R11

PRELIM. SPECS.

F O R T R A N   II

for the

U N I V A C   S O L I D - S T A T E   C O M P U T E R S

(U S S   I   &   U S S   I I)

(by D. E. Knuth)

As presented at
UNIVAC USERS Conference
Palm Springs, California
October 3, 1962

U N I V A C

D I V I S I O N   O F   S P E R R Y   R A N D   C O R P O R A T I O N

* * * * * *

* * * *

* *

*

CHAPTER I

I. INTRODUCTION

Fortran is a language which was created to simplify the preparation of problems for computers. A programmer can write out his method of solution in the Fortran language, which resembles ordinary mathematical notation, and the computer will carry out the translation from this language into its own peculiar machine code. It is not necessary for the programmer to know anything about machine code; he need only concern himself with the Fortran II language, since the other details are carried out automatically by the Fortran translator program.

As an example of how a Fortran program looks, here is a simple one which reads in 100 numbers A(1) through A(100), computes 1/A(1) + 1/A(2) + ... +1/A(100), and then prints out the numbers together with the answer. The numbers are punched 10 to a card, each occupying 8 columns of the card. The program in FORTRAN could be written as follows:

```
C     EVALUATE SUM OF RECIPROCALS
      DIMENSION  A(100)
      READ 3, A
      SUM = 0.0
      DO 1, I = 1, 100
1     SUM = SUM + 1.0/A(I)
      PRINT 2, A, SUM
      STOP
3     FORMAT (10F8.2)
2     FORMAT (19HTHE 100 NUMBERS ARE, 10(2/10F10.2), 2/,
     1        31HTHE SUM OF THEIR RECIPROCALS IS, E30.8, 43/)
```

The C at the left of the first line indicates it is merely a comment. The second line says that A(1) through A(100) is an "array" of 100 numbers. The third line causes 10 cards to be read in, containing the 100 values. Then a running sum is started at zero. The "DO" statement says that the next line, statement 1, is to be executed for values of I = 1, 2, 3, ..., 100 .

After 100 times, the desired sum has been evaluated, and the PRINT statement causes printing out of the data and the answer.

Do not expect to understand the preceding Fortran program in detail, it was just given as an example to introduce the flavor of Fortran languages.

We will now begin to discuss the details of the language.

USS FORTRAN II.

This manual contains the specifications for the Fortran II compilers for Univac Solid State Computers. There are five versions of the compiler:

Version Number	Configuration
8001	80-column cards with 5000-word drum.
8002	80-column cards with 2600-word drum, 1280-word core.
9000	90-column cards with 5000-word drum, no tape.
9001	90-column cards with 5000-word drum, tape system.
9002	90-column cards with 2600-word drum, 1280-word core.

These systems must have hardware multiply-divide, a read-punch unit, a card reader, and a printer with at least 100 print positions. No use is made of magnetic tape in these versions. The entire Fortran system programs are built to be run with the "9800 suppress" switch set ON to turn off the band-modification feature with index registers.

This Fortran II language is actually somewhat more than many Fortran II translators will accept. Programs written for other Fortran II translators are acceptable to this translator without change, except

(1) The word length is a fixed size of 8-place precision for floating point numbers and 4 digits for fixed point numbers.

(2) The allocation of COMMON storage is made in ascending order.

(3) The 8002 and 9002 versions do not accept EQUIVALENCE statements.

(4) Arithmetic Statement functions must be rewritten in a straightforward way as FUNCTION subprograms.

(5) Six-letter identifiers whose first five characters agree, such as ALPHA1 and ALPHA2, must be changed so that their first five characters are different. There are also a few new reserved words.

(6) Names of library functions should be changed (i.e. LN for LOGF), and this can be done by defining the word LOGF (see Appendix I).

(7) IF (SENSE SWITCH) has no meaning on the solid state computer.

CHAPTER II

Constants, Identifiers and Special Symbols

The Fortran language is made up of constants, identifiers, and special symbols.

Constants

Constants are used to represent numbers which appear in formulas. Each constant corresponds to a ten-digit number inside the machine.

There are four kinds of constants:

A. Integer constants.

The simplest kind of constant is an "integer constant" which is merely composed of one to four digits. These appear inside the computer as a number with extra zeroes attached at both the right and the left. Examples:

Fortran constant	Machine representation
1234	0012340000
123	0001230000
1	0000010000
0	0000000000

B. Floating point constants.

Integer constants can represent only whole numbers, but "floating-point" constants allow a great range of numbers. Examples of floating point constants are 3.1415927 or .005 or 6024E23. The last constant means 6.024 times 10^{23}; the letter E is used in Fortran to indicate a power of 10.

You can always tell a floating-point constant from an integer, since a floating point constant always has a decimal point or an "E" in it. The 19 possible types of floating-point constants are summarized in chart form as:

$$\left\{ \begin{array}{l} \text{number} \\ \text{blank} \end{array} \right\} \, . \, \left\{ \begin{array}{l} \text{number} \\ \text{blank} \end{array} \right\} \left\{ \begin{array}{l} \text{blank} \\ \text{E number} \\ \text{E + number} \\ \text{E - number} \end{array} \right\}$$

or

$$\left\{ \text{number} \right\} \left\{ \begin{array}{l} \text{E number} \\ \text{E + number} \\ \text{E - number} \end{array} \right\} .$$

Floating constants may have any number of leading zeroes; but after leading zeroes they should be rounded to eight significant digits at most.

Inside the machine a standard representation is given which accounts for the name "floating point". The ten-digit number YYXWWWWWWW is either

(1) zero, representing 0.0
(2) non-zero, in which case it is true that X is not zero. Then the number represents X.WWWWWWW E(YY - 50) .

For example, 5010000000 represents 1.0 E 0 = 1.0 ; similarly, 5212000000 represents 1.20 E 2 = 120.

More examples of floating point constants:

Fortran constant	Machine representation
1.0	5010000000
3.1415927	5031415927
.2345E-27	2223450000
5E20	7050000000
.0000000000001	3710000000

There are at least 34 different ways to write the number 1.0 as a floating point constant using 5 characters or less; how many of these can you find? They all have the same machine representation.

C. Hollerith Constants

The remaining two types of constants are used mainly in non-scientific Fortran programs. Alphabetic information is represented by so-called "Hollerith constants", named after a man who pioneered in punched card equipment many years ago.

These constants consist of 1 to 5 letters, and are written as nH followed by the alphabetics, where n is the number of letters. (Numerics and special characters may be used as well as alphabetic letters.)

The machine representation of Hollerith constants differs for 80 and 90 column equipment. In the 80-column versions this representation is ZZZZZNNNNN where Z represents the zone part and N the numeric part of the character in MC-6 machine code. On 90-column systems the representation is PPPPPUUUUU where P represents the primed part and U the unprimed part of the character in Remington card code. If the number of letters is less than 5, zeroes are filled in at the right.

Examples of Hollerith constants:

Fortran	80-column machine	90-column machine
1HA	1000010000	2000070000
5HABC12	1110012312	2010277122
1H/	3000010000	3000090000

D. Machine constants

Machine constants make it possible to represent any 10-digit bit configuration on a Solid State computer. These are written nM followed by n numbers or the letters A, B, C, F, G, or H. The letters are used to represent "undigits" 4/1, 4/2, 4/3, 4/6, 4/7, and 4/8 respectively. If n is less than 10, zeroes are filled in at the left.

Examples of machine constants:

Fortran constant	Internal representation
1M5	0000000005
10M120ABCFGH9	$120^{444444}_{123678}9$
5MBBBBB	00000^{44444}_{22222}

Types

Quantities manipulated in the Fortran language are of three "types": integer, floating-point, or unspecified. It is important for the Fortran programmer to be careful about the "types" of the quantities he uses.

Integer type means the values have a range of integers only, with the possible values of

-9999, -9998, ... , -1, 0, +1, ..., +9998, +9999.

Floating point type means the values have a range of anywhere from 1.0E-50 to 9.9999999E49, positive or negative; or zero.

Unspecified type means the value can be thought of as either integer or floating, or that the type is immaterial.

A constant of zero has unspecified type; other integer constants have type integer, and other floating point constants have floating point type. Hollerith and machine constants have unspecified type.

The question that probably is occurring now to the reader is: Why so many types? All the possible values for type integer occur also in the floating-point range, so why could we not dispense with integer type completely? The reason is that calculations with floating-point quantities are more difficult for the computer, so they take much more time for execution. Therefore if one is working solely with integers it is much faster to use integer type, and only when fractional quantities or very large quantities are involved need the slower floating-point arithmetic be used. Unspecified type is useful when performing operations on alphabetic and other non-numeric quantities.

Identifiers

Identifiers are the names given to Fortran quantities. An identifier is a string of letters and digits, starting with a letter. Examples are A, I, ALPHA, A23, A24, B280, FORTRAN. The letters and digits used do not have any special meaning in the Fortran language. The programmer is free to choose the names for his own convenience. There is no connection between A23 and A24. An identifier ends with the first non-letter or non-digit following it.

It is preferable to choose names which have five characters or less, although it is possible to use more letters. When using longer names there is a chance that two different ones will be treated as though they were the same. In USS Fortran II two identifiers are "the same" if

(1) they have the same length, and
(2) they have the same first five characters.

ALPHA is different from ALPHA1, but ALPHA1 is the same as ALPHA2.

There are some identifiers which must not be used as names, because they have special meaning in Fortran. These words, such as READ, PRINT, IF, DO, AND, etc. will be introduced in the text giving them special significance, and they are all listed together in Appendix II for reference.

Important note: The first letter of the identifier determines its type.

Identifiers are used to give names of variables, and each variable takes on either integer values or floating-point values. If the first letter is I, J, K, L, M, or N, the variable may take on only integer values and is of type integer. Any other first letter means the variable may take on only floating-point values, and is of floating-point type.

Meaning of Blank Spaces

In most cases blank spaces may be inserted freely in Fortran statements; that is, we might equally well write X = (Y + 1.0) as X = (Y + 1 . 0) . However, spaces must not appear in the middle of an identifier. More exactly, a space is ignored except that

(1) It marks the end of an identifier
(2) After the letter H in Hollerith constants or in Format strings it indicates the character "space".
(3) After the letter M in machine constants a blank space should not be used. That is, 3M123 must not be written 3M 123.

A space is important at times to separate identifiers; for example in the program given earlier it would have been incorrect to write READ2 or DO1, or PRINT3. There are other cases such as "C AND Y" where both spaces are necessary, since "CANDY" has quite a different meaning.

Special Symbols

We have so far introduced uses for letters, digits, spaces, and decimal points. The other special characters which appear in USS computers have significance in the language too, as follows:

- # is used for an = sign
- , ( ) are punctuation marks used in the ordinary manner
- + - * / are used for arithmetic operations
- ; may be used to put several statements on one line, rather than starting a new line for each one
- ' (apostrophe) must never be used except in Hollerith constants

Since some "Fortran" keypunches for 80 column cards have special codes for certain symbols, the other codes are also acceptable and are equivalent internally, according to the correspondence below:

% may be used for (
: may be used for)
& may be used for +
$ may be used for ;

CHAPTER III

Variables and Expressions

Simple Variables, Array Variables

All symbols in Fortran are written on one line, so "subscripts" are not really written as lowered numbers (e.g., A_1, A_2, etc.) but rather are written using the parenthesis convention A(1), A(2), etc. They are called subscripts anyway.

By a "simple variable" we mean a variable which has only a single value and therefore has no subscripts. It is also possible to have subscripted variables, in which case a single name references many values. For example, in the first program given the 100 numbers A(1), A(2), etc. were all values of the subscripted variable A. Subscripted variables are called "array variables".

It is possible to use many subscripts, if desired. A vector variable usually has just one subscript, but matrix variables usually have two; when more than one subscript appears, commas are used to separate them, e.g., B(3,4).

Before an array variable is used in a Fortran program, a "Dimension declaration" is given to tell how many subscripts there are, and how many values the array has. For example,

```
DIMENSION  A(100), B(10,20), K(2,2,2,4)
```

specified that A, B, and D are array variables. A is a vector with elements A(1) through A(100). B is a matrix with 10 rows and 20 columns, and K is a four-dimensional array with 32 elements (a typical element is K(2,1,1,3)). The DIMENSION declaration will be discussed in more detail later.
The type of an array is determined by the first letter of the name, just as for simple variables. Thus, A and B take on only floating point values, while all elements of K have integer values.

Arithmetic Expressions

Variables and constants are combined with arithmetic operations to form expressions which look very much like ordinary algebraic expressions. But in this case also it is important to be able to write the expression on one line. Thus, instead of

$\frac{X + Y}{Z}$ we would write (X + Y)/Z .

Exponents (superscripts), as in X^2, cannot be used in formulas for the same reasons, so there is a new way to write exponents. Two asterisks in a row are used to indicate exponentiation (i.e., taking to a power). For example, X^2 is written X**2. The expression

X^{Y^Z} would be written X**(Y**Z) .

The symbols + and - are used in the usual way, for addition and subtraction, and a minus sign may also be used to indicate negation, e.g., -X.

The symbol * is used for multiplication. This sign must always be used, and it is important not to drop it out as is often done in algebra. One should not write 2(I + J), it should rather be written 2*(I + J):

The symbol / is used for division.

Use of Types in Arithmetic

Care should be taken not to mix integer quantities with floating-point quantities in an arithmetic expression. It is perfectly proper to write X + Y or I + J, but one must not write X + J or I + Y. Similarly, it is correct to write A(3) + 1.0, but not correct to write A(3)+1 since the constant 1 is integer type and the array A is floating-point. Array subscripts (as the 3 in A(3)) are integer.

If either operand is of unspecified type the calculation will proceed as

if it has the type of the other specified operand. The following table lists all possibilities for the binary operations +, -, *, / . Here I stands for integer, F for floating, U for unspecified.

Operation	Type of Arithmetic Used	Types of Result
F op F	F	F
F op I	not allowed	--
F op U	F	F
I op F	not allowed	--
I op I	I	I
I op U	I	I
U op F	F	F
U op I	I	I
U op U	I	U

If the programmer wishes to perform the operation X + J it is possible to do this in floating-point arithmetic by writing X + FLOAT(J). In general if we have any integer expression like J+3*K(1,2,1,4) it can be changed into floating-point type by writing FLOAT(J+3*K(1,2,1,4)).

The oposite function to "FLOAT" is "FIX". If we have any floating-point expression like A(3) + 3.0*X we can write FIX (A(3) + 3.0*X) to convert it to integer type. In this case any digits to the right of the decimal point are dropped, there is no rounding. For example, FIX(3.1) = 3, FIX(+3.9) = 3, FIX (-3.9) = -3. If rounding is desired for a positive quantity Q, one could write FIX(Q+.5). General rounding for positive or negative Q can be accomplished by using some tricks which will be explained later, by writing

```
FIX(Q + (Q AND 0 OR .5)).
```

Integer division operates like FIX in dropping the fractional part. For example, 12/5 = 2.

Exponentiation has some special rules. When doing integer exponentiation I**J, if J is negative the fractional part is dropped as in FIX. This usually gives a result of zero, e.g., 3**-2 = 0.

When doing floating-point exponentiation X**Y the base X must be a positive quantity, not zero or negative. In this case the logarithm of X is evaluated and multiplied by Y, then antilogarithms taken to give the proper result.

It is possible to mix floating-point and integer types only in one case, and that is with exponentiation. Although it is illegal to write X + I, it is quite all right to use X**I. It is, however, still not possible to use I**X. When doing floating-point to an integer power as in X**I there is no restriction that X be a positive quantity.

Boolean Operations

When doing non-numeric calculations it is occasionally useful to use the 10-digit numbers manipulated by the Univac in other ways as if they were 40 independent bits which were either on or off. To use all facilities of the Boolean operations of Fortran requires a knowledge of the way information is represented in the computer. However, for operations on the integers 0 and 1 (taking 0 as "false", 1 as "true") the operations NOT, OR, and AND have the following effect:

I	NOT I
0	1
1	0

I	J	I AND J	I OR J
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

The result of a Boolean operation has unspecified type.

It is also possible to obtain more complex Boolean manipulations on entire computer words. NOT interchanges 1 and 0 throughout a word. AND is the solid-state "erase" command which produces 0 bits except where inputs are 1;

OR is the Solid-state "buff" command which produces 1 bits unless both inputs are 0. NOT, AND, OR operate differently on values of type integer, floating and unspecified. The result of any Boolean operation is of unspecified type.

```
Examples:  1.0 AND  10MHH00000000 = 10M5000000000,
           1.0 AND   8MHHHHHHHH   =   1000,
               NOT  P  OR  P      = 10MHHHHHHHHHH
               NOT   8MHHHHHHHH   = 10MHH00000000 .
```

The result of NOT always has a plus sign, but the result of AND and OR with signs is difficult to predict. For simple operations like "P AND Q" where P and Q are simple variables or constants the sign of P is the sign of the result. It is best not to use AND and OR with signed numbers, except in the cases

```
Q AND 0 OR .5
```

which gives a constant of .5 with the sign of Q,
or

```
-1 AND 0 OR Q
```

which unconditionally attaches a minus sign to the quantity Q . Note that -10MHHHHHHHHHH is _not_ the constant "minus all H's" because of the way subtraction works on the computer; to get "minus all H's" one should write -1 AND 0 OR 10MHHHHHHHHHH .

Precedence of Operators

It is customary in Algebra to perform certain operations first; for example, in A+B*C we would calculate B times C first, then add A. If one writes X**Y**Z, however, some people would interpret this as X**(Y**Z) and others would interpret it as (X**Y)**Z. Fortran takes the latter interpretation (which is effectively X**(Y*Z)). In certain cases it is important to know which order is taken by Fortran.

Operations are done in the following order:

NOT
**
* /
+ -
AND
OR

In case of ties, operations are done left to right.

Examples:	Fortran interpretation
A+B*C	A+(B*C)
A*B+C	(A*B)+C
A/B*C	(A/B)*C
A*B/C	(A*B)/C
-X**Y	-(X**Y)
-X-Y	(-X)-Y
A+B OR I	(A+B) OR I
-X-Y-Z	((-X)-Y)-Z
-X**-Y	-(X**(-Y))

Note especially the third line A/B*C = (A/B)*C which is contrary to many people's ideas and is a frequent source of errors.

As a final example of writing expressions the reader should try to write the discriminant for a quadratic equation: $b^2 - 4ac$. The correct answer is

B**2 - 4.0*A*C .

Notice several things:

(1) The exponent 2 is written using the ** operator.
(2) The multiplication signs between 4 and a and c must be written.
(3) It would have been wrong to write 4*A*C because 4 is an integer constant which can't be mixed with floating variables A and C.

(4) It would have been wrong, on the other hand, to write B**2.0 even though B is floating point, since B**2.0 is allowed only for positive values of B. B**2 is the correct form for positive, zero, and negative values of B.

(5) The precedence of operators causes the above expression to be interpreted (B**2) - ((4.0*A)*C).

If any of these things seem unfamiliar to the reader at this time, he should reread the present section.

Array subscripts

Any expression which is of integer type may be used for subscripts, with the restriction that the final value of the expression is between 1 and the maximum size of the dimension. A subscript expression which has the value 0 or less must never be used. Here are some examples, which show various degrees of complexity in subscripts:

```
A(I+5)
K(2,2,2,2)
A(I**J+3*FIX(X-Y/Z))
M(M(M(I+2)))
B(M(I), M(J))
A(FIX(A(I)))
K(K(2,2,2,2), I, J, M(3))
```

More information about array subscripts is given in Appendix III.

Standard functions

To take the absolute value of a quantity, write ABS; e.g., ABS(X+Y) or ABS(I). The type of the result is unchanged, i.e. ABS(I) is integer type, while ABS(X+Y) is floating point.

The standard functions of analysis are standard equipment in Fortran also. These functions are:

SQRT(X)	square root of X
LN(X)	natural logarithm of X
EXP(X)	2.7182818**X (e to the X).
SIN(X)	sine of X, X in radians
COS(X)	cosine of X, X in radians
TAN(X)	tangent of X, X in radians
ARCTAN(X)	arctangent of X, in radians

Only floating point expressions may be used as parameters. It is incorrect to write SQRT(I).

Final examples of arithmetic expressions:

The quadratic formula $\frac{-b + \sqrt{b^2 - 4ac}}{2a}$

= (-B + SQRT(B**2-4.0*A*C))/(2.0*A).

The Wolontis function $\frac{\text{SIN } X}{\sqrt{1+e^{-X^3}}}$

= SIN(X)/SQRT(1.0+EXP(-X**3))

CHAPTER IV

STATEMENTS

In the first three chapters we have developed the basic fundamentals of the Fortran language. Now we are ready to learn how to put them together to describe a problem-solving procedure.

The Arithmetic Statement

The arithmetic statement, or replacement statement as it is sometimes called, is of the form

variable = expression

and means that the value of the expression is to be evaluated, then stored as the new value of the variable.

Examples:

A = B	means store the value of B in A.
I = I+1	means increase the value of I by 1.
T(I) = 0	means clear the value of T(I) to zero. T is an array variable.
I+1 = I	is improper; there must be only a variable at the left of the equal sign.

If the expression is of type integer and the variable is floating point, the statement is automatically converted into

"variable = FLOAT (expression)" .

Similarly a floating expression with an integer variable would be converted into

"variable = FIX (expression)" .

If several variables are to be set to the same value, this can be done by using several equal signs, e.g.

X = Y = Z = 0,

which sets X, Y, and Z all to zero. The only restriction is that the variables must all be of the same arithmetic type.

Ambiguous usage should be avoided; e.g, consider

```
M(I) = I = 1 .
```

Supposing I has a value of 2, then is M(2) or M(1) set equal to 1? This situation is undefined in Fortran and should not be tried.

General Rules for Statements

A Fortran program is a series of statements punched onto cards, normally one card per statement. The arithmetic statement which we have just discussed is one of over 20 kinds of Fortran statements.

Refer to the program in Chapter I; it begins with a DIMENSION statement, then a READ statement, and then SUM = 0.0, an arithmetic statement.

If desired, statements can be numbered, as the statements numbered 1, 3, and 2 in that example. These numbers needn't be in order; they merely serve as reference numbers. The READ statement says READ 3, and this refers to the FORMAT statement number 3.

Several statements may be put on one card, if desired, by separating them with semicolons; for example,

```
X = Y = 0; I = 1; M(3) = 3*J
```

The GO Statement

Fortran statements are executed in the order written, unless special statements to break this sequence are used. The simplest way to specify a change in sequence is with the "GO statement".

Example:

```
GO   TO   3
```

means the next statement to be executed is statement number 3.

There is also a more complex form for a GO TO statement, where transfer is made to one of a list of statements. Example:

```
GO   TO   (3,3,2,4,25,1,1), I
```

This means, "If I is 1 or 2, GO TO 3; if I is 3, GO TO 2; if I is 4, GO TO 4; if I is 5, GO TO 25; if I is 6 or 7, GO TO 1; if I is zero, continue on".

The list of statement numbers may be of any length. In place of I, any integer expression may be used. The value of this expression must not be negative or greater than the amount of statement numbers.

Assigned GO TO

The statement "ASSIGN 3 TO I" means that the simple variable I is replaced by the machine address of statement number 3. It is possible then to say "GO TO I", which would mean GO TO 3 in this case. Subscripted variables may not be used in this context.

IF statement

A powerful 3-way branch is provided by the "IF statement",

IF (expression) n, z, p.

Here n, z, and p are statement numbers, and the meaning is: if the value of the expression is negative, GO TO n; if the value of the expression is zero, GO TO z; if the value of the expression is positive, GO TO p. The parentheses around the expression are important, even though the expression may be quite simple.

Examples:

IF(X)3,3,5	if X is greater than zero go to 5 otherwise to 3
IF(I-J)6,10,6	if I equals J go to 10 elst to 6
IF(ABS(X-XBAR)-EPSILON)2,4,4	if $\lvert X-\bar{X} \rvert < \varepsilon$ go to 2 else 4.

The above definition of the IF statement is good enough for all Fortran programs, except those which are using alphabetic or non-numeric data, when a more precise definition is necessary. This is because the positive constant 1MA for example, which involves the undigit A, is less than zero. A more accurate definition of the IF statement deals with the internal ten-digit machine representations, as follows:

The form of an IF statement is changed automatically to

IF (expression - expression) n, z, p,

if it is not of this form already, by rewriting it

IF (expression - 0) n, z, p.

Once the IF statement is in this form a comparison between the values of the two expressions is made; they are *not* subtracted. The result of this comparison, less, equal, or greater is used to make the 3-way branch. The rules for comparison are:

(1) Any number with a minus sign is less than any number with a plus sign

(2) The order of the digits and undigits is
$C < B < A < 0 < 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < H < G < F$.

It is impossible to generate the value "minus zero" in Fortran object programs without using Boolean operations.

The DO Statement

A common occurrence in programming is the desire to execute a series of statements repeatedly as the value of some variable gets changed a step at a time. For example, in our first program (Chapter I) we wanted to add up 1/A(1) + ... + 1/A(100). It is the "..." which is an immediate clue that a DO statement is desired.

The statement "DO 4, I = 1, 100, 3", as one example, means: "Do all statements from here down to statement number 4 for values of the variable I equal to 1, then 4, then 7, then 10, etc. until I is greater than 100". The "1" means I is to start at 1; the "100" means I is to stop at 100; and the "3" means that I goes up by 3 each time.

The general form for a DO statement is

DO statement-number, simple-variable = start, stop, step.

The comma after the statement number is optional and may be omitted. In fact in many Fortran systems it <u>must</u> be omitted! The other commas are essential. Another form for a DO statement is

Do statement-number, simple-variable = start, stop

in which case "step" is taken to be 1.

In these forms, "start", "stop", and "step" represent expressions of the same arithmetic type as the simple variable. If these expressions are anything more complicated than a simple variable or constant, they are evaluated only once, <u>before</u> the DO "loop" is started. These expressions must all have positive values, and they must not involve the simple variable; e.g., it is <u>improper</u> to write

```
DO   2,   I = 1, I
```

The statements after the DO up to and including the final statement indicated by the statement number are called the "range" of the DO statement. The simple variable is called the "index" of the DO statement. A DO statement is often called a "DO loop".

Example:

```
      DO 12, J = 1, 10
         B(J) = M(J)
12       A(J) = J*M(J)
```

Here the range of the DO statement is the pair of arithmetic statements, and the index is J. These statements cause the following to take place:

(1) M(1) is converted to floating point and stored in B(1)
(2) 1*M(1) is converted to floating point and stored in A(1)
(3) M(2) converted to floating point and stored in B(2)
(4) 2*M(2) is converted and stored in A(2)
⋮
(20) 10*M(10) is converted and stored in A(10).

If several statements separated by semicolons are on the same card as the numbered statement, they are also included in the range of the DO. For example, the above could have been written equivalently

```
        DO 12, J = 1, 10
   12      B(J) = M(J); A(J) = J*M(J)
```

There may be other DO statements in the range of a DO. For example:

```
        DO 7, I = 1, 19
        DO 7, J = I+1, 20
    7   B(I,J) = B(J,I)
```

These statements cause the following sequence of operations:

```
  (1)  B(1,2)   = B(2,1)
  (2)  B(1,3)   = B(3,1)
   .
   .
 (19)  B(1,20)  = B(20,1)
 (20)  B(2,3)   = B(3,2)
 (21)  B(2,4)   = B(4,2)
   .
   .
 (37)  B(2,20)  = B(20,2)
 (38)  B(3,4)   = B(4,3)
   .
   .
(190)  B(19,20) = B(20,19)
```

The range of a DO statement is always executed at least once. The sequence can be written in terms of other Fortran statements, e.g.:

```
          variable = start
          GO TO  1
    2     IF (variable + step - stop) 1,1,3
          variable = variable + step
    1     range of DO statement
          GO TO  2
    3     next part of program.
```

There are three important rules to observe when using DO statements:

1. If the range of a DO statement includes another DO statement, the range of the latter must be entirely contained within the range of the former.

2. It is not legal to GO TO a statement inside the range of a DO statement from outside the range of that statement. Entry must be made only via the word DO. It is quite all right, however, to go out of the DO loop from inside, and this terminates the loop.

3. After a DO loop is finished, there is no telling what value the index variable has. If the termination of the loop was caused by a GO TO leading out of the loop, however, the index variable will retain its last value.

Example: It is known that one of the elements of the 100-place array A is zero, but the problem is to find the **first** one which is zero. Solution

```
          DO 2 J = 1,100
          IF(A(J))  2,3,2
2         CONTINUE
3         at this point J indicates the first zero value.
```

(The CONTINUE statement does nothing but it is often used to indicate the end of a DO loop.)

A final example for DO loops is to calculate n! = 1*2*3*...*n. For n bigger than 7, n! is bigger than four digits, so we will calculate the value in floating point. Solution:

```
            FN = N
          FACT = 1.0
         DO 1,X = 2.0,FN,1.0
1         FACT = FACT*X
```

PAUSE and STOP statements

There are two ways to stop the computer if necessary while it is running your program, using PAUSE and STOP statements. The allowable forms are

```
          PAUSE
          STOP
or        PAUSE   expression
          STOP    expression .
```

In the latter cases the value of the expression is displayed in register A when the machine stops, After a PAUSE, the machine can be restarted at the same point in the program; but STOP is a final, unrestartable halt.

NOTE: PAUSE can be used in a peculiar way for low-volume input and output. Consider the statement

```
X = PAUSE (Y)
```

THE MACHINE STOPS, displaying the value of Y. The operator keys a value in floating-point code into register A and then depresses RUN, whereupon the keyed-in value is stored in X.

CHAPTER V

INPUT-OUTPUT

We now know how to compute the answers for the most complex problems (more or less) but still haven't a good way to get data into the computer and to get answers out. Fortran provides an elaborate set of facilities for both input and output, allowing quite flexible formats for these purposes.

In the example program in Chapter I, the output statement

```
PRINT 3, A, SUM
```

appears. Here 3 is the statement number of a FORMAT statement (we will discuss FORMAT statements later), which specifies in what manner the answers are to appear on the page. Following the format number is a list of the values which are to be output. In this case A is an array which appeared on the dimension declaration as A(100), while SUM is a simple variable. This is a list of 101 outputs, 100 for A and 1 for SUM.

Another way to write the same statement would be

```
PRINT 3, (A(I), I = 1,100), SUM
```

where a notation analogous to DO loops is used.

To illustrate these "implied DO loops" we will dive into an extremely complicated example. If the reader can figure out the following example he will have no trouble with any input-output lists:

```
PRINT 3, A, B(3),(C(I), D(I,K)**2, I=1,10),
((E(I,J), I=1,10,2), F(J,3), J=1,2*K)
```

The following "program" describes what happens:

```
        DO 1, I = 1,100
1       OUTPUT  A(I)
        OUTPUT  B(3)
        DO 2, I = 1,10
        OUTPUT  C(I)
2       OUTPUT  D(I,K)**2
        DO 3  J = 1, 2*K
        DO 4  I = 1,10,2
4       OUTPUT E(I,J)
3       OUTPUT F(J,3)
```

As another extreme, the list may be completely empty, although the comma must still appear:

```
PRINT  3,
```

In this case no answers are output, but the format 3 might be merely a title line or one which skips to the top of a page.

The items appearing in an input list may only be variables, but any expression is permitted in output lists. In the example above, D(I,K)**2 would be allowed in an output list, but not in an input list. The only restriction is that functions occurring in input-output lists do not use any input or output themselves.

The general forms of input-output statements are:

```
READ format-number, input list
PUNCH format-number, output list
PRINT format-number, output list
```

Instead of the format number, a simple variable may be given which was assigned the format number in an ASSIGN statement.

Input-output of arrays with more than one dimension are done by varying the left-hand dimension most rapidly. For example,

```
DIMENSION  D(2,4,6,8)
READ    2, D
```

would cause the input to go in the following order:

```
          DO 1    I = 1,8
          DO 1    J = 1,6
          DO 1    K = 1,4
          DO 1    L = 1,2
    1     INPUT   D(L,K,J,I)
```

FORMAT Statements

The format for input-output is given as a string of peculiar looking items which are a kind of program in themselves. The format statement looks like

```
    n      FORMAT    ( String )
```

where n is the number of format statement.

Example: 2 FORMAT (15H THE ANSWERS ARE,3X, 3I5)

In this example the first 15 columns are for the title "THE ANSWERS ARE". Then the "3X" specification means to leave 3 blank spaces. Finally, 3I5 indicates 3 outputs of integer numbers, each taking a 5-column field.

If we wrote PRINT 2, 30, - 512, 01 this would print a line with the following information at the left:

THEbANSWERSbAREbbbbbb30b-512bbb1

where we use the small letter "b" to indicate a blank column.

Output Formats

Now we will give the more precise rules for specifying format. At first, we will consider only output formats, as if they were written for a printer. It would be similar for a punch, except that "skipping a line" means punching a blank card, and only 80 or 90 columns may be punched on a card while a printer can hold up to 130 columns.

The format specifications are of the following forms:

1. wH followed by w characters, means the w columns are to be filled with the characters specified. For example 3HXY= means "XY=" is to be printed.

2. wX, means insert w blank columns.

3. Iw, must be used only with output expressions which are not floating point. The integer value is printed at the right of a field w columns wide, with a preceding minus sign if it is negative. The example above, with I5 and a value of -512, prints b-512. If for any of these outputs w is too small, an asterisk is printed. For example, trying to print -512 with the format I3 would print bb*.

4. Ew or Ew.d, (Ew is equivalent to Ew.0) means floating-point output and must not be used with integer outputs. The number of columns, w, must be at least d+6. Otherwise the field overflow asterisk is printed. The number is printed in floating-point constant format with the E being used to indicate power of ten notation. Signs are printed for both the exponent and mantissa. A plus is indicated by a blank column. If more room exists in the field, blanks will be filled on the left.

5. Fw or Fw.d, (Fw is equivalent to Fw.0) means floating-point output with a fixed place for the decimal point on the answer form, without an E exponent. F must not be used with integer outputs. Exactly d digits are printed to the right of the decimal point and the sign appears at the left of the field with a plus sign being indicated by a blank column. The field width w must be at least d+2

Examples

Machine Representation	Format	Output
5031415927	F3	b3.
-5031415927	F3	-3.
5031415927	F8.4	bb3.1415
5031415927	F6.5	bbbbb*
-4827160000	F4.1	bbb-.0
6031415927	F10.8	bbbbbbbbb*
6031415927	F13	b31415927000.

(Here b is used to denote a blank column)

6. Aw, means alphabetic output of w columns. The output should be values in the form ZZZZZNNNNN or PPPPPUUUUU as described in Chapter II under Hollerith constants. If w is greater than 5, an error is indicated during the running of the object program.

7. Mw, means machine output of w columns. The output should be values in the form as described in Chapter II under Machine constants. If w is less than 10 the rightmost w characters are used. If w is greater than 10 an error will be indicated during the running of the object program.

8. I, E, F, A, and M formats may be preceded by a repeat number which indicates how many adjacent fields have the same specification. For example, "2I10,3E20.6" is equivalent to "I10, I10, E20.6, E20.6, E20.6".

9. A string of specifications may be grouped in parentheses and preceded by a repeat number. For example, "2(I10,F20.6)" is entirely equivalent to "I10, F20.6, I10, F20.6". If the repeat number is omitted, it means infinite repeat, i.e. repeating until the output list is exhausted. In particular, the whole FORMAT string is enclosed in parentheses without a repeat number, thus specifying infinite repeat. The termination of output occurs when

   a. An I, E, F, A, or M field is to be processed but there are no more values in the output list.

   b. The ")" of an indefinite repeat is encountered and there are no more values in the output list.

10. /, means print the preceding information: (or a blank line if there was no preceding information). One can also say n/ which means print (or punch) the preceding information followed by n-1 lines (or cards).

11. Format specifications are separated by commas. No comma is actually necessary after a / or P or X or H field, but extra commas don't hurt.

12. The specification nP or +nP or -nP found in certain versions of Fortran is allowable in a format string but is ignored by the standard USS format package.

Input Formats

Input formats are very similar to output formats; in fact, exactly the same format string may be used to input data from cards as was used to punch it onto cards. There is much more flexibility, however, which is provided for economy in key-punching the cards.

1. Blank columns are ignored, and the number may appear in any place in the field on I, E, and F formats.

2. Plus signs can be indicated by blank columns, or by punching a + sign or an & sign.

3. Numbers input for E or F fields may be in either E or F format. The decimal point may be punched in any column, so that if a decimal point is punched in the card the "d" part in Ew.d has no effect. However, if no decimal point is punched, d decimal places are assumed.

4. An E need not be punched, it may be replaced by a plus sign (or, if the exponent is negative, by the minus sign). Thus, 3.14+5 is like 3.14E5, 3.14-5 is like 3.14E-5.

5. M input is not allowed on the 90-card system (version 9000 of Fortran).

6. wH on input means that the w characters read from the card are to <u>replace</u> the w characters of the format string; later references to this format string use the new characters.

CHAPTER VI

SUBPROGRAMS AND SPECIFICATIONS

A Fortran program may be divided into independent parts called subprograms. The general layout of the program is:

```
{subprogram}
    END
{subprogram}
    END
     .
     .
     .
{subprogram}
    END
```

FUNCTION declarations

The beginning of a subprogram may be a Function declaration, e.g.,

```
FUNCTION  F(X,Y)
```

Here F is the name of the function, and X and Y are parameters of the function. There are one or more parameters, whose identifiers are simply listed on the FUNCTION card.

The value of the function is determined by assigning a value to the function name, treating it as a simple variable.

Exit from a function must be made by using a RETURN statement or by running across the END card.

As a simple example of a function, here is one which evaluates $\sqrt{X+Y}$ if X+Y is positive, else is defined to be zero:

```
          FUNCTION  F  (X,Y)
          IF  (X+Y)1,1,2
    1     F = 0.0; RETURN
    2     F = SQRT(X+Y)
          END
```

In another program, one could write S=F(T,Q)+F(2.0,F(T,Q)).

DIMENSION declarations

Every array variable must appear in a DIMENSION declaration before it is first used as an array. The DIMENSION declaration merely lists arrays with the maximum subscript sizes, e.g.

```
DIMENSION A(10),B(3,15),K(8,2,2,4,7)
```

indicates that A, B, and K are array variables, with 1, 2, and 5 subscripts, respectively.

COMMON declarations

Because subprograms are independent, the identifier X will represent an entirely different quantity in a different subprogram. Various subprograms can communicate with each other by using parameters, but there is another powerful way to communicate, by using the COMMON declaration. For example suppose in one subprogram we have the declarations

```
DIMENSION   A(10), B(20,20)
COMMON      A,B,I
```

and in another we have

```
DIMENSION   A(100), B(200)
COMMON      A,B,X
```

The memory space used by programs is divided into three parts: program storage, data storage, and common storage. Program storage is used for the machine code instructions corresponding to the Fortran program, data storage is used for constants and variables not declared COMMON, and common storage is used for variables which appeared in the COMMON declaration. Each subprogram has its own program area, and data areas, but all use the same COMMON area. In the example above COMMON would look like this:

Location (C is base)	Subprogram 1	Subprogram 2
C	A(1)	A(1)
C+1	A(2)	A(2)
C+9	A(10)	A(10)
C+10	B(1,1)	A(11)
C+11	B(2,1)	A(12)
C+29	B(20,1)	A(30)
C+30	B(1,2)	A(31)
C+99	B(10,5)	A(100)
C+100	B(11,5)	B(1)
C+299	B(10,15)	B(200)
C+300	B(11,15)	X
C+409	B(20,20)	not used
C+410	I	not used

There are always at least 700 memory cells allocated to COMMON storage, and this storage has the highest possible priority, so as much data should be put into COMMON as possible. More details about storage allocation will be found in Appendix V.

EQUIVALENCE declarations

NOTE: The 8002 and 9002 versions of Fortran do not allow EQUIVALENCE.

The EQUIVALENCE declaration allows several identifiers to name the same variable. For example,

```
EQUIVALENCE (X,Y), (I,J,K)
```

states that variables X and Y are the same, and variables I, J, and K are the same.

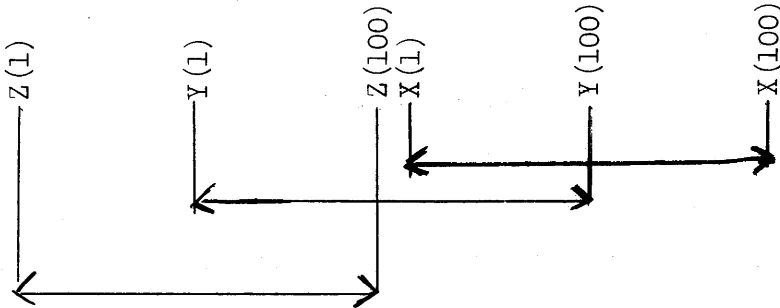
The EQUIVALENCE declarations are affected by preceding DIMENSION and COMMON. Suppose, for example, X is an array variable but Y is not. Then EQUIVALENCE (X,Y) indicates that the simple variable Y is the same as the first location X(1) of the array X.

If a variable occurring in an EQUIVALENCE has appeared in a dimension statement, constant subscripts may be given for it, e.g. EQUIVALENCE (X(3),Y) which states that Y is the same as the third location X(3) of the array X.

Several dimensioned variables may be equivalenced. For example,

DIMENSION X(100), Y(100), Z(100)
EQUIVALENCE (X(0), Y(50), Z(100))

The subscript 0 is allowed only in an EQUIVALENCE statement; negative subscripts are never allowed. The preceding reserves 200 locations

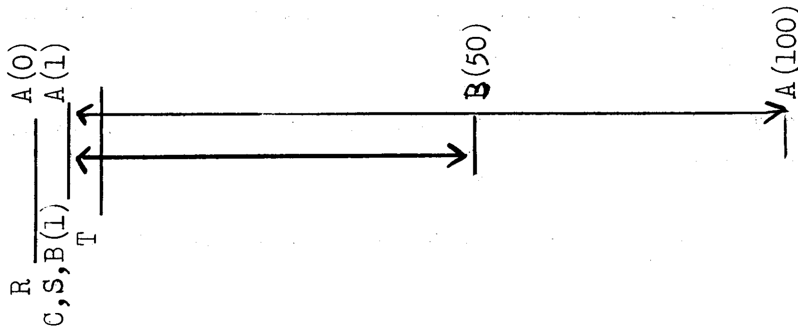


according to the pattern shown in the diagram.

A further example of what can be done using EQUIVALENCE is:

DIMENSION A(100), B(50)
EQUIVALENCE (R,A(0)), (S,A(1)), (T,B(2)), (A,B,C)

which allocates 101 storage locations as follows:



When any variables are equivalenced to variables in COMMON, no change of the COMMON assignment occurs. The COMMON statement specifies, once and for all, how COMMON storage is allocated.

The important thing is, what <u>cannot</u> be done using EQUIVALENCE?

(1) Two things may not explicitly or implicitly be equivalenced if they have both been assigned in COMMON.

(2) Two things which are already equivalent may not be made equivalent again. For example, you can't say EQUIVALENCE (X,Y), (X,Z), (Y,Z). Redundant things should be dropped.

(3) Nothing should be assigned to a place less than the lowest address of COMMON or greater than the highest address of COMMON.

(4) Nothing should be equivalenced to a parameter.

<u>Order of Declarations</u>

<u>Important</u>: The declarations listed above must appear in the following order if used:

```
FUNCTION or SUBROUTINE
 DIMENSION
 COMMON
 EQUIVALENCE
 Rest of program
 END
```

<u>Functions and Subroutines</u>

A subprogram may be designated as a FUNCTION, as mentioned earlier, or as a SUBROUTINE. The differences are:

(1) A FUNCTION has a numerical value when it is used, but a SUBROUTINE does not.

(2) A SUBROUTINE need not have any parameters at all.

With both SUBROUTINE and FUNCTION, exit should be made by means of a RETURN statement.

The parameters to a FUNCTION or SUBROUTINE may be

(1) Simple variables

(2) Array variables.

If a parameter is an array variable, it should appear in a DIMENSION statement.

For example,

```
SUBROUTINE   SUB1(A,IA, S)
DIMENSION   A(20,20), IA(100), B(30)
```

indicates that A and IA are array parameters, while S is a simple variable parameter, and B is a new array which is internal to the subroutine.

A maximum of 40 other subprograms may be called by any one subprogram.

A SUBROUTINE is used by another subprogram by using the CALL statement, e.g.,

```
CALL   SUB1(Q,KA,T)
```

which means that the array Q is used in SUB1 wherever A appears there, that KA is used for IA, and that the simple variable T is used for S . Suppose in SUB1 we have the statement S = A(IA(3),J)* S; then when CALL SUB 1(Q,KA,T) is performed the action is

```
T = Q(KA(3),J)*T
```

The variables A, IA, and S are called "formal parameters", while the variables Q, KA, and T are called "actual parameters". When a FUNCTION or SUBROUTINE is called, the actual parameters effectively replace the formal parameters.

In a statement such as S=expression, where S is a formal parameter, it is assumed that the actual parameter will be a variable also, for it would make no sense if the actual parameter were a constant or an expression. It is also assumed that the actual parameter is not the index variable of a DO loop, for it is not legal to change that index variable by means of a FUNCTION or SUBROUTINE.

A SUBROUTINE without parameters would be declared simply by giving its name, e.g.,

```
SUBROUTINE   ADJ
```

This subroutine is used by saying

```
CALL   ADJ
```

The number of parameters when calling a FUNCTION or SUBROUTINE can be less than the number of parameters in the declaration as long as the missing parameters are not used. A FUNCTION can be called by using a CALL statement but then its numerical value is lost; to get the numerical value, a FUNCTION should be embedded in an arithmetic expression. It would be wrong, however, to call a SUBROUTINE by embedding it in an expression since there is not numerical value.

NOTE: The object program for FUNCTION and SUBROUTINE assumes that simple variable parameters will be used at least 4 places within the subprogram for most efficiency. If, however, a formal parameter which is a simple variable is used only once or twice inside the subprogram, it is efficient to declare it as an array variable of length 1 and to use it in that fashion. For example, if S as a formal parameter is only used in one place such as T=S in the subprogram, it is best to write

```
              DIMENSION   S(1)
  and then    T = S(1)
```

achieving the same effect with a faster running program.

Here is an example of a SUBROUTINE which finds the roots of a quadratic equation, assuming there are two real roots:

```
          SUBROUTINE  QUADROOT(A,B,C, ROOT1,ROOT2)
          D = B**2 - 4.0*A*C
          IF (D)    1,2,2
    1     STOP
    2     E = 2.0*A; F = - B/E; D = SQRT(D)/E
          ROOT1 = F+D; ROOT2 = F-D
          END
```

A very slight increase in the speed of the running program could be made here by writing parameters as "dimensioned variables" since this subroutine is so short.

In another subprogram the statement

```
CALL    QUADROOT(1.0,B,C+H,X1,X2)
```

would set X1 and X2 to the two roots of the equation $X^2 + BX + C + H = 0$. It is important to match up arithmetic types between the actual and formal parameters. One could not get any meaningful answer if he wrote

```
CALL    QUADROOT(1,B,C+H,X1,X2)
```

To solve $X^2 + IX + J = 0$ we could write

```
CALL QUADROOT(1,0, FLOAT(I), FLOAT(J), X1, X2).
```

CHAPTER VII

OPERATING INSTRUCTIONS

Punching Fortran Statements onto Cards

Fortran programs are written on coding forms which correspond to the way the statements get punched onto cards.

If column 1 is a "C" the entire card is a comment so it is not processed by the translator except to appear on the listing of the program.

If column one is a "#" the card is a special reserved-word card described in Appendix I.

Otherwise columns 1-5 are used for statement numbers. If this is not a numbered statement, columns 1-5 are left blank; otherwise the statement number is punched anywhere in columns 1-5.

Column 6 is the "continuation column". It is usually left blank, but if it is punched it should be punched with a numeric digit. The number 0 is equivalent to blank, but any other digit indicates that this card is a continuation of the preceding statement. Columns 1-5 are ignored on continuation cards.

Column 7 is where the Fortran statement starts. On 80-column cards the statement is punched in columns 7-72; Columns 73-80 are ignored.
On 90-column cards the statement is punched in columns 7-40 and in columns 46-85. Columns 41-45 and 86-90 at the right edge of the card are ignored.

END cards are special in that information after the word END is ignored; and there should be no continuation cards after an END card.

Batch compilation of subprograms

Subprograms are of three kinds:

(1) FUNCTION, indicated by a FUNCTION declaration.

(2) SUBROUTINE, indicated by a SUBROUTINE declaration.

(3) MAIN PROGRAM, indicated by a lack of FUNCTION or SUBROUTINE declaration.

Each subprogram terminates with an END card.

Any number of subprograms may be compiled together.

The object program cards can be separated (their format is given in Appendix V), and each subprogram is an independent unit.

When running the object program, any subprograms (possibly even compiled on different days or on different machines) may be put together with the restriction that there is exactly one MAIN program, all the rest are FUNCTIONs or SUBROUTINEs. The program starts by executing the MAIN program.

There is another form of compilation, called "load-and-go", which is possible. In this form, all subprograms which are to be run are compiled together and then they are immediately loaded and the object program is executed.

Header cards

Each subprogram has associated "header cards" which give its requests for amounts of program, data, and common storage and also its requests for other subroutines used. Header cards have an identifying punch in column 6 (0 on 80-column cards while other cards have 1, blank on 90-column cards while other cards have 0). On 80-column systems header cards are ejected automatically into a special pocket after punching.

The header information is printed with each subprogram. The user will ordinarily not concern himself with this information but the meaning will be given here for reference:

First word	:	ZZZZZNNNNN name of the subprogram (MAIN* is used for a MAIN program) in MC-6.
Second word	:	POCCCC QQQQ P = priority of data storage, CCCC = amount of common storage, QQQQ = amount of data storage.
Third word	:	P10001QQQQ P = priority of program storage, QQQQ = amount of program storage.

Succeeding words: ZZZZZNNNNN names of subrottined called, in MC-6 code. Library subroutines are named FLPK*, EXPK*, EDPK*, TRIG*.

Input deck

The input decks are arranged as follows:

Batch Compiling

Compiling:

1. Translator
2. Fortran subprograms (in any order, each ends with END card)

Running:

1. Loader
2. Header cards (for each subprogram desired)
3. Library
4. Program cards (for each subprogram desired)
5. Transfer cards
6. Data (if any)

Load-and-go

1. Translator
2. Fortran subprograms (in any order, each ends with END card. One of these is the MAIN program)
3. Loader
4. Library
5. Program cards from the punch hopper-delete header cards.
6. Transfer card
7. Data (if any)

Note that header cards are not necessary in the load-and-go mode of operation. Also that it does not hurt to include header cards or program cards for programs that will not be used.

Calling sequence

To load in either the Fortran Translator or the Fortran Loader, set the console switch to "9800 suppress" and proceed as follows:

1. Set computer on one instruction
2. Key 96 0001 0025 for 80-column versions,
   96 0000 0021 for 90-column version into register A

3. Key 72 0000 000A into register C.

4. Depress CONTINUOUS, GENERAL CLEAR, and RUN.

5. Stand back.

If only one card has read in and the computer loops, you forgot to throw the switch to 9800 suppress. It is not necessary to reload the cards, you can restart without any trouble:

a. Set the computer on one instruction.
b. Set the switch to 9800 suppress.
c. Key into register C the instruction
   0B 0399 0073 for 80-column cards
   0B 0399 0122 for 90-column cards
d. Depress CONTINUOUS, GENERAL CLEAR, RUN.

If the machine stops with 1111 in the m address during loading of the decks one of the following has occurred:

a. The reader hopper is full; to continue remove the cards from the full hopper and depress GENERAL CLEAR, RUN.

b. The cards are out of sequence (the sequence number is in columns 7 thru 10). The number of the next card expected is displayed in the four low order digits of rA. Take the cards from the reject hopper, reorder them and place the deck back into the reader. Depress GENERAL CLEAR, RUN.

c. A sum check failure occured. This is probably just a reader error, take the cards from the reject hopper and place them in front of the remaining deck. Depress GENERAL CLEAR, RUN to try again.

The object programs start in location 0004 and end in location 0010.

Error stops during translation

The following stops may occur during translation (identified by the m address in register C)

1111 : Read-punch unit off normal. Fix, depress RUN.

1112 : The card which has just been read at the 2nd read station of the punch unit has either been punched improperly or has been read back improperly. This is a potentially dangerous and unrestartable situation, but if you choose to ignore it you may depress RUN. The loader will reject the card if it is actually punched improperly; and it may be possible to correct a bad card by hand, if you are familiar with the card format (Appendix V). The safest policy, however, is to run an END card followed by the last subprogram back through the translator, if doing batch compilation; else start from scratch if doing load-and-go compilation. Discard the mispunched cards.

1212 : Too many identifiers and/or constants in a subprogram. Correct program by dividing it into several smaller subprograms.

2222 : High-speed reader off normal. FIX, depress RUN.

2223 : The high-speed reader should now be off-normal; this is the normal stop on "out of cards". Insert more cards into hopper, depress GENERAL CLEAR, and RUN. Or, if this is the end of a batch compilation, take your cards out of the reader and punch and sit down, you are done.

If the 2223 stop occurs with the high-speed reader normal this indicates that somehow somebody stopped the translator program and hit GENERAL CLEAR before restarting, or else there is a card jam. Check if all cards in the reader stacker have been printed on the printer; insert in the hopper the Fortran cards starting with the first card not printed.

3333 : The high-speed printer is off normal. Fix, depress RUN.

OA2A : This stop looks like "HLT X X". An attempt is being made to do load-and-go compilation but either

1. An error has been detected in the Fortran statements so you cannot "GO".

or 2. The number of words of header information has exceeded the internal capacity, so the program should be run as if it were a batch compilation. This is the case if no error messages were printed during translation.

The translator should not stop for any other reason. If it does it represents an accumulation of errors in the Fortran statements. The user may attempt to restart by transferring control to location 4140 without depressing GENERAL CLEAR.

Error stops while loading:

2222 : Reader off normal. Fix, depress GENERAL CLEAR, RUN.

2221 : A sum check failure occurred. This is probably just a reader error, take the cards from the reject hopper and place them in front of the remaining deck. Depress GENERAL CLEAR, RUN to try again.

1001 : The cards are out of sequence (the sequence number is in columns 7 thru 10). If the problem can be solved by reloading the deck beginning with the expected sequence number do so and continue by depressing GENERAL CLEAR and RUN. (the expected sequence number is displayed in rA). Otherwise you will have to restart.

1002 : Two definitions for the same external reference entry have occurred. This is not restartable.

1003 : An entry definition line (relocation digit 6) has occurred for a routine that does not exist. This is not restartable.

1004 : Header cards out of sequence. This is not restartable.

1007 : The header cards for a necessary subroutine have been omitted. This is not restartable.

1008 : Too many subroutines have been used. Reduce the number so that there are fewer than 200.

1009 : The Fit Table does not contain a memory section long enough to fill the current request (see Appendix V).

3333 : The printer is off normal, repair, depress GENERAL CLEAR, and RUN.

Error stops while running

When the object program is running the following stops may occur:

1245 : A PAUSE statement has occurred. The contents of rA may be changed. Depress RUN to continue.

5421 : A STOP statement has occurred. Depressing RUN sends control to location 0010.

1111 : Punch off normal. Fix, depress GENERAL CLEAR, RUN.

2222 : Reader off normal. Fix, depress GENERAL CLEAR, RUN.

3333 : Printer off normal. Fix, depress GENERAL CLEAR, RUN.

In addition to the above stops the following errors may be printed on the High Speed Printer in the form ERROR nnn.

001 Exponent overflow

002 Division by zero

003 Square root of a negative number

004 Logarithm of a number less than or equal to zero

005 Sine of a number greater than 10^8

006 Integer division with a result whose magnitude is greater than 9999

007 Result of fixing a floating-point number is larger than 9999 or less than -9999.

900 Undefined format Letter.

902 Format and input (or output) list is incompatible.

906 The w field in an A format specification is bigger than 5.

907 The w field in an M output specification exceeds 10.

967 The M output specification was given on the 9000 compiler system.

968 The M input specification was given on the 9000 compiler system.

969 More than 5 columns of A input or more than 10 columns of M input were requested.

971 An "E" or a "." occurred on an I specification input.

972 An input value has too large a power of ten exponent.

Error messages during translation

Extensive checking of the Fortran program is made by the translator. If errors are found they are printed along with the source statements.

Such an error message will always appear after the line on which the error was made, but it may appear one or two lines after the erroneous line. That is, an error message need not apply to the immediately preceding line and in many cases it doesn't.

After an error has been detected, the translator attempts to keep running. Since it is then dealing with potentially bad information, however, successive error messages after the first may not be actual errors in the program. If no apparent errors are noted, fix the first one and sometimes the others will go away as if by magic.

No guarantee is made that all syntactical errors will be detected, just most of them. In fact, no doubt many users will discover that quite a few of the restrictions in this manual are not enforced and that a knowing use will allow them to achieve certain "nice" effects in their programs. To these people we say, "Do not try to run your programs with any other Fortran system, or on next year's Fortran system".

Here is a list of error printouts which might occur, and their theoretical significance:

Error	Meaning
A/I ERROR	Attempt to mix up arithmetic types, e.g. A/I
A*I ERROR	Attempt to mix up arithmetic types, e.g. A*I
A+I ERROR	Attempt to mix up arithmetic types, e.g. A+I
BAD CONSTANT	An improper constant; e.g., a floating point constant out of range. Or, perhaps a multiplication symbol was forgotten.
BAD DIMENSION	An array of too large a size has been declared, or a DIMENSION declaration has been fouled up e.g., DIMENSION A(B), C(3.8) etc.
BAD EQUIVALENCE	One of the rules for EQUIVALENCE statements has been violated, or an attempt has been made to include an EQUIVALENCE declaration in version 8002 or 9002 of the translator.

BAD LABEL	Something besides a label has occurred in label context; perhaps in columns 1-5 of the card. By label we mean a statement number.
BAD MESS	An accumulation of errors has caused the translator to get completely lost. There is no telling what was the last straw.
EXP # ERROR	An expression has occurred on the left side of an equal size, not just a variable; e.g., X+Y = Z. Cause might be an array variable which was not DIMENSIONed; or an attempt to write an arithmetic function statement (which is not allowed in this FORTRAN) such as F(X) = X+Z.
EXTRA COMMA	A comma has occurred where there was no occasion for it.
EXTRA OPERAND	Too many quantities and not enough operators to go around, e.g. "Y = X3" with a space between X and 3; or an expression which is not set equal to anything. This may also indicate a "spurious continuation card" with some strange thing in column 6 by mistake.
EXTRA RIGHT PARENTHESIS	A right parenthesis has occurred with no corresponding left parenthesis.
EXTRA SUBSCRIPT	More subscripts have appeared than were given in the DIMENSION declaration for an array variable.
F(I) ERROR	Integer expression used as parameter to a function such as SQRT.
FLOATING SUBSCRIPT	A floating-point expression has been used for a subscript.
FLOATING SUBSCRIPT OR I*A ERROR	Floating subscript or attempt to mix up arithmetic types, e.g., I*A.
I+A ERROR	Attempt to mix up arithmetic types e.g. I+A.
I**A ERROR	Attempt to mix up arithmetic types, e.g., I**A.
I/A ERROR	Attempt to mix up arithmetic types, e.g., I/A.
I'M FULL	1212 stop has occurred. See above under 1212 stop.
MISSING COMMA	You forgot a comma somewhere, e.g., in an IF statement.
MISSING LEFT PARENTHESIS	A left parenthesis has been left out e.g., after the word IF.

MISSING OPERAND	A quantity has been forgotten, e.g,, a statement "Y=" with nothing at the right. This may also indicate a "spurious continuation card" with some strange thing in column 6 by mistake.
MISSING RIGHT PARENTHESIS	An unmatched left parenthesis was used.
MISSING SUBSCRIPT	A dimensional variable used without a subscript in an illegal place.

[IF NOTHING ELSE WORKS, READ THE INSTRUCTIONS]

APPENDIX I

In addition to the features already described, a few more things have meaning in USS Fortran II.

THROUGH

The word THROUGH can be used in place of the word DO. For example,

```
THROUGH 7, V = 5.0, X-Y, 1.0
```

can be used. THROUGH is almost equivalent to DO; the only difference is a possible difference in the object program, as explained in Appendix IV.

TRACE

To aid in debugging a Fortran program, "tracing" is provided. When tracing is in operation, the result of every arithmetic statement is printed on the printer. For example, the printer listing might contain

```
Y =    5010000000        0437
```

when the statement Y = 1.0 is executed. The printout is

```
variable name  =  value in internal code      loc
```

(The "loc" is a cryptic reference to the place in the program and is the machine location of the beginning of the following statement. This can be correlated to the place in the symbolic program, if necessary, as explained in Appendix VI.)

This printout occurs for

1. Arithmetic statements
2. ASSIGN statements
3. READ statements on input.

On the 90-card system (version 9000) the name Y is not printed in alphabetic form, only in ZZZZZNNNNN code (MC-6).

APPENDIX I (continued)

Modes of Compilation

There are four independent modes which can each be "turned on" or "turned off" when compiling. These are LIST, TRACE, CORE, and CARDS.

The statement "LIST", for example, may be inserted anywhere in the Fortran program and it causes LIST mode to be turned on. Any of these four words may be preceded by the word "NO"; for example, "NO LIST" would cause LIST mode to be turned off.

TRACE mode: When trace mode is on, additional instructions are inserted in the object program to cause entry to the TRACE subroutine described above whenever an assignment occurs to a variable.

LIST mode: When list mode is on, the compiled instructions of the object program are printed as they are generated. The format is explained in Appendix VI.

CARDS mode: The object program is punched onto cards; if CARDS mode is off there is no object program punched. "NO CARDS" is used when merely checking for errors in a newly-written program.

CORE mode: Whenever a solid state II is used and there is a chance that the data storage or common storage may be located in core, CORE mode must be in effect. There are two differences in the object program when CORE mode is on:

1. The method of assigning program locations is changed to make the program run approximately 8 times as fast when data is in core, or 4 times slower when data is not in core.

2. In certain cases the object program which would work on the drum will fail when relocated in core storage, so extra instructions are added when CORE mode is in effect to make it work in either case.

APPENDIX I (continued)

When the translator is loaded into the computer it is in the following state:

<u>8001-9000-9001 versions</u>	<u>8002-9002 versions</u>
NO TRACE	NO TRACE
NO LIST	NO LIST
CARDS	CARDS
NO CORE	CORE

Using CORE and NO CORE it is possible to compile programs on one computer configuration to be run on another configuration. (Programs may be run on a step configuration or any combinations of drum or core.)

<u>Caution</u>. The modes of compilation are not reset when entering another subprogram, so if using any non-standard mode it should be reset at the end of each man's program so as not to affect others in a large batch compilation.

Another word of warning: It is completely disastrous to change from CORE to NO CORE, or vice versa in the middle of a subprogram! These should be changed only at the beginning or <u>after</u> the END card, or else end the program with

```
STOP    ;   NO CORE
END
```

instead of merely END. There is no similar problem with TRACE, LIST, or CARDS which may be turned on or off anywhere.

<u>Reserved words</u>

The following feature is actually intended only for those with an intimate knowledge of the Fortran translator, but can be used in certain cases by anyone to create a new reserved word.

A card with a "#" in column 1 is used for this purpose. Starting in column 7, give an identifier which is <u>not</u> already a reserved word. After

APPENDIX I (continued)

the identifier give a 10-digit constant which is in a special code telling the "meaning" of the new reserved word. The following cards can be used to enter new reserved word codes for standard functions to end with "F", for compatibility with some other Fortran compilers:

```
#    SQRTF    10M9841190000
#    SINF     10M9841200000
#    COSF     10M9841210000
#    TANF     10M9841220000
#    ATANF    10M9841230000
#    LOGF     10M9841240000
#    EXPF     10M9841250000
#    ABSF     10M9841260000
```

These cards should only be used at the beginning of a subprogram. They affect all succeeding subprograms.

Punching a Translator Deck

To punch a translator deck, if it is desired to modify the translator, make the modifications when it has been loaded on the drum (possibly adding a few reserved words) and read in a card like

C   USS FORTRAN II *** VERSION (number-month-date-place)

This title will appear as the first line printed on every compilation.

Transfer control manually to location 2200; the compiler will be completely punched out except for several cards of its loading routine which are unchanged at the beginning and the five cards at the end of the deck. Cards are identified by sequence numbers in columns 1-10.

APPENDIX II

Summary of Statements and Reserved Words

Statements allowed

(Brackets are used to indicate optional parts of the format.)

Arithmetic statement	:	variable = expression or variable = variable = expression, etc.
GO statement	:	GO {TO} statement-number
Assigned GO	:	GO {TO} simple-variable
Computed GO	:	GO {TO} (list of statement numbers), integer expression
Assign statement	:	ASSIGN statement-number {TO} simple-variable
IF statement	:	IF (expression) negative, zero, positive
DO statement	:	DO statement number {,} variable-start, stop {,step}
THROUGH statement	:	THROUGH statement number {,} variable = start, stop {,step}
CONTINUE statement	:	{CONTINUE}
PAUSE statement	:	PAUSE {expression}
STOP statement	:	STOP {expression}
FUNCTION declaration	:	FUNCTION name (parameter list)
SUBROUTINE declaration	:	SUBROUTINE name {(parameter list)}
CALL statement	:	CALL name {(parameter list)}
READ statement	:	READ format, input list
PUNCH statement	:	PUNCH format, output list
PRINT statement	:	PRINT format, output list
FORMAT statement	:	FORMAT (format string)
RETURN statement	:	RETURN
DIMENSION declaration	:	DIMENSION array list
COMMON declaration	:	COMMON variable list
EQUIVALENCE declaration	:	EQUIVALENCE list of equivalences (not on versions 8002 or 9002)

APPENDIX II (continued)

MODE declarations	:	{NO} TRACE
		{NO} LIST
		{NO} CORE
		{NO} CARDS
END statement	:	END

List of reserved words

ABS	DIMENSION	IF	RETURN
AND	DO	LIST	SIN
ARCTAN	END	LN	SQRT
ASSIGN	EXP	NO	STOP
CALL	EQUIVALENCE	NOT	SUBROUTINE
CARDS	FIX	OR	TAN
COMMON	FLOAT	PAUSE	THROUGH
CONTINUE	FORMAT	PRINT	TO
CORE	FUNCTION	PUNCH	TRACE
COS	GO	READ	

APPENDIX III

ARRAY SUBSCRIPTING

Arrays are stored internally so that the left-most subscript varies most rapidly, then the next from the left and so on. For example, consider the 8 locations allocated for DIMENSION A(2,2,2)

```
Loc 1    A(1,1,1)
Loc 2    A(2,1,1)
Loc 3    A(1,2,1)
Loc 4    A(2,2,1)
Loc 5    A(1,1,2)
Loc 6    A(2,1,2)
Loc 7    A(1,2,2)
Loc 8    A(2,2,2) .
```

Suppose we have, in general,

```
DIMENSION  A(w, x, y, z)
```

the address of A(I, J, K, L) would be

```
Address of  A(1,1,1,1)
   +    (I-1)
   +    w*(I-1)
   +    w*x*(K-1)
   +    w*x*y*(L-1).
```

It is possible to use an array with less dimensions than appear in the DIMENSION declaration. In this case, additional subscripts of 1 are added automatically. The feature is often convenient for dealing with a whole array at once.

For example, to set a 20 x 20 matrix to zero, we could write

```
      DO  5   I = 1,20
      DO  5   J = 1,20
  5   A(I,J) = 0
```

but it would be much easier to write

```
      DO  5   I = 1,400
5         A(I) = 0
```

(and the latter version is also much faster running.)

In the main body of the text the restriction was made that subscripts must be greater than zero and at most the amount appearing on the DIMENSION statement. This rule is, more precisely:

(1) Each subscript position must have a value greater than zero.

(2) The final computed address must lie within the storage allocated for this array.

APPENDIX IV

DO loops

"DO" loops and "DONT" loop

In the object program, DO loops are divided into "DO" and "DONT" loops, meaning "DO keep the value of the index variable in index register 1" or "DONT keep the value of the index variable in an index register".

When a DO statement occurs, the loop becomes a "DO" loop, unless one of the following occurs, in which case it is a "DONT" loop:

1. This loop is specified by the word THROUGH instead of DO.

or 2. This loop is nested inside of a "DO" loop.

or 3. The index variable is floating point.

or 4. The start and step values are not both constants.

A "DO" loop causes a speed increase of approximately a factor of 3 whenever the index variable occurs in a simple way in the left-most subscript of an array, since the array is addressed in a single instruction.

Since "DO" loops are not nested - in a nest of DO's only one index variable is kept in an index register - there can be gains in efficiency depending on which index variable is kept in an index register. Sometimes this is desired in the inner loop, sometimes in the outer loop.

For example,

```
          DO   5  I = 1,10
          DO   5  J = 1,10
     5    A(I,J) =  B(I)
```

it is better to put I in an index register, because it occurs in the left part of subscripts and J doesn't.

The compiler's rule, given above, will often choose the outerloop as the "DO" loop. In order to get the compiler to choose an inner loop, in those cases where it is desired, use the word THROUGH for the outer loops. For example, if the last statement in the above had been

```
     5    A(J,I) = B(J)
```

APPENDIX IV (continued)

one could write

```
        THROUGH  5  I = 1,10
        DO     5    J = 1,10
    5   A(J,I)       = B(J) .
```

The following instructions were given for DO statements in the main part of this manual:

(1) Do not GO TO from outside into the range of a DO statement

(2) Do not assume any value for the index variable at the close of a DO statement

(3) Do not change the value of the index variable by using a function or subroutine call

(4) Start stop, and step must not involve the index. Start stop must be printed.

Actually these restrictions apply only to "DO" loops and not to "THROUGH" loops. In a THROUGH loop the index variable has the last used value at exit.

APPENDIX VI

The Object Program Listing

The translator allocates storage for the main program in an interlaced fashion; in a band the locations used are

000
057
114
171
028
.
.
.
143
200
257, etc.

when in NO CORE mode. With CORE mode the increment is 7 rather than 57.

Thus, the drum is like a sequential storage, and if the location of an instruction is given one can determine where it appears in the program.

The Program Listing

With LIST mode on, the interested user can see if his program gets translated correctly.

The format is

RRR S AAAA OPMMMMCCCC Comment

The RRR specify, for A, M, or C respectively, the type of relocation. S is the sign

R = 0 absolute
R = 1 date relative
R = 2 program relative
R = 3 common relative
R = 5 external reference (see table following)
R = 6 request another subprogram.

APPENDIX VI (continued)

The comment part is filled in with names of variables in order to help understand the M address of the instruction. This field might also say "CONST" or "TEMP" referring generically to a constant or temporary storage location, respectivelu. The comment "(CORE ONLY)" comes out only when the compiler has had to add additional instructions to properly handle relocation in core. If "(CORE ONLY)" does not appear on the listing, the program could have been compiled with NO CORE and still be relocated in core.

When the comment is all asterisks, the line is an out-of-sequence line. In this case the program listing would appear more logical perhaps if the line <u>following</u> a series of asterisk lines is brought ahead of all the asterisk lines.

There is often a one-card delay between the card images as printed and the beginning of the instructions for those statements.

On the 9000 version the comment is not given in alphabetic form, but rather in the ZZZZZNNNNN MC-6 code.

At the right of the card image the <u>program-relative</u> approximate location of the current object coding is listed for later reference. This can be used together with the interlace tables to find locations in a program.

APPENDIX V

The Fortran Loader

Card Formats

The general card formats for the operational decks in the USS Fortran systems are as follows:

Information	90 Column Cards	80 Column Cards
Identification	1 thru 5	1 thru 5
Card Type	6	6
Card Number	7 thru 10	7 thru 10
Data Words	11 thru 40 46 thru 75	11 thru 70
Check Sum	76 thru 85	71 thru 80

Cards are punched untranslated in the 90 systems, while they are punched translated in the 80 systems with the following undigit conventions.

Undigit	Card Character	Punch Combination
A	-	11
B	blank	blank
C	)	1-4-8
F	(	3-5-8
G	;	4-5-8
H	'	4-6-8

The check sum is generated by adding up the numeric part of the first seven words on the card from left to right.

APPENDIX V (Continued)

The Loader Functioning

The loader program is divided into two sections, the body loader and the allocator. The body loader operates from locations 0400 to 0899 and the allocator from locations 2000 to 2399 and 4000 to 4199. The rest of the 2600 word Step configuration memory is used for tables.

The loader program is the first deck through the reader. When it is loaded, the allocator portion assumes control and reads all of the Header Cards and retains them in memory. If the "Load and Go" option was used the Header Card information is left in memory by the compiler and any Header Cards from the input deck are added to it. Header Cards for the same program must be in sequence, but there are no other sequence rules governing Header Cards. In fact, Header Cards for unwanted subroutines can be included without harm.

The allocator begins by repeatly searching the Header Card file for information about the main program MAIN*. It determines which subroutines MAIN* requires and scans for information about these, and also about any subroutines these subroutines may require. Scanning continues until all the header information about all the necessary subroutines has been collected, or until the machine determines that Header Cards for an essential routine have not been loaded. In the latter case an unrestartable stop occurs.

Header Cards

Header cards are identified by a 0 in column 6. All Header cards must appear first. There are six entries per card and as many cards as required. Entries of zero are ignored. The Headers for a new subroutine are detected by 0000 in the Card Number.

The first three entries are required and have the format as follows.

A. Entry 1 - Routine Name

zzzzznnnnn

where zzzzz is the zone information of the routine's MC6 coded name
nnnnn is the numeric information of the routine's MC6 coded name

The name of the main program must be MAIN*

APPENDIX V (Continued)

B. Entry 2 - Data Specification

p 0 cccc dddd

where p	is the priority of the data space for the routine = 0 for floating package only
cccc	is the amount of COMMON required by the routine
dddd	is the number of local data locations that must be available to the routine.

C. Entry 3 - Program Specifications

p' 1 eeee iiii

where p'	is the priority desired for the program instruction area = 0 for the floating point package only
eeee	is the number of entry points to this routine
iiii	is the amount of storage required for program

D. Additional Entries

All additional entries are of the form

zzzzz nnnnn

and name the other routines required for operation of the subroutine.

All of the body decks for the library and compiler output will then be loaded or passed over as required. When the transfer deck is encountered it will be passed over also and execution will commence at location 0004.

The following is a tabulation of the additional card formats:

Body Cards

Each body card contains three two-word groups defined as follows:

WORD 1	a m c 0 s 0 AAAA	
WORD 2	op MMMMCCCC	
where	a	is the relocation code (R) applicable to the address
	m	is the relocation code applicable to the m-address of the word to be loaded
	c	is the relocation code for the c-address of the word to be loaded
	s	is the sign of the word to be loaded
	AAAA	is the address (unrelocated) of the instruction
	opMMMMCCCC	is the word to be loaded into nominal AAAA with sign s, and after relocation according to m and c

APPENDIX V (Continued)

A. Instruction Entries

For computer instruction words, the values of relocation, R, are as follows. (The 4-bit is ignored in a, m, c.)

R	EXPLANATION
0	The nominal address is to be used as the true address
1	The nominal address refers to the data area for this routine, and will be relocated by the beginning location of own local data.
2	The nominal address refers to the program area for this routine, and will be relocated by the first location assigned to the routine.
3	The nominal address references COMMON and will be relocated by the base of the COMMON area.
5	The nominal address is an external reference, xxyy, and will be replaced by the true address as required.
8	This code is reserved for the floating-point package, and for that subroutine complex is treated as program relative.

Note: Codes 3 and 5 should never be used in a, especially 5.

B. External Reference Entry

WORD 1 600000XX00
WORD 2 zzzzznnnnn
where XX is the number used by this routine to reference the subroutine named (in MC6) in WORD 2

C. Entry Definition

WORD 1 70c0000000
WORD 2 0000yyCCCC
where c is the relocation code for CCCC in WORD 2, and may not be 5
yy is a number of an entry point into this routine
is the nominal address equivalent to entry number yy

For the Data cards of each routine, the first card must be numbered 0000, with cards in strict sequence. The first data word pair is treated as follows:

WORD 1 ignored
WORD 2 zzzzznnnnn

Where WORD 2 contains the MC6 name of the routine being loaded. It is not required by the program being loaded, it will be ignored; the cards are passed through the reader without processing.

Deck Arrangement

Decks are loaded in the following sequence:

1. Loader
2. Header Cards for programmers routines
3. Library
4. Body Cards for programmers routines, with each routine beginning with card number 0000
5. Body Transfer Deck
6. Data for the program

Loader Transfer Card

80 Column Cards	90 Column Cards	Contents
1 thru 10	1 thru 10	00 0002 0000
11 thru 70	11 thru 40	all zeroes
	46 thru 75	
71 thru 80	76 thru 85	00 0002 0000

Body Transfer Deck

This deck consists of 6 cards. Card i $(0 \leq i \leq 5)$ looks like

80 Column Cards	90 Column Cards	Contents
1 thru 10	1 thru 10	00 0003 000i
11 thru 70	11 thru 40	
	46 thru 75	all zeroes
71 thru 80	76 thru 85	00 0003 000i

All of these cards fall into the standard card format described earlier.

During the scanning of the Header information, the items which corresponded to entries 2 and 3 of the Header Card are removed from the Header Card table and placed in a separate table called the Request Table. This table controls memory allocation.

In the object program there are three types of memory: Program, Data and Common.

a) Program. This class of memory contains object instructions compiled by Fortran and Library programs. A separate memory area is allocated for each program.

APPENDIX V (Continued)

b) Data. This class of memory contains values of variables, arrays, format statements, subroutine parameters and other things compiled by Fortran. One Data area is allocated for each program.

c) Common. The Common area contains all quantities named in the Common declaration of each program. Only one Common area is allocated.

Because it contains a copy of the loader during loading, Common will always be at least 600 locations and probably more. Programs using large data stores should declare as much of the data as possible to be Common. This may save a large number of memory cells.

The allocation process is carried out as follows:

a) The Request Table is sorted. Since the leading digit of each entry is the priority digit, this controls the sort.

b) The requests in the Request Table are allocated beginning with priority zero (priority zero is reserved for the floating point arithmetic packages).

Located in locations 4000 to 4009 is a table called the Fit Table. It contains entries in the format

00 nnnn bbbb

where nnnn is the number of words available and bbbb is the first address.

Each word in the table specifies a block of available memory. An ending sentenial of -50 0000 0000 concludes the table.

Memory is allocated by searching the table, beginning at location 4000, until a specification is found which is large enough to satisfy the request. The memory specification is then updated and the next request is allocated. The request for Common storage must be and is allocated first.

The Fit Table for the Step memory configuration looks like

```
4000     00 0200 4000
4001     00 2000 0400
4002    -50 0000 0000
```

There are a few restrictions governing the entries in the Fit Table.

a) The floating point routines will be placed in upper core if and only if the first entry of the Fit Table specifies a block in core.

APPENDIX V (Continued)

b) Locations B000 to B029 should not be made available.

c) Locations 0000 to 0399 should not be made available.

d) Common should not begin at any of the locations 0401 to 0899. It should begin at 0400 or above 0900. Remember Common is allocated first and contains more than 600 locations.

e) All base addresses should be multiples of 50. <u>All allocation is done in multiples of 50</u>. Pairs of addresses coded to handle c+1 conditions are not permitted on the level pairs

0049, 0050
0099, 0100
0149, 0150
0199, 0000

Their levels, and hence their operation may change during relocation, creating mysterious new bugs if this rule is violated. For the same reason I-O interlaces are verboten in hand-coded subroutines. Hand-coded subroutines may use band 0000 for a card interlace and band 0200 for a tape interlace if necessary.

Every attempt should be made to favorably place data either in high-speed bands or in core.

The Fit Table is loaded from the seventh and sixth last cards of the loader as follows:

Seventh last card

Columns	Location
21 thru 30	4000
31 thru 40	4001
41 thru 50	4002
51 thru 60	4003
61 thru 70	4004

Sixth last card

Columns	Location
21 thru 30	4005
31 thru 40	4006
41 thru 50	4007
51 thru 60	4008
61 thru 70	4009

You won't be able to forget the check sum (which is in columns 71 thru 80).

APPENDIX V (Continued)

At the end of the allocation phase a page is printed giving the memory allocation which was chosen. Common base address is printed on the top line. Three columns are printed. The first column contains the name of the program (on the 9000 compiler system this appears in zzzznnnn MC6 code). The second column contains the Program base and the third the Data area base. This printout is invaluable as an aid to debugging for the skilled machine language coder.

After the allocation printout, the loader begins loading the body cards. The first deck in should be a relocatable copy of the loader which will load into the lower locations of the Common area. (This is the reason for the 0401 thru 0899 rule for Fit Table entries. The loader must overlay itself either exactly or not at all). When a loader transfer card appears, control is transferred to the relocated loader. If common goes into a higher speed memory, loading will be accelerated.

Communication between programs

It is sometimes desirable to communicate between programs by leaving data sitting in the Comma area. This is possible in USS Fortran under the following restrictions.

1) Common for both programs must allocate to the same place. One may have to fudge with the Fit Table to accomplish this.

2) The data to be transmitted must be at the end of Common. The first 0600 plus locations are overlayed by the loader. It is difficult to know how big plus is, but the formal rules for calculating the amount are

    606 location, plus
    3 for each package or subroutine or function used, plus the total number of entry points in all the routines used.

    800 will almost always be adequete and 1000 certainly will be.

3) The part of Common chosen for communication should also avoid locations 0000 to 2399 and 4000 to 4199 since these are used also by the loader and allocator.

Card Formats for the Compiler and Loader Decks

Each user may have need to modify the compiler or loader for his individual application, hence the card formats and the deck construction are given here.

APPENDIX V (Continued)

The compiler and loader decks have the following construction

1) The Bootstrap Loader (twenty four cards). These cards are recognized by an identification (columns 1 thru 10) of the form:

$$\text{*BOOT} \frac{L}{T} \text{nn}$$

where nn is a sub-sequence number starting at one and either a L or a T appears depending upon whether is bootstrap loader is for the loader or compiler, respectively.

2) The compiler or loader deck. Cards in this deck have the format:

80 Column Cards	90 Column Cards	Information
1 thru 10	1 thru 10	Identification
11 thru 70	11 thru 40 46 thru 75	Instructions
71 thru 80	76 thru 85	Check Sum

The identification is of the form

*LOADniiii or *TRANniiii

where n is a type number and iiii is the sequence number. If n is 4 the following instructions are loaded into locations rB_1 thru rB_1+5; if n is 9 then columns 11 thru 20 contain an instruction of the form OB xxxx 000C, which is executed and causes the following five instructions to be loaded into locations rB_1 thru rB_1+4.

3) Five trailer cards which transfer control to either the compiler or the loader. These cards have the formats

Card 1

80/90 Column Cards	Information
1 thru 10	*LOAD9ffff, for the loader or *TRAN9ffff, for the compiler
11 thru 20	96 0001 0130, for the 80 systems or 96 0000 0031, for the 90 systems.

APPENDIX V (Continued)

Cards 2 thru 4

80/90 Column Cards	Information
11 thru 20	96 0001 0130, for 80 systems or 96 0000 0031, for 90 systems

Card 5

11 thru 20	00 xxxx xxxx

where xxxx is the starting location for the compiler or loader. (Unfortunately this will vary with time as the compiler or loader is re-assembled).

Card Types

Card formats have all been given in appropriate places in the text. To summarize the card types given in column 6, we have

Column 6	Card Type
0	Header Card
1	Body Card
2	Loader Transfer Card
3	Transfer Card
4	Loader or Compiler Deck Card
9	Loader or Compiler Deck Card

APPENDIX VII

Library Glossary and Subroutine and Function Conventions

The following information defines those subroutine entrance and exit conditions required of elements in the Fortran object program. The use of non-Fortran produced subroutines is permitted by the system as long as the compatible interface is achieved. Subroutines externally prepared must be provided according to the requirements of the Fortran Loader.

Intrinsic Functions

The intrinsic functions are divided into two groups: those which are generated in the compiler and those hardware extensions (e.g. floating-point arithmetic) which are obtained through subroutines. The elements of this latter category are used as follows:

A. Entrance Requirements
Arguments (not over two) must be provided in registers A and L. The following notation applies to the arguments:

I	is the fixed point argument required in rA,
J	is the fixed point argument required in rL,
A	is the floating-point argument required in rA,
B	is the floating-point argument required in rL.
X, Y	are the arguments in rA and rL with arbitrary type

B. Entrance:

LIR 3 return name

where	return	is the location at which control is restored on completion
	name	is an entry into one of the three intrinsic packages

C. Exit Conditions:

1. The result will appear in both rA and rL.
2. rX will contain nothing of value.
3. rB and rB_2 will be destroyed in most cases. rB_1, if used, will be resotred to its entry value.

D. Subroutine Packages
The standard packages provided with Fortran II are (1) FLPK*, floating-point, (2) EXPK*, exponentiation, roots, and logarithms, and (3) TRIG*, trigonometric functions. Reference to any entry in a package causes the entire package to be incorporated in the program.

APPENDIX VII (Continued)

External Functions

Routines defined by FUNCTION and SUBPROGRAM compilations are called by the following sequence:

A. Entrance Requirements:

1. A parameter list of the n arguments, if any, must begin at some location p, as follows:

p	return instruction		
p+1	00	loc_1	0000
p+2	00	loc_2	0000
.			
.			
.			
p+i	00	loc_i	0000
.			
.			
.			
p+n	00	loc_n	0000

where return instruction will be executed on completion

loc_i is the first location of the i-th actual parameter

2. rA, rX, rL are not used.

B. Entrance:

LIR3 p name

where p is the first location of the parameter list
name is the entry to the subprogram.

For any CALL or function designation, Fortran generates references to entry 00 which is understood to correspond to the name of the routine.

Error Subroutine

A single routine, ERR*, is included in all object programs, The routine may be entered by any external function to record the presence of an error condition. The user may program his own error procedure as required by installation operating conventions.

APPENDIX VII (Continued)

A. Entrance Requirements

1. rB_2 contains an error number for identifying the type of error.

2. rB_3 specifies the return location.

B. Entrance:

```
LIR2    code
LIR3    retrn    ERR*
```

C. Exit Conditions:

The particular action will be noted and return to retrn will occur. The contents of fA, rL, rX and rB, will <u>not</u> be saved.

APPENDIX VII (Continued)

Floating-Point Package

Subprogram Name: FLPK*

Entry number	Name	Explanation
0	MUL*	I*J
1	NMUL*	-(I*J)
2	DIV*	I/J
3	NDIV*	-(I/J)
4	RDIV*	J/I
5	NROV*	-(J/I)
6	FADD*	A+B
7	RFSB*	B-A
8	FSUB*	A-B
9	FMUL*	A*B
10	NFML*	-A*B
11	FDIV*	A/B
12	NFDV*	-A/B
13	RFDV*	B/A
14	NRFD*	-B/A
15	COMP*	-X (Boolean NOT)
16	FLT*	FLOAT(I)
17	VFLT*	-FLOAT(I)
18	LFLT*	FLOAT(J)
19	NLFT*	-FLOAT(J)
20	FIX*	IFIX(A)
21	NFIX*	-IFIX(A)
22	LFIX*	IFIX(B)
23	NLFX*	-IFIX(B)
24	FXSQ*	I**2
25	FLSQ*	A**2
27	PAUSE	
28	STOP	

APPENDIX VII (Continued)

Exponentiation Package

Subprogram Name: EXPK*

Entry Number	Name	Explanation
0	SQRT*	A**.5
1	NSQR*	(-A)**.5
2	NEXP*	e**-A
3	EXP*	e**A
4	NLN*	ln(-A)
5	LN*	ln(A)
6	PLL*	A**B
7	PLL2*	(-A)**B
8	PLL3*	A**(-B)
9	PLL4*	(-A)**(-B)
10	RPLL*	B**A
11	PLL6*	(-B)**A
12	PLL7*	B**(-A)
13	PLL8*	(-B)**(-A)
14	PLX*	A**J
15	PLX2*	(-A)**J
16	PLX3*	A**(-J)
17	PLX4*	(-A)**(-J)
18	RPLX*	B**I
19	PLX6*	(-B)**I
20	PLX7*	B**(-I)
21	PLX8*	(-B)**(-I)
22	PXX*	I**J
23	PXX2*	(-I)**J
24	PXX3*	I**(-J)
25	PXX4*	(-I)**(-J)
26	RPLX*	J**I
27	PLX6*	(-J)**I
28	PLX7*	J**(-I)
29	PLX8*	(-J)**(-I)

APPENDIX VII (Continued)

Trigonometric Routines

Subprogram Number	Name	Explaination
0	SIN*	sin(A)
1	COS*	cos(A)
2	TAN*	tan(A)
3	ATAN*	arctan(A)

APPENDIX VIII

SAMPLE FORTRAN PROGRAM

INPUT FORTRAN DECK

```
 EVALUATE SUM OF RECIPROCALS
 DIMENSION A(100)
 LIST
 READ 5,A
 SUM#0.0
 DO 1, I#1,100
 SUM#SUM+1.0/A(I)
 PRINT 2, A, SUM
 STOP
 FORMAT (10F8.2)
 FORMAT (19HTHE 100 NUMBERS ARE, 10(2/10F10.2), 2/,
1         31HTHE SUM OF THEIR RECIPROCALS IS, E30.8, 43/)
 END
```

LIST OF OBJECT PROGRAM - Page 1

```
DIMENSION A(100)                                                        0000

LIST                                                                    0000

READ 5,A                                                                0000
            600 0 0400        112225472        **********
            120 0 0002        0000570000       **********
            150 0 0001        0004050000       **********
            000 0 0000        0000000000        00 -

SUM=0.0                                                                 0057
            015 1 0004        0 00000400
            120 0 0000        0001140000
            225 1 0171        0 00280401       EDPK*
            212 0 0028        2501030085        *I*
            220 0 0085        700142000-
            212 0 0142        5000020199           A
            202 0 0114        0700010056
            202 0 0199        0700010113
            100 0 0104        0001000000       **********
            212 0 0113        7001030170        *I*
            212 0 0170        3001040084       CONST
            222 0 0084        8700270056
            212 0 0056        6001030171        *I*

DO 1, I=1,100                                                           0084
            225 1 0027        0 01410404       EDPK*
            222 0 0141        2601980198

SUM=SUM+1.0/A(I)                                                        0198
            212 0 0198        6001050055         SUM
            202 0 0055        0 00010112
            202 0 0169        0100010026
            212 0 0026        3001040140       CONST
            222 0 0140        8700830112

PRINT 2, A, SUM                                                         0197
            100 0 0106        5010000000       **********
            220 0 0112        0001970000
            212 0 0197        2501060054       CONST
            600 0 0100        122226372        **********
            212 0 0054        3400020111           A
            225 1 0111        0 01680111
            212 0 0168        5001050025         SUM
            225 1 0025        0 00820106       FLPK*
            120 0 0109        0001390000       **********
```

LIST OF OBJECT PROGRAM - Page 2

```
             120 0 0107       0001960000
             212 0 0053       2501030110      *I*
             220 0 0110       700167000-
             212 0 0167       2500020024        A
             225 1 0024       0 00810401     EDPK*
             202 0 0196       0700010138
             202 0 0031       0700010195
             212 0 0195       7001030052      *I*
             212 0 0052       3001040166     CONSI
             222 0 0166       8701090138

   STOP                                                          0166
             212 0 0138       6001030053      *I*
             212 0 0109       2501050023      SUM
             225 1 0023       0 00800401     EDPK*

   FORMAT (10F8.2)                                               0080
             225 1 0080       0 01370404     EDPK*
             222 0 0137       2601940194
             225 1 0194       0 00570128     FLPK*
             100 0 0110       0000000000     **********
             100 0 0111       0000000000     **********
             100 0 0112       0501000802     **********
             100 0 0113       0101000002     **********

   FORMAT (19HTHE 100 NUMBERS ARE, 10(2/10F10.2), 2/,            0194
             100 0 0114       9900000000     **********
             212 0 0057       0201100139
             100 0 0115       0000000000     **********
             100 0 0116       0000000000     **********
             100 0 0117       0801900019     **********
             100 0 0118       31100385 1     **********
             100 0 0119       0002300 54     **********
             100 0 0120       2112342592     **********
             100 0 0121       01210 1950     **********
             100 0 0122       0001000010     **********
             100 0 0123       0000000000     **********
             100 0 0124       1000000200     **********
             100 0 0125       0501001002     **********
             100 0 0126       0101000002     **********
             100 0 0127       1000000200     **********

 1        31HTHE SUM OF THEIR RECIPROCALS IS, E30.8, 43/)        0194
             100 0 0128       0803100031     **********
             100 0 0129       31103385 2     **********
             100 0 0130       3202144 66     **********
             100 0 0131       03111 3859     **********
```

LIST OF OBJECT PROGRAM - Page 3

```
100 0 0132          202119 953          **********
100 0 0133          1222197963          **********
100 0 0134          12301132 9          **********
100 0 0135          3000020000          **********
100 0 0136          0400003008          **********
100 0 0137          1000004300          **********
100 0 0138          0100000000          **********
                                                           0194
100 0 0139          9900000000          **********
210 0 0139          0201150010
                    211224195           (HEADERS)
                    4000000140          (HEADERS)
                    7100010200          (HEADERS)
                    112225472           (HEADERS)
                    122226372           (HEADERS)
```

INPUT DATA TO OBJECT PROGRAM

1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0
11.0	12.0	13.0	14.0	15.0	16.0	17.0	18.0	19.0	20.0
21.0	22.0	23.0	24.0	25.0	26.0	27.0	28.0	29.0	30.0
31.0	32.0	33.0	34.0	35.0	36.0	37.0	38.0	39.0	40.0
41.0	42.0	43.0	44.0	45.0	46.0	47.0	48.0	49.0	50.0
51.0	52.0	53.0	54.0	55.0	56.0	57.0	58.0	59.0	60.0
61.0	62.0	63.0	64.0	65.0	66.0	67.0	68.0	69.0	70.0
71.0	72.0	73.0	74.0	75.0	76.0	77.0	78.0	79.0	80.0
81.0	82.0	83.0	84.0	85.0	86.0	87.0	88.0	89.0	90.0
91.0	92.0	93.0	94.0	95.0	96.0	97.0	98.0	99.0	100.0

MEMORY ALLOCATION TABLE PRODUCED BY FORTRAN LOADER

```
MEMORY ALLOCATION          COMMON          4000
        PROGRAM        PROG       DATA
        MAIN*          1700       4700
        EDPK*           800       4850
        FLPK*           400       4700
        ERR*           4850       4850
        IOPK*          1900       4850
```

PRINTED LISTING PRODUCED BY OBJECT PROGRAM

```
THE 100 NUMBERS ARE
    1.00     2.00     3.00     4.00     5.00     6.00     7.00     8.00     9.00    10.00
   11.00    12.00    13.00    14.00    15.00    16.00    17.00    18.00    19.00    20.00
   21.00    22.00    23.00    24.00    25.00    26.00    27.00    28.00    29.00    30.00
   31.00    32.00    33.00    34.00    35.00    36.00    37.00    38.00    39.00    40.00
   41.00    42.00    43.00    44.00    45.00    46.00    47.00    48.00    49.00    50.00
   51.00    52.00    53.00    54.00    55.00    56.00    57.00    58.00    59.00    60.00
   61.00    62.00    63.00    64.00    65.00    66.00    67.00    68.00    69.00    70.00
   71.00    72.00    73.00    74.00    75.00    76.00    77.00    78.00    79.00    80.00
   81.00    82.00    83.00    84.00    85.00    86.00    87.00    88.00    89.00    90.00
   91.00    92.00    93.00    94.00    95.00    96.00    97.00    98.00    99.00   100.00
THE SUM OF THEIR RECIPROCALS IS          .51873774E 01
```