (I, F and E) describe numeric input and output. Their effect on output is as follows:

(1) The "I" specification, written "Iw", causes the variable from the I/O list to be written as an integer. It will be preceded by a minus sign if the number is negative. If the variable is Real, it will be truncated to an integer prior to output.

(2) The "F" specification, written "Fw.d", causes the variable from the I/O list to be written with a decimal point. It will be preceded by a minus sign if the number is negative.

(3) The "E" specification, written "Ew.d", causes the variable from the I/O list to be written with both a decimal point and an exponent. The number will be written as a decimal fraction between 0.1 and 1.0, preceded by a minus sign if negative, and followed by a four character exponent. The exponent will be written as the letter E, a sign (blank if positive), and a two digit exponent of which the first digit will be blank if the exponent is less than ten.

In all cases, the total width of the field will be "w" characters of which leading positions may be blank. It is necessary to insure that the field width "w" is sufficiently large to accommodate the number, allowing for a sign, at least one digit before the decimal point, a decimal point and an exponent as applicable.

The I, F and E specifications also describe numeric input. In this case, however, their action is almost identical except that no "d" may be given with the "I" specification. Furthermore, as described later, only the "F" specification may be modified by a scale factor. In each case the number contained in the external field is converted to an internal Real or Integer number as determined by the item in the I/O list. If no decimal point appears in the external number, then "d" digits of the number are taken to lie to the right of the decimal point ("d" is considered to be zero for the I specification). If a decimal point appears, it overrides the "d" given in the format specification.

9.4 continued

Blanks in the input field are ignored except that they form part of the field width "w". If the number is not signed, it is taken to be positive. If an exponent appears, it may have one or two digits and if unsigned, will be taken to be positive. Furthermore, the "E" preceding the exponent may be omitted if the exponent is signed.

Both the "E" and "F" specifications may be modified by a scale factor. This factor, written "sP", may be included in the list, and will cause shifting of the decimal place for all E and F specifications. "S" is an integer which may be preceded by a minus sign. It is not necessary to use a comma to separate the factor from any following specification. If applied to an "E" type for output, the factor will cause the decimal to be shifted "s" places to the right and the exponent to be decreased by "s". The scale factor has no effect on "E" type input.

On "F" type output, the scale factor will cause the number to be multiplied by 10^S, and on input, divided by 10^S.

A scale factor affects only "E" and "F" type specifications, and applies to all such specifications encountered subsequently until a different factor is given. Thus if it is desired to modify only one specification, it will be necessary to follow that specification with a scale factor of zero.

Examples of "E" and "F" type specifications are shown in Table 9.1 overleaf.

9.5 Character Input and Output

Information in the form of text may be read and written by a FORTRAN program. This consists of characters which may be numerals, letters of the alphabet or other characters. Information held in this form is not suitable for calculation, but only for input and output.

The "A" specification, written "Aw", is used to transmit such information between an external medium and the variable specified in the I/O list. A Real variable or array element may be used for the storage of up to eight characters and an Integer variable or array element for up to four characters. If the specification calls for more than eight (or four) characters, and if the corresponding variable of the I/O list is an array element, additional characters will be stored in successive elements of the array. Care must be taken that information to be transmitted does not exceed the capacity of the variable or array element(s).

The "H" specification, written "wH", is followed immediately in the specification by "w" characters which may include blanks. This is the only place in any FORTRAN statement in which blanks are not ignored. On input, "w" characters are extracted from the input record and replace the "w" characters included in the specification. On output, "w" characters following the specification, or the characters which replaced them, are written as part of the output document.

The "X" specification, written "wX", causes "w" characters to be skipped on input and provides "w" blank characters on output.

It is not necessary to use a comma to separate the X specification or the characters following the H specification from following specifications.

9.6 Scanning of the FORMAT Statement

The FORMAT statement will be linked to the I/O statement and the corresponding data by a scanning process. This scanning proceeds from left to right, linking the first field with the first item in the I/O statement and the first specification in the FORMAT statement, and continues through data and lists until the end of the I/O statement is reached. The normal left-to-right scanning (of the FORMAT statement) may be interrupted by the use of a repetition number, "n". This is an unsigned integer which may be written immediately preceding any of the I, F, E or A type specifications. "n" indicates the number of successive fields to which the specification preceded by "n" applies. Any specification or group of specifications may be enclosed in parentheses and preceded by a repetition number. This will cause the scanning to return to the left-hand parenthesis and rescan the enclosed specifications, looping as many times as specified by the repetition number before continuing. If list data remains to be transmitted after the format specification has been completely "used", a new record is commenced, and the scanning of the FORMAT statement is repeated as follows:

(1) If the FORMAT statement contains no parentheses other than the pair enclosing the complete list of specifications, then rescanning of the FORMAT statement repeats at the beginning of the list.

(2) If the FORMAT statement contains internal parentheses, then scanning of the FORMAT statement is recommenced at the repetition number preceding the left parenthesis corresponding to the right-most internal right parenthesis.

TABLE 9.1

INPUT		
FIELD SPECIFICATION	EXTERNAL DATA	INTERNAL NUMBER
E15.6	+12345678	+ 12.345678
	1234.5678E-2	+ 12.345678
	-1.2345678E5	- 123456.78
	.012345678	+.012345678
	-1.2	-1.2
	+1234-3	+.000001234
F 15.6	+12345678	+ 12.345678
	1234.5678	+ 1234.5678
	-1.2345678	- 1.2345678
	.012345678	+.012345678
	-1.2	-1.2
	+1234	+.001234
3PF 15.6	+12345678	+ 12345.678
	-1.2	-.0012
	1234	+1.234
OUTPUT		
	INTERNAL DATA	EXTERNAL NUMBER
E15.6	+12345678.	.123457E 8
	+1234.5678	.123457E 4
	-1.23	-.123000E 1
	+.000123	.123000E -3
3P E15.6	+1234.5678	123.456780E 1
	-1.23	-123.000000E -2
	+.000123	123.000000E -6
F 15.6	+12.345678	12.345678
	-1.2345678	-1.234568
	+.012345678	.012346
	-1.2	-1.200000
3PF 15.6	+12.345678	1234567.800000
	+.012345678	12.345678
	-1.2	-1200.000000

9.6 continued

As an example of the use of the repetition number, consider the reading of an entire array. If the statement

READ (2,21) A

is written, where "A" is a previously dimensioned array whose elements all have the same format and "2" refers to an input device, the FORMAT statement

21 FORMAT (E20.7)

would be sufficient to specify the format of all elements of the array. In this case each record can contain only one element of the array. On the other hand, if the FORMAT statement had been

21 FORMAT (3E20.7)

then three elements would be read from each record. It is quite valid for a FORMAT statement in connection with an I/O statement to specify less information than is, or can be, contained in a record; however, caution must be taken that the specifications for one record never exceed the maximum size of that record.

In general, transmission ends when a new variable from the I/O list is required and there is none available. Note, however, that a "wH" or "wX" specification does not require a corresponding variable from the I/O list. Therefore, if in a transfer the next specification in a FORMAT statement is wH or wX, no call is made for a variable, and even though the last variable of the I/O list may have been used, a transfer is made according to the specification. A transfer is completed when an I, F, E or A specification or the final right bracket is encountered and no more variables remain in the I/O list.

If it is desired to start a new line at other than the final right bracket, the character "/" may be inserted following the last specification to be included on a line. In this case, the comma separating specifications may be omitted. "n + 1" slashes will cause "n" blank lines in the output record.

9.7 The Binary READ and WRITE Statements

On occasion it will be desired to be able to read or write information in binary form. Information will be written in binary format when it is desired only to read this information in again with no necessity for human interpretation of the external medium. For this purpose, two modified forms of the READ and WRITE statements have been provided. They are

READ (i) list and WRITE (i) list

where "i" and "list" have the same meaning as for the basic READ and WRITE statements (section 9.2). The binary READ statement reads binary data from the peripheral device to the locations specified in the list. The binary WRITE statement transmits binary data from the locations specified in the list to the peripheral device designated. No "n" is specified, as a FORMAT statement is not required with these READ and WRITE statements.

The binary WRITE statement produces one "logical record". This may consist of more than one physical record (eg., punched card). The binary READ statement cause one logical record to be read. The sequence of modes (Real or Integer) of the two lists must correspond. If the list of the READ statement is shorter than the logical record, the remainder of the logical record is ignored. If the list of the READ statement is longer than the logical record, only one logical record is read and the remaining elements of the list are unaltered.

9.8 The RUNOUT Statement

This is a statement of the form

RUNOUT i

where "i" is a reference to a paper tape punch. Approximately 3 inches of blank tape will be output. "i" is an integer which refers to a particular tape punch specified by a PERIPHERAL statement.

9.9 The ENDFILE Statement

This is a statement of the form

ENDFILE i

where "i" refers to a paper tape punch. The statement will cause the following to be output from a punch; 13 inches of blank tape, a form feed character, an "End of Tape" marker (TC4), 8 erase characters, and finally another 13 inch length of blank tape. "i" is an integer which indicates a particular tape punch specified by a PERIPHERAL statement.

CHAPTER 10

MISCELLANEOUS STATEMENTS

10.1 The INTEGER and REAL Statements

In the preceding chapters it has been stated that the name of a variable, array or function defines its mode. This means that the mode of a variable such as JOE would be Integer while a variable BOB would be Real.

In FORTRAN there are two statements which enable the programmer to remove this restriction on names. These are the INTEGER and REAL statements, written in the form

INTEGER $a_1, a_2, a_3, \ldots$

and REAL $a_1, a_2, a_3, \ldots$

where each element a_i, is any name which would ordinarily indicate the mode of the data it represents. No name may occur in more than one "type" statement (INTEGER or REAL) within a segment, nor may it occur in a "type" statement after it has appeared in an executable statement.

An INTEGER statement defines the mode of all elements in its list to be "Integer", regardless of the actual names in the statement.

Similarly, a REAL statement defines the mode of all elements in its list to be "Real". A "type" statement applies only to the segment in which it appears. Thus if a name that occurs in a "type" statement appears in a PUBLIC statement in the same segment, it must also appear in a "type" statement in any other segment containing a PUBLIC statement which includes that name. Similarly a function name which appears in a "type" statement in one segment must also appear in a "type" statement in any other segment in which it occurs, including the FUNCTION segment itself.

10.2 The ENTRY Statement

A program will normally be started at the first executable statement in the MASTER segment. If it is desired to start at a different point, however, the ENTRY statement may be used. This is a statement of the form

ENTRY $(n_1, \ldots n_k)$

where the successors, n_i, are statement labels. Blank or zero successors are not allowed. This statement allows the programmer to define nine additional entry points besides the normal one. The operator has the facility of starting the program at any of the additional entry points if so desired. The statement may appear only within the MASTER segment. Only one ENTRY statement may occur in this segment and there can be at most 9 successors.

The ENTRY statement may be helpful in establishing a restart routine if a program must be interrupted and continued later at the point at which it was interrupted. The statement can also be useful if a program is designed to do several closely related jobs, the particular job to be done being selected depending upon the entry point.

10.3 The PAUSE Statement

The PAUSE statement may be used at a point in the program at which operator attention is required (such as loading a new input tape). It is written in the form

PAUSE or PAUSE n

where "n" is an integer less than 4096.

10.3 continued

The statement, when obeyed, causes the compiled program to be suspended and a message to be printed on the console typewriter:

INSTRUCT #name or INSTRUCT #name n

respectively, where "name" is the name of the program as defined by the first four characters of the name in the MASTER statement. The use of "n" is helpful when there is more than one PAUSE statement and it is desired to distinguish one from another. The operator may then continue or take special action as desired dependent upon the particular situation.

10.4 The STOP Statement

The STOP statement may be used as the last statement to be executed by the running program, or to indicate that the program can proceed no further (due to such things as invalid input data). In the former case, the program has been completed and normally may then be deleted from store. The statement is written in the form

STOP or STOP n

where "n" is an integer less than 4096. The program will be suspended and the following message, asking the operator to verify that the program is to be deleted, will be typed on the console typewriter.

ABOLISH? #name or ABOLISH? #name n

Here "name" is the program name as specified by the first four characters of the MASTER statement. "n" may be used, as for the PAUSE statement, as a halt number to distinguish various error stops from one another and from the end-of-program STOP statement.

10.5 Further Means of Dimensioning Arrays

If an element of a COMMON, PUBLIC, INTEGER or REAL statement is an array name, it may have its dimensions appended as Integer constants separated by commas.

Example:

Array name	A
Dimensioned array	A (7,9,4)

If an array is dimensioned in one of the foregoing statements it may not be dimensioned elsewhere in the same segment. If not dimensioned in one of these statements, the array name must appear in a DIMENSION statement.

CHAPTER 11

PROGRAM TESTING

Before using a FORTRAN program with actual data, the user should test it. This may be done by using simple test data as input to the program, manually calculating the results which ought be be produced, and checking the results produced by the compiled program against them. If the program produces incorrect results, it will be necessary to find the error which caused them. To facilitate this, three statements have been supplied. The first is:

MONITOR

This statement indicates to the compiler that any statements of the other two types encountered during compiling are to be included in the object program. If this statement does not occur before the appearance of the other statements, then they will be ignored during the compiling of the program. Only one MONITOR statement is necessary even in a multi-segment program, and by convention, it will be the first statement to be read by the FORTRAN compiler.

The other two statements are the MONITOR WRITE and MONITOR FORMAT statements and are written in the same way as the WRITE and FORMAT statements except that the word MONITOR precedes the normal statements. If the MONITOR statement has appeared prior to the occurrence of the MONITOR WRITE statement, the compiled program will write out a list of the variables named in the statement. The format in which they will be written will be determined by the MONITOR FORMAT or FORMAT statement specified in the MONITOR WRITE statement.

If the MONITOR WRITE statement is written at various points throughout the program, then the contents of variables at these points may be written out during the testing of the program. By checking their values, as the program progresses, against the intermediate values obtained in manually calculating the results which ought to be given from the test data, any differences can be noted and the points of error located.

When the program has been tested and found correct, the initial MONITOR statement may be deleted from the source program. When this final version of the program is compiled, any MONITOR WRITE or MONITOR FORMAT statements will be ignored and only the actual program will be compiled.

A further facility in the use of these statements is provided. The MONITOR WRITE statement can be ignored at execution time even though compiled into the program. However, this is controlled by the computer operator.