Reference Manual
IBM 7030 Data Processing System
FORTRAN IV

© 1963 by International Business Machines Corporation
Chapter 1. General Definitions

A FORTRAN source program consists of a sequence of FORTRAN statements.

Each statement is a string of characters grouped into records of 72 characters. If a statement is too long to fit into a single 72-character record, it may be continued on up to nine successive records. The first through fifth characters of a continuation record should not be used, and the sixth character of a continuation record must contain a non-blank, non-zero character. A record which is not a continuation record must have a zero or blank sixth character.

The order of the statements is governed solely by the order of records.

A number, less than six digits long, may be used to label any statement. This would appear among the first five characters of the first 72-character record.

If the first character is a "C" the record is treated as a comment and does not affect the object program in any way.

The seventh through 72nd characters of the record contain the statement proper.

Blanks are simply ignored by FORTRAN and may be used freely to improve the readability of the listings. (An exception is that blanks within an alphanumeric field are significant.)

Chapter 2. Constants, Variables, Subscripts, and Expressions

As required of any programming language, FORTRAN provides a means of expressing numerical constants and variable quantities. In addition, a subscript notation is provided for expressing one-, two-, or three-dimensional arrays of variables.

Constants

Three types of constants are permissible in the FORTRAN source program language: fixed point (restricted to integers), floating point (characterized by being written with a decimal point), and logical (one of the values, .TRUE. or .FALSE.).

Fixed Point Constants

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to 11 decimal digits. A preceding + or - sign is optional. The magnitude or absolute value of the constant must be less than 2^18.</td>
<td>3, +1, -28987, +31415928535</td>
</tr>
</tbody>
</table>

Where a fixed point constant is used for the value of a subscript, it is treated modulo 2^18.

Floating Point Constant

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 15 decimal digits, with a decimal point at the beginning, at the end, or between two digits. A preceding + or - sign is optional. The magnitude of the constant must be between 10^-38 and 10^38. A decimal exponent preceded by an E may follow a floating point constant. A + or - sign between E and the exponent is optional.</td>
<td>5.0, 5.0E3 (meaning 5.0 × 10^3), 5.0E+3 (meaning 5.0 × 10^3), 5.0E-7 (meaning 5.0 × 10^-7)</td>
</tr>
</tbody>
</table>

Logical Constants

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>There are only two forms of logical constants: .TRUE. and .FALSE.</td>
<td>.TRUE., .FALSE.</td>
</tr>
</tbody>
</table>

Variables

Three types of variables are permissible: fixed point, floating point, and logical. References to variables are made in the FORTRAN source language by symbolic names consisting of alphabetic and numeric characters.

Names

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 to 6 alphabetic or numeric characters, the first of which is alphabetic.</td>
<td>R, M23A, SOBNO</td>
</tr>
</tbody>
</table>

Fixed Point Variables

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>A variable name starting with I, J, K, L, M, N if not declared otherwise, or any variable name declared INTEGER.</td>
<td>IIII4, M23A, JOBNO</td>
</tr>
</tbody>
</table>

Floating Point Variables

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>A variable name starting with any letter but I-N, if not declared otherwise by a type declaration, or any variable name declared REAL.</td>
<td>ABC, B9, DELTA, R</td>
</tr>
</tbody>
</table>
Logical Variables

Rules for Constructing Normal Arithmetic Expressions

1. A normal arithmetic expression may be either fixed point or floating point, but must not be a mixed expression. This does not mean that a floating-point quantity cannot appear in a fixed-point expression, but rather that a quantity of one type can appear in an expression of another type only in certain ways.

a. A floating-point quantity can appear in a fixed-point expression only as an argument of a function.

b. A fixed-point quantity can appear in a floating-point expression only as an argument of a function, or as a subscript, or as an exponent.

2. Constants, variables, and subscripted variables are also expressions of the same type as the constant or variable name. For example, the fixed point variable J53 is a fixed point expression.

3. Functions are expressions of the same type as the function name, provided that the arguments of the function are of the types assumed in the definition of the function. For example, if SOMEF (A,B) is a function with a floating point name, then SOMEF (C, D) is a floating point expression if C and D are of the same types as A and B, respectively.

4. Exponentiation of an expression does not affect the type of the expression; however, a fixed point expression may not be given a floating point exponent. Note: The expression A**B**C is interpreted as ((A**B)**C).

5. Preceding an expression by a + or − does not affect the type of the expression produced. For example, E, +E, and −E are all expressions of the same type.

6. Enclosing an expression in parentheses does not affect the type of the expression. For example, A, (A), (A), and ((A)) are all expressions of the same type.

7. Expressions may be connected by operators to form more complex expressions provided:

a. No two operators appear in sequence. For example, A + −B is illegal but A + (−B) is legal.

b. Items so connected are all of the same type.

Hierarchy of Operations

When the hierarchy of operations in an expression is not explicitly specified by the use of parentheses, it is understood by FORTRAN to be in the following order (from innermost operations to outermost):

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>−</td>
<td>Unary minus</td>
</tr>
<tr>
<td>* and /</td>
<td>Multiplication and Division</td>
</tr>
<tr>
<td>+ and −</td>
<td>Addition and Subtraction</td>
</tr>
</tbody>
</table>

Subscripts

A variable can be made to represent any element of a one-, two-, or three-dimensional array of quantities by appending one, two, or three subscripts to it, respectively. The variable is then a subscripted variable. These subscripts are fixed-point quantities whose values determine the member of the array to which reference is made. The variable in a subscript must not itself be subscripted.

Subscripted Variables

<table>
<thead>
<tr>
<th>General Form</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>A variable followed by parentheses enclosing one, two or three subscripts which are separated by commas.</td>
<td>A(1), K(3), BETA(5*J-2,K+2,L)</td>
</tr>
</tbody>
</table>

Each variable which appears in subscripted form must have the size of its array (i.e., the maximum values which its subscripts can attain) specified in a DIMENSION or COMMON statement preceding the first appearance of the variable in the source program.

Expressions

A FORTRAN expression is any sequence of constants, variables (subscripted or not subscripted), and functions (see Chapter 6) separated by operation symbols, commas, and parentheses so as to form a meaningful mathematical or logical expression.

The arithmetic operation symbols are:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>−</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
</tr>
</tbody>
</table>
For example, the expression
\[ G = -A + B / (C + D ** E) \]
will be taken to mean
\[ G = ( (((-A) + (B / C)) + D) ** E) \]

**Ordering Within a Hierarchy**

Parentheses that have been omitted from a sequence of consecutive multiplications and divisions (or consecutive additions and subtractions) will be understood to be grouped from the left. Thus, if \( \cdot \) represents either * or / (or either + or −), then
\[ A \cdot B \cdot C \cdot D \cdot E \]
will be taken by FORTRAN to mean
\[ (((A \cdot B) \cdot C \cdot D) \cdot E) \]

**Rules for Constructing Logical Expressions**

1. Quantities which are not logical variables or constants may only appear in logical expressions in the following ways:
   a. As arguments to logical functions.
   b. As parts of relations.
2. A relation is a logical expression of the form:
   
   - E.GT.F
   - E.GE.F
   - E.LT.F
   - E.LE.F
   - E.EQ.F
   - E.NE.F

   (where E, F are arithmetic expressions of the same type). The operations mean:
   
   - E.GT.F is true when E > F
   - E.GE.F is true when E ≥ F
   - E.LT.F is true when E < F
   - E.LE.F is true when E ≤ F
   - E.EQ.F is true when E = F
   - E.NE.F is true when E ≠ F
3. A logical constant or variable is an expression.
4. A relation is an expression.
5. An expression enclosed in parentheses is an expression.
6. \( \textbf{.not.} \) any expression is an expression.
7. Any two expressions connected by .AND. create an expression.
8. Any two expressions connected by .OR. create an expression.

**Hierarchy of Operations**

Arithmetic expressions are calculated first, then values are found for relations. These, and variables, are then operated upon by .NOT.. Next, results are connected by .AND. and these results are connected by .OR.. Parentheses indicate any requisite recursions. Associated with this expression, then, is a value .TRUE. or .FALSE. but not both.

**Chapter 3. Arithmetic Statements**

**Normal Arithmetic Statements**

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;a=b&quot; where a is a variable (subscripted or not subscripted) and b is an expression.</td>
<td>Q1 = K ( A(I) = B(I) \cdot \sin (C(I)) ) (SIN and other functions are discussed in Chapter 6.)</td>
</tr>
</tbody>
</table>

The arithmetic formula defines a numerical or logical calculation. A FORTRAN arithmetic statement resembles very closely a conventional arithmetic formula. However, in a FORTRAN arithmetic statement the equal sign means “is to be replaced by,” not “is equivalent to.” Thus, the arithmetic statement

\[ Y = N - \text{LIMIT}(I - 2) \]

means that the value of \( N - \text{LIMIT}(I - 2) \) is to replace the value of Y. The result is stored in fixed-point or in floating-point form if the variable to the left of the equal sign is a fixed-point or a floating-point variable, respectively.

If the variable on the left is fixed point and the expression on the right is floating point, the result will first be computed in floating point and then truncated and converted to a fixed-point integer. Thus, if the result is +3.872, the fixed-point number stored will be +3, not +4. If the variable on the left is floating point and the expression on the right fixed point, the latter will be computed in fixed point, and then converted to floating point.

The following examples illustrate types of arithmetic statements:

<table>
<thead>
<tr>
<th>EXAMPLES</th>
<th>MEANING</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=B</td>
<td>Store the value of B in A.</td>
</tr>
<tr>
<td>I=B</td>
<td>Truncate B to an integer, convert to fixed point, and store in I.</td>
</tr>
<tr>
<td>A=I</td>
<td>Convert I to floating point, and store in A.</td>
</tr>
<tr>
<td>I=I+1</td>
<td>Add 1 to I and store in I. This example illustrates the fact that an arithmetic statement is not an equation, but is a command to replace a value.</td>
</tr>
<tr>
<td>A=3.0*B</td>
<td>Replace A with the product of B multiplied by 3.0.</td>
</tr>
<tr>
<td>A=1*B</td>
<td>Not permitted. The expression is mixed, i.e., contains both fixed-point and floating-point variables.</td>
</tr>
<tr>
<td>A=3*B</td>
<td>Not permitted. The expression is mixed.</td>
</tr>
</tbody>
</table>

**Logical Statements**

A logical variable is equated to a logical expression. The expression will be evaluated and the appropriate value .TRUE. or .FALSE. will be stored as the current value of the variable. Example:

\[ T = Q \cdot \text{AND}(\text{.NOT.R}) \cdot \text{AND(V.GT.5))} \]
Chapter 4. Control Statements and END Statement

This class of FORTRAN statements is the set of control statements which enable the programmer to state the flow of his program.

UNCONDITIONAL GO TO

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GO TO n&quot; where n is a statement number. This statement causes transfer of control to the statement with statement number n.</td>
<td>GO TO 3</td>
</tr>
</tbody>
</table>

COMPUTED GO TO

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GO TO (n1, n2, ..., nm), i&quot; where n1, n2, ..., nm are statement numbers and i is a non-subscripted fixed point variable.</td>
<td>GO TO (30, 42, 50, 9), I</td>
</tr>
</tbody>
</table>

Control is transferred to the statement with statement number n1, n2, n3, ..., nm depending on the current value of i. Thus, in the example, if i is 3 at the time of execution, a transfer to the third statement of the list, (statement 50), will occur. This statement is used to obtain a computed many-way fork.

ASSIGNED GO TO

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;GO TO i,&quot; where i is a non-subscripted fixed-point variable appearing in a previously executed ASSIGN statement.</td>
<td>GO TO K, GO TO JOE, (10, 20, 30)</td>
</tr>
</tbody>
</table>

This statement causes transfer of control to the statement with statement number equal to that value of i which was last assigned by an ASSIGN. The assigned GO TO is used to obtain a pre-set many-way fork. An optional form of this statement is GO TO i, (n1, n2, ..., nm) where n1, n2, ..., nm are statement numbers.

ASSIGN

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ASSIGN n TO i&quot; where n is a statement number and i is a non-subscripted fixed point variable which appears in an assigned GO TO statement.</td>
<td>ASSIGN 12 to k</td>
</tr>
</tbody>
</table>

This statement causes a subsequent GO TO i to transfer control to the statement with the statement number n.

IF

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;IF(t)S&quot; where t is a logical expression and S is any executable FORTRAN statement, except DO, or another IF statement.</td>
<td>IF(A.AND.B)F=SIN(B) IF(16.GT.L)GO TO 24 IF(D.OR.X.NE.Y)GO TO 18,20</td>
</tr>
</tbody>
</table>

If t is .TRUE., statement S is executed; if t is .FALSE., control passes to the next statement.

DO

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DO n i=m, m&quot; or &quot;DO n i=m1, m2, m3&quot; where n is a statement number, i is a non-subscripted fixed point variable, and m1, m2, m3 are each either an unsigned fixed point constant or non-subscripted fixed point variable. If m3 is not stated, it is taken to be 1.</td>
<td>DO 30 I=1, 10 DO 30 I=1, M, 3 DO 30 I=N, M, L</td>
</tr>
</tbody>
</table>

The DO statement is a command to execute repeatedly the statements which follow, up to and including the statement with statement number n.

The first time, the statements are executed with i = m1. For each succeeding execution, i is increased by m2. After they have been executed with i equal to the highest of this sequence of values which does not exceed m3, control passes to the statement following the last statement in the range of the DO.

The range of a DO is that set of statements which will be executed repeatedly, i.e., it is the sequence of consecutive statements immediately following the DO, up to and including the statement numbered n.

The index of a DO is the fixed point variable i, which is controlled by the DO in such a way that its value begins at m1 and is increased each time by m2 until it is about to exceed m3. Throughout the range it is available for computation, either as an ordinary fixed-point variable or as the variable of a subscript. After the last execution of the range, the DO is said to be satisfied.

Suppose, for example, that control has reached statement 10 of the program:

```
.
.
.
10 DO 11 I=1, 10
11 A(1)=1*N(1)
   12
```

The range of the DO is statement 11, and the index is I. The DO sets I to 1 and control passes into the range. The value of 1*N(1) is computed, converted to floating point, and stored in location A(1). Since statement 11 is the last statement in the range of the DO and the DO is unsatisfied, I is increased to 2 and control returns to the beginning of the range. Statement 11. The value of 2*N(2) is then computed and stored in location A(2). The process continues until statement 11 has been executed with I = 10. Since the DO is satisfied, control then passes to statement 12.
DO's Within DO's

Among the statements in the range of a DO may be other DO statements. When this is so, the following rule must be observed:

Rule 1: If the range of a DO includes another DO, then all of the statements in the range of the second must also be in the range of the first.

A set of DO's satisfying this rule is called a nest of DO's.

Transfer of Control

Transfers of control from and into the range of a DO are subject to the following rule:

Rule 2: No transfer is permitted into the range of any DO from outside its range. Thus, in the configuration (Figure 1), 1, 2, and 3 are permitted transfers, but 4, 5, and 6 are not.

Figure 1

Preservation of Index

When control leaves the range of a DO in the ordinary way (i.e., when the DO becomes satisfied and control passes on to the next statement after the range) the exit is said to be a normal exit. After a normal exit from a DO occurs, the value of the index controlled by that DO is not defined, and the index cannot be used again until it is redefined.

If exit occurs by a transfer out of the range, however, the current value of the index remains available for any subsequent use. If exit occurs by a transfer which is in the ranges of several DO's the current values of all the indexes controlled by those DO's are preserved for any subsequent use.

Restrictions on Statements

Only one type of statement is not permitted in the range of a DO, namely, any statement which redefines the value of the index or of any of the indexing parameters. In other words, the indexing of a DO loop must be completely set before the range is entered and the initial, terminal, and incremental values of the index may not be altered.

Exits

When a CALL statement is executed in the range of a DO, care must be taken that the subprogram does not alter the DO index or indexing parameters. This applies as well when a closed function is invoked in the range of a DO. (See Chapter 6.)

CONTINUE

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;CONTINUE&quot;</td>
<td>CONTINUE</td>
</tr>
</tbody>
</table>

CONTINUE is a dummy statement which gives rise to no instructions in the object program. It is most frequently used as the last statement in the range of a DO to provide a transfer address for IF and GOTO statements which are intended to begin another repetition of the DO range.

As an example of a program which requires a CONTINUE, consider the table search:

```
  
  10   DO 12 I=1,100  
       IF (ARG.NE.VALUE(I)) GO TO 12  
       GO TO 20  
       12  CONTINUE  
   
   
   
   
```

The program will scan the 100-entry VALUE table an entry at a time until it finds the first entry which equals the value of the variable ARG, whereupon it exits to statement 20 with the value of I available for fixed point use; if no entry in the table equals the value of ARG, a normal exit to the statement following the CONTINUE will occur.

PAUSE

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;PAUSE&quot;</td>
<td>PAUSE</td>
</tr>
</tbody>
</table>

A PAUSE statement causes a temporary halt in object program execution, providing the operator with the option of resuming or of abandoning the job. The contents of the instruction counter are displayed on the console numerical display.

STOP

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;STOP&quot;</td>
<td>STOP</td>
</tr>
</tbody>
</table>

This statement causes an “Abnormal End-of-Job” return to MCP (“Normal End-of-Job” return to MCP is accomplished by the RETURN statement. See Chapter 6.)
This statement differs from the previous statements discussed in this chapter in that it does not affect the flow of control in the object program being compiled. The END statement marks the end of any given FORTRAN source program, separating it from the program that follows. Each FORTRAN source program must be terminated with an END statement.

Chapter 5. Input-Output Statements

Nine statements in IBM 7030 FORTRAN IV specify the transmission of information to or from magnetic tapes, disks, card readers, card punches, and printers. These statements may be grouped as follows:

1. System Input and System Output Statements:
   Three statements, READ, PUNCH and PRINT, cause the transmission of edited (decimal) information to or from the system input and system output devices.

2. General Read/Write Statements: Two statements, READ ( ) and WRITE ( ), cause the transmission of either edited (decimal) or non-edited (binary) data to or from magnetic tapes, disk or, under certain conditions, non-system readers, punches, and printers. The choice between edited and non-edited transmission is indicated by the presence or absence of a "format" specification within the parentheses of these so-called generalized Read/Write commands.

3. Manipulative Statements: Three statements, END FILE, BACKSPACE, and REWIND, affect magnetic tapes or the disk unit.

4. One statement, FORMAT, which is a non-executable statement, specifies the editing process required in transmission of information to or from the i-o device.

Array and List Specifications

Specifying Lists of Quantities

Some i-o statements call for the transmission of information and must, therefore, include a list of the quantities to be transmitted. This list is ordered, and its order must be the same as the order in which the words of information exist (for input), or will exist (for output) in the input-output medium.

The formation and meaning of a list is best described by an example:

| A, B(3), C(1), D(1,K), C(2), D(2,K), ..., C(10), D(10,K), E(1,1), E(3,1), ..., E(9,1), F(1,3), E(1,2), E(3,2), ..., E(9,2), F(2,3), ..., E(1,K), E(3,K), ..., E(9,K), F(K,3) |

Similarly, if this list were used with an input statement, the successive words, as they were read from the external medium, would be placed into the sequence of storage locations just given.

Thus the list reads from left to right with repetition for variables enclosed within parentheses. Only variables, and not constants, may be listed. The execution is exactly that of a DO-loop, as though each opening parentheses (except subscripting parentheses) were a DO, with indexing given immediately before the matching closing parentheses, and with the DO range extending up to that indexing information. The order of the above list can thus be considered the equivalent of the "program":

```
A
B(3)
DO 5 I=1, 10, 1
C(I)
5 D(I,K)
DO 9 J=1, K, 1
DO 8 I=1, 10, 2
8 E(I,J)
9 F(J,3)
```

Note that indexing information, as in DO's, consists of three constants or fixed point variables, and that the last of these may be omitted, in which case it is taken to be 1.

For a list of the form K, (A(I,K)) or K, (A(I), I=1, K) where an index or indexing parameter itself appears earlier in the list of an input statement, the indexing will be carried out with the newly read-in value.

Input-Output in Array Form

FORTRAN treats variables according to conventional matrix practice. Thus, the input-output statement,

```
READ 1 ((A(I,J), I=1,12), J=1,3)
```

causes the reading of I x J (in this case 2 x 3) items of information. The data items will be read into storage in the same order as they are found on the input medium.

Input-Output of Entire Arrays

When input-output of an entire matrix is desired, an abbreviated notation may be used for the list of the input-output statement; only the name of the array need be given and the indexing information may be omitted.

Thus, if A has previously been listed in a DIMENSION statement, the statement,

```
READ 1, A
```

is sufficient to read in all of the elements of the array A.
Format Specifications

**FORMAT**

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;FORMAT (Specification)&quot; where Specification is as described below.</td>
<td>FORMAT (12/(E12.4,F10.4))</td>
</tr>
</tbody>
</table>

The system input/output and general READ/WRITE statements may contain, in addition to the list of quantities to be transmitted, the statement number of a FORMAT statement describing the information format to be used. Alternatively, the name of an array containing A-type data may be supplied. In this case, this data is to be interpreted (at object time) as a FORMAT specification. It also specifies the type of conversion to be performed between the internal machine-language and external notation. FORMAT statements are not executed; their function is merely to supply information to the object program. Therefore, they may be placed anywhere in the source program.

For the sake of clarity, the details of writing a FORMAT specification are given below for use with PRINT statements. The description is valid for any case, however, simply by generalizing the concept of "printed line" to that of unit record in the input-output medium.

A unit record may be:
1. A printed line with a maximum of 132 characters.
2. A punched card with a maximum of 72 characters.
3. A BCD tape record with a maximum of 132 characters.

**Numeric Fields**

Three basic types of decimal-to-binary or binary-to-decimal conversion are available:

<table>
<thead>
<tr>
<th>INTERNAL</th>
<th>TYPE</th>
<th>EXTERNAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Floating point variable</td>
<td>E</td>
<td>Floating point, decimal</td>
</tr>
<tr>
<td>Floating point variable</td>
<td>F</td>
<td>Fixed point, decimal</td>
</tr>
<tr>
<td>Fixed point variable</td>
<td>I</td>
<td>Decimal integer</td>
</tr>
</tbody>
</table>

The FORMAT specification describes the line to be printed by giving for each field in the line (from left to right, beginning with the first type wheel):
1. The type of conversion (E, F, or I) to be used.
2. The width (w) of the field.
3. For the E- and F-type conversion, the number of places (d) after the decimal point that are to be printed (d is treated modulo 14).

These basic field specifications are given in the forms

\[ \text{Iw, Ew.d, and Fw.d} \]

with the specification for successive fields separated by commas. Thus, the statement FORMAT (12, E12.4, F10.4) might give the line:

\[ 27 \quad -0.9321E03 \quad -0.0076 \]

As in this example, the field widths may be made greater than necessary so as to provide spacing blanks between numbers. In this case, there is one blank following the 27, one blank after the E (automatically supplied except in cases of a negative exponent, when a minus sign will appear), and three blanks after the 03. Within each field the printed output will always appear in the rightmost positions.

**Alphabetic Fields**

There are two ways of reading or writing alphabetic information; the specifications for this purpose are Aw and wH. Both result in storing the information internally in A8 form. The basic difference is that information handled with the A specification is given a variable or array name and hence can be referred to by means of this name for processing and/or modification. Information handled with the H specification is not given a name and cannot be referred to or manipulated in storage in any way.

The specification Aw causes w characters to be read into, or written from, a variable or array.

The specification wH is followed in the FORMAT statement by w alphabetic characters. Example:

\[ 32\text{HSHE WAS POOR, BUT SHE WAS HONEST} \]

Note that blanks are considered alphabetic characters, and must be included as part of the count w.

It is possible to read and write alphabetic information only, by giving no list with the input-output statement and specifying no I, E, or F fields in the FORMAT statement.

Consider an alphabetic field in a FORMAT statement at the time of execution of the object program. If the FORMAT statement is used with an input statement, the alphabetic text listed in the FORMAT statement will be replaced by whatever text is read in from the corresponding field in the input-output medium. When that same FORMAT statement is used for output, whatever information is then in the FORMAT statement will appear in the output data. Thus, text can be originated in the source program, or as input to the object program.

**Blank Fields**

Blank characters may be provided in an output record and characters of an input record may be skipped by means of the specifications wX where \( 0 < w \leq 132 \) (w is the number of blanks provided or characters skipped). When the specification is used with an input record, w characters are considered to be blank regardless of what they actually are, and are skipped over. (The control character X need not be separated by a comma from the specification of the next field.)
Repetition of Fields

It may be desired to print n successive fields within one record, in the same fashion. This may be specified by giving n (an unsigned fixed point constant) before E, F, I or A. Thus, the statement:

\[
\text{FORMAT} \ (12, \ 3E12.4)
\]

might give

\[
27 \ -0.9321E \ 02 \ -0.7580E \ 02 \ 0.5536E \ 00
\]

Repetition of Groups

A limited parenthetical expression is permitted in order to permit repetition of data fields according to certain format specifications within a longer format statement specification. Thus, FORMAT \ (2(F10.6, \ E10.2, \ I4) \) is equivalent to \ (F10.6, \ E10.2, \ F10.6, \ E10.2, \ I4) \.

Scale Factors

To permit more general use of F-type conversion, a scale factor followed by the letter P may precede the specification. The scale factor is defined such that:

\[
\text{Printed number} = \text{Internal number} \times 10^{\text{scale factor}}.
\]

Thus, the statement FORMAT \ (12, \ 1P3F1.3) \ used with the data of the preceding example would give:

\[
27 \ -932.096 \ -0.076 \ 5.536
\]

whereas FORMAT \ (12, \ -1P3F1.3) \ would give:

\[
27 \ -9.321 \ -0.001 \ 0.055
\]

A positive scale factor may also be used with E-type conversion to increase the number and decrease the exponent. Thus, FORMAT \ (12, \ 1P3E12.4) \ would produce with the same data:

\[
27 \ -9.3210E \ 01 \ -7.5804E \ 03 \ 5.5361E \ 01
\]

The scale factor is assumed to be zero if no other value has been given. However, once a value has been given, it will hold for all E- and F-type conversions following the scale factor within the same format statement. (This applies to both single-record and multiple-record formats.) Once a scale factor has been given, a subsequent scale factor of zero in the same format statement must be specified by P. Scale factors have no effect on I-conversion.

Multiple-Record Formats

To deal with a block of more than one line of print, a format specification may have several different one-line formats, separated by a slash (/) to indicate the beginning of a new line. Thus, FORMAT \ (3F9.2, \ 2F10.4/SE14.5) \ would specify a multi-line block of print in which lines 1, 3, 5, . . . have format \ (3F9.2, \ 2F10.4), \ and lines 2, 4, 6, . . . have format \ SE14.5. \n
If a multiple-line format is desired such that the first two lines will be printed according to a special format and all remaining lines according to another format, the last line-specification should be enclosed in a second pair of parentheses; e.g., FORMAT \ (12, \ 3E12.4/2F10.3, \ 3F9.4/(10F12.4)). \n
If data items remain to be transmitted after the format specification has been completely "used," the format repeats from the last open parenthesis.

As these examples show, both the slash and the closing parenthesis of the format statement indicate termination of a record.

Blank lines may be introduced into a multi-line format statement, by listing consecutive slashes. N + 1 consecutive slashes produce N blank lines.

FORMAT and Input-Output Statement Lists

The FORMAT statement indicates, among other things, the size of each record to be transmitted. In this connection, it must be remembered that the FORMAT statement is used in connection with the list of some particular input-output statement, except when a format statement consists entirely of alphabetic fields. In all other cases, control in the object program switches back and forth between the list (which specifies whether data remains to be transmitted) and the FORMAT statement (which gives the specifications for transmission of that data).

Ending a FORMAT Statement

During input-output of data, the object program scans the format statement to which the relevant input-output statement refers. When a specification for a numeric field is found and list items remain to be transmitted, input-output takes place according to the specification and scanning of the FORMAT statement resumes. If no items remain, transmission ceases and execution of that particular input-output statement is terminated. Thus, a decimal input-output operation will be brought to an end when a specification for a numeric field or the end of the FORMAT statement is encountered, and there are no items remaining in the list.

FORMAT Statements Read in at Object Time

FORTRAN will accept a variable format address. This provides the facility of specifying a record format at object time. Examples:

\[
\text{DIMENSION FMT}(9), \ C(5)
\]

1 FORMAT \ (9A8) 
5 READ 1, \ (FMT(1), \ I=1,9) 
10 READ FMT, \ A,B,(C(1), \ I=1,5)

Thus, A, B, and the array C would be converted and stored, according to the FORMAT specification read into the array FMT, by statement 5.

The format read in at object time must take the same form as a source program FORMAT statement, except that the word FORMAT is omitted, i.e., the variable format begins with a left parenthesis.
Carriage Control

A write statement that includes a format number prepares a BCD tape which can be used to obtain off-line printed output. Off-line printing will be done on a 1401 with a program which operates in one of three modes: single space, double space, and program control. Under program control, which gives the greatest flexibility, the first character of each BCD record controls spacing of the 1403 printer and that character is not printed. The control characters and their effects are:

- blank: Single space before printing
- 0: Double space before printing
- 1: Restore paper before printing

Any other control character will be passed to the output device unchanged. Thus, a format statement for this type of write statement will usually begin with 1H followed by the appropriate control character, if the printing is to be done under program control.

Data Input to the Object Program

Decimal input data to be read when the object program is executed must be in essentially the same format as given in the previous examples. Thus, a card to be read according to FORMAT (I2, E12.4, F10.4) might be punched:

```
27  -0.8321E 02  -0.0076
```

Within each field, all information must appear at the extreme right. Plus signs may be omitted or indicated by a blank or +. Minus signs may be punched with an 11-punch or an 8-4 punch. Blanks in numeric fields are regarded as zeros. Numbers for E- and F-type conversion may contain any number of digits, but only the high-order digits will be retained (no rounding will be performed). Numbers for I-type conversion will be treated modulo 2^32.

To permit economy in punching, certain relaxations in input data format are permitted:

1. Numbers of E-type conversion need not have 5 columns devoted to the exponent field. The start of the exponent field must be marked by an E, or if that is omitted, by a + or - (not a blank). Thus E2, E02, +2, +02, E-02, and E+02 are permissible exponent fields.

2. Numbers for E- or F-type conversion need not have their decimal point punched. If it is not punched, the format specification will supply it; for example, the number -09321+2 with the specification E12.4 will be treated as though the decimal point had been punched between the 0 and the 9. If the decimal point is punched in the card, its position overrides the indicated position in the format specification.

System Input and System Output Statements

**System READ**

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;READ f, list&quot; where f is either the statement number of a FORMAT statement, or the name of an array containing A-type data to be used as a format specification.</td>
<td>READ 10,A,B(3), C(1),I=1,5</td>
</tr>
</tbody>
</table>

The READ statement is used to input decimal information from the System reader. Card after card is read until the complete list has been satisfied.

**System PUNCH**

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;PUNCH f, list&quot; where f is either the statement number of a FORMAT statement, or the name of an array containing A-type data to be used as a format specification.</td>
<td>PUNCH 251, (I(Z(I,J)), I=1, 5), J=1, I</td>
</tr>
</tbody>
</table>

The PUNCH statement causes cards to be punched by the systems punch. Cards are punched in accordance with the specified format until the entire list has been satisfied.

**System PRINT**

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;PRINT f, list&quot; where f is either the statement number of a FORMAT statement, or the name of an array containing A-type data to be used as a format specification.</td>
<td>PRINT FMT,K,(A(J), J=M,K,2)</td>
</tr>
</tbody>
</table>

The PRINT statement causes the object program to prepare a system output tape for subsequent off-line printing. Successive lines are written in accordance with the specified format until the complete list has been satisfied.

General Read/Write Statements

Two basic types of input/output activity are controlled by the general Read/Write statements: decimal (in a format) data transmission, and binary (not in a format) data transmission. The particular action required is specified by the presence or absence of a format specification in the statement concerned.

**Binary READ**

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;READ (unit) list&quot; where unit is an unsigned fixed-point variable, and list is as described elsewhere.</td>
<td>READ (5)A,B,C(3,1,9), READ (J) (M(N,N), N=1,NMAX)</td>
</tr>
</tbody>
</table>
This form of the general READ statement causes the input of binary data from either binary magnetic tapes, or from disk, depending upon the definition supplied for the value of “unit” at 100 compile time. (See Part II, Chapter 11.)

If the list calls for less data than is present in the logical record being brought in, then the remaining, unread, data is spaced over.

**(Decimal) READ**

<table>
<thead>
<tr>
<th>General Form</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;READ (unit, f) list&quot; where unit is an unsigned fixed-point variable or constant, and f is either the statement number of a FORMAT statement, or the name of an array containing A8 data to be used as a format specification.</td>
<td>READ (6,10)A, READ (ITAPE, 707)K, (A(1), I=1, K)</td>
</tr>
</tbody>
</table>

This version of the general READ statement causes the input of information, in accordance with the specified format, from either BCD tape or under special conditions, as described in the next section, from the system reader.

The actual I/O device actuated by this statement depends upon the running time value of “unit,” and the particular definition supplied at 100 compile time for this value.

If the list calls for less data than is present in the logical record being brought in, then the remaining, unread data is spaced over.

**(Binary) WRITE**

<table>
<thead>
<tr>
<th>General Form</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;WRITE (unit) list&quot; where unit is an unsigned fixed-point variable or constant, and list is described elsewhere.</td>
<td>WRITE (FILE)((A(I,J), I=1,10), J=1,10), WRITE (NO1)PQ,R</td>
</tr>
</tbody>
</table>

This form of the general WRITE statement causes the output of binary data onto either binary magnetic tape or disk, depending upon the actual value of “unit” at run time, and the definitions supplied at 100-compile time.

The output of a single WRITE statement is considered to be just one logical record.

**(Decimal) WRITE**

<table>
<thead>
<tr>
<th>General Form</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;WRITE (unit, f) list&quot; where unit is an unsigned fixed-point variable or constant, and f is either the statement number of a FORMAT statement, or the name of an array containing A8 data to be used as a format specification.</td>
<td>WRITE (LIST,FORMAT) D,A,T,U,M, WRITE (99,10)(Q(3*1-2), I=1,200)</td>
</tr>
</tbody>
</table>

This form of the general WRITE statement may be used to put out data according to the specified format, either onto BCD magnetic tape or, under special circumstances, onto system output tapes. It may also be used to put information on the non-system printer. These various options are controlled by programmer action at 100-compile time.

**Manipulative Statements**

**END FILE**

<table>
<thead>
<tr>
<th>General Form</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;END FILE unit&quot; where unit is an unsigned fixed-point variable or constant.</td>
<td>END FILE 10, END FILE JTAPE</td>
</tr>
</tbody>
</table>

The END FILE statement causes the object program to write an end-of-file mark on the designated symbolic unit, provided the definition supplied via an 100 card is meaningful in this context.

**BACKSPACE**

<table>
<thead>
<tr>
<th>General Form</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;BACKSPACE unit&quot; where unit is an unsigned fixed-point variable or constant.</td>
<td>BACKSPACE 3, BACKSPACE M1</td>
</tr>
</tbody>
</table>

The BACKSPACE statement is used to backspace the symbolic file “unit” one logical record.

**REWIND**

<table>
<thead>
<tr>
<th>General Form</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;REWIND unit,&quot; where unit is an unsigned fixed-point constant or variable.</td>
<td>REWIND 3, REWIND K</td>
</tr>
</tbody>
</table>

The REWIND statement causes the object program to rewind the symbolic file identified by “unit.” If “unit” is defined to be a disk file, then the meaning here is to locate the first arc of the file.

**Chapter 6. Subprogram Statements, Functions and Subroutines**

It is possible to construct, in the FORTRAN language, subprograms which are referred to by other programs. These subprograms may, in turn, refer to still other lower level subprograms, which may also be coded in FORTRAN language. It is therefore possible to code problems using several levels of subprograms. This configuration may be thought of as a total problem consisting of one main program and any number of subprograms.

Because of the interrelationship among several different programs, it is possible to include a block of hand-coded instructions in a sequence including in-
Instructions compiled from FORTRAN source programs. It is only necessary that hand-coded instructions conform to rules for subprogram formation, since they will comprise a distinct subprogram.

This section presents a discussion of the various types of subprograms possible, two of which are the FORTRAN-coded Function subprogram and the Subroutine subprogram.

Although Function subprograms and Subroutine subprograms are treated together and may be viewed as similar, it must be remembered that they differ in two fundamental respects.

1. The Function subprogram is always single-valued, whereas the Subroutine subprogram may be multi-valued.

2. The Function subprogram is called or referred to by the arithmetic expression containing its name; the Subroutine subprogram can only be referred to by a CALL statement.

Each of these two types of subprograms, when coded in FORTRAN language, must be regarded as independent FORTRAN programs. In all respects, they conform to rules for FORTRAN programming. They may be compiled with the main program of which they are parts, however, by means of multiple program compilation. In this way, the result of multiple program compilation will be a complete main program-subprogram sequence ready to be executed.

Subprograms may be referred to in two distinct ways. One of these is by the use of the name in an arithmetic expression. This applies to Functions. The other, which applies to Subroutine subprograms, is by means of a CALL statement.

Following are examples of arithmetic expressions including Function names:

\[ Y = \text{ABS}(B-C) \]
\[ C = \text{AMIN0}(M, L) + \text{ABC}(B \times \text{FORTF}(Z), E) \]

The **appearance in the arithmetic expression serves to call the Function**; the value of the Function is then computed, using the arguments which are supplied in the parentheses following the Function name. (A value is here defined to be a single quantity.)

### Definitions

There are three different types of FORTRAN Functions:

1. **Open (or built-in) Functions.**
2. **Arithmetic or Logical Statement Functions.**
3. **Closed Functions.**

### Open (or Built-in) Functions

These are Functions which are predefined, and exist in the processor. They generate instructions which are compiled in-line every time the function is referred to.

The IBM 7030 FORTRAN IV System will contain the 21 built-in Functions shown in Figure 2. It has the capacity for more Functions to be added.

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
<th>No. of Args</th>
<th>Argument Mode</th>
<th>Function Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>(</td>
<td>\text{Abs}</td>
<td>)</td>
<td>1</td>
</tr>
<tr>
<td>IABS</td>
<td></td>
<td></td>
<td>Fixed</td>
<td>Fixed</td>
</tr>
<tr>
<td>ALNT</td>
<td>Sign of Arg times</td>
<td>1</td>
<td>Floating</td>
<td>Floating</td>
</tr>
<tr>
<td>INT</td>
<td>Largest Integer&lt;</td>
<td>Arg</td>
<td></td>
<td>Fixed</td>
</tr>
<tr>
<td>AMOD</td>
<td>Arg mod Arg</td>
<td>2</td>
<td>Floating</td>
<td>Floating</td>
</tr>
<tr>
<td>MOD</td>
<td></td>
<td></td>
<td>Fixed</td>
<td>Fixed</td>
</tr>
<tr>
<td>AMAX0</td>
<td>Max (Arg1, Arg2, ...)</td>
<td>2</td>
<td>Floating</td>
<td>Floating</td>
</tr>
<tr>
<td>AMAX1</td>
<td></td>
<td></td>
<td>Fixed</td>
<td>Floating</td>
</tr>
<tr>
<td>MAX0</td>
<td></td>
<td></td>
<td>Floating</td>
<td>Fixed</td>
</tr>
<tr>
<td>MAX1</td>
<td></td>
<td></td>
<td>Fixed</td>
<td>Floating</td>
</tr>
<tr>
<td>AMIN0</td>
<td>Min (Arg1, Arg2, ...)</td>
<td>2</td>
<td>Fixed</td>
<td>Floating</td>
</tr>
<tr>
<td>AMIN1</td>
<td></td>
<td></td>
<td>Floating</td>
<td>Fixed</td>
</tr>
<tr>
<td>MIN0</td>
<td></td>
<td></td>
<td>Fixed</td>
<td>Floating</td>
</tr>
<tr>
<td>MIN1</td>
<td></td>
<td></td>
<td>Floating</td>
<td>Fixed</td>
</tr>
<tr>
<td>FLOAT</td>
<td>Fixed to floating</td>
<td>1</td>
<td>Fixed</td>
<td>Floating</td>
</tr>
<tr>
<td>IFIX</td>
<td>Floating to fixed</td>
<td>1</td>
<td>Floating</td>
<td>Fixed</td>
</tr>
<tr>
<td>SIGN</td>
<td>(Sign of Arg) times</td>
<td>2</td>
<td>Floating</td>
<td>Floating</td>
</tr>
<tr>
<td>ISIGN</td>
<td></td>
<td></td>
<td>Fixed</td>
<td>Floating</td>
</tr>
<tr>
<td>DIM</td>
<td>Arg1 - min(Arg1, Arg2)</td>
<td>2</td>
<td>Floating</td>
<td>Fixed</td>
</tr>
<tr>
<td>IDIM</td>
<td></td>
<td></td>
<td>Fixed</td>
<td>Floating</td>
</tr>
<tr>
<td>SQRT</td>
<td>Arg1</td>
<td>1</td>
<td>Floating</td>
<td>Floating</td>
</tr>
</tbody>
</table>

**Note:** The function AMOD (Arg1, Arg2) = Arg1 \(\bmod\) Arg2, where \(\bmod\) means "integer part of." Hence, this quantity is the remainder of Arg1/Arg2 when the quotient is expressed as an integer, and the sign of Arg1 is given to the remainder.

Figure 2

### Functions

**Naming**

The rules for naming Functions are the same as those for naming variables. The name may be one to six alphanemic characters (except special characters), and the first must be alphabetic. The mode of the Function must be given either by a Type statement or, in the case of integer and real, by the proper first letter of the name as described in the rules for naming variables.

### Arithmetic or Logical Statement Functions

These are Functions which are defined by a single FORTRAN arithmetic or logical statement and apply only to the particular program or subprogram in which their definition appears.
The right-hand side of an arithmetic statement Function may be any expression, not involving subscripted variables, that meets the requirements specified for expressions. In particular, it may involve Functions freely, provided that any such Function, if it is not built-in or available, has been defined in a preceding arithmetic statement Function. As many as desired of the variables appearing in the expression on the right-hand side may be stated on the left-hand side to be the arguments of the Function. Since the arguments are only dummy variables, their names are unimportant (except as indicating fixed or floating point mode) and may even be the same as names appearing elsewhere in the program.

Those variables on the right-hand side which are not stated as arguments are treated as parameters. Thus, if \texttt{FIRST(X)} is defined in an arithmetic statement Function as \texttt{FIRST(X)=A*X+B} then a later reference to \texttt{FIRST(Y)} will cause \texttt{A*Y+B} based on the current values of \texttt{a}, \texttt{b}, and \texttt{y}, to be computed. The naming of parameters, therefore, must follow the normal rules of uniqueness.

A Function of this type may be used just as any other Function. In particular, its arguments may be expressions and may involve subscripted variables; thus, a reference to \texttt{FIRST(Z+Y+1)}, with the above definition of \texttt{FIRST}, will cause \texttt{A*(Z+Y+1)+B} to be computed on the basis of the current values of \texttt{a}, \texttt{b}, \texttt{y}, and \texttt{z}.

Functions defined by arithmetic or logical statements are always compiled as closed subroutines.

NOTE: All the Function definitions in a program must precede the first executable statement of the program.

CLOSED FUNCTIONS

These are Functions that are prewritten and/or precompiled and may exist on the library tape or in prepared card decks. These Functions are compiled as "closed" subroutines; instead of appearing in the object program for every reference that has been made to them in the source program, they appear only once regardless of the number of references.

Closed Functions may be conveniently defined by a conventional FORTRAN program. In this instance com-
names. That is, during object program execution other actual arguments are substituted for them. Therefore, the arguments which follow the Function reference in the calling program must agree with those in the FUNCTION statement in the subprogram in number, order, and type. Furthermore, when a dummy argument is an array name, the corresponding actual argument must also be an array name. Each of these array names must appear in DIMENSION statements of their respective programs with the same dimensions, unless the dimensions are variable.

None of the dummy variables may appear in EQUIVALENCE statements in the Function subprogram.

Hand-coding closed Functions may be added to the library. Among the closed functions distributed in the FORTRAN library are:

<table>
<thead>
<tr>
<th>NAME</th>
<th>TYPE OF FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALOC</td>
<td>Natural Logarithm</td>
</tr>
<tr>
<td>SIN</td>
<td>Trigonometric Sine</td>
</tr>
<tr>
<td>COS</td>
<td>Trigonometric Cosine</td>
</tr>
<tr>
<td>EXP</td>
<td>Exponential</td>
</tr>
<tr>
<td>ATAN</td>
<td>Arctangent</td>
</tr>
<tr>
<td>TANH</td>
<td>Hyperbolic Tangent</td>
</tr>
</tbody>
</table>

**Subroutines**

**SUBROUTINE**

```plaintext
“SUBROUTINE Name (a1, a2, ..., an) where Name is the symbolic name of a subprogram, and the arguments a1, a2, ..., an, if any, are non-subscripted variable names, or the names of lower level subprograms. The name of the subprogram may consist of one to six alphanumeric characters, the first of which is alphabetic. Also, the name of the subprogram must not be listed in a DIMENSION statement of any program which calls the subprogram, or in a DIMENSION statement of the subprogram itself.

This statement is used as the first statement of a subroutine subprogram and defines it to be such. A subprogram introduced by the SUBROUTINE statement must be a FORTRAN program and may contain any FORTRAN statements except FUNCTION or another SUBROUTINE statement.

Unlike the Function-type subprogram which returns only a single numerical value, the Subroutine subprogram uses one or more of its arguments to return output. The arguments so used must, therefore, appear on the left side of an arithmetic statement within the program.

If an argument is an array name, it must appear in a DIMENSION statement in the subroutine.

None of the arguments may appear in EQUIVALENCE statements in the Subroutine subprogram. These subprograms may be compiled independently or multiple-compiled with others.

**CALL**

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>“CALL Name (a1, a2, ..., an)”</td>
<td>CALL MATMPY</td>
</tr>
<tr>
<td>where Name is the name of a Subroutine subprogram, and a1, a2, ..., an are the arguments which take one of the forms described below.</td>
<td>(x5,10,y7,z)</td>
</tr>
<tr>
<td></td>
<td>CALL QDRTIC</td>
</tr>
<tr>
<td></td>
<td>(p*9.732,q/4.536, r-s**2, x1, x2)</td>
</tr>
</tbody>
</table>

This statement is used to call Subroutine subprograms; the CALL transfers control to the subprograms and presents it with the parenthesized arguments. Each argument may be one of the following:

1. Fixed point constant, floating point constant, logical constant.
2. A variable, with or without subscripts, fixed, floating or logical types.
3. Arithmetic expression, or logical expression.
4. Alphameric characters. Such arguments must be preceded by nH where n is the count of characters included in the arguments, e.g., 9HEND POINT. Note that blank space and special characters are considered characters when used in alphameric fields.

5. The name of a Closed Function or another Subroutine subprogram (in which case an EXTERNAL statement is required).

The arguments presented by the CALL statement must agree in number, order, type and array size with the corresponding arguments in the SUBROUTINE statement of the called subroutine.

A subprogram may contain a DIMENSION statement for an array with variable dimensions. The actual dimensions of the array must be given in the calling program and passed along in the Subroutine CALL. (See Chapter 7.)

**EXTERNAL**

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>“EXTERNAL name, name,.....name” where name is the name of a Function or Subroutine</td>
<td>EXTERNAL SIN, COS, TAN</td>
</tr>
<tr>
<td></td>
<td>EXTERNAL BOB, JOE</td>
</tr>
</tbody>
</table>

Whenever a subroutine or function name appears in the argument list of:

1) a CALL
2) a function name in an expression
all such names must appear in an EXTERNAL statement. Example:
EXTERNAL SIN, A
CALL JOE (B, SIN)
C=F(X, A)
.
.
RETURN

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;RETURN&quot;</td>
<td>RETURN</td>
</tr>
</tbody>
</table>

This statement terminates any subprogram, whether of the type headed by a subroutine or a function statement, and returns control to the calling program. A return statement must, therefore, be the last executed statement of the subprogram; it can be any point reached by a path of control and any number of return statements may be used.

This statement should also be used to terminate normal execution of a main program. It causes a normal End-of-Job condition to be established within MCP (c.f. the stop statement, which causes an "Abnormal End-of-Job" return).

Chapter 7. Declarative Statements

Another type of FORTRAN statement is the declarative statement. Statements in this category are non-executable.

DIMENSION

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DIMENSION v, v, v, ... where each v is the name of a variable, subscripted with 1, 2, or 3 unsigned fixed point constants. Any number of v's may be given. In a SUBROUTINE or FUNCTION subprogram, the subscripts may be fixed point variables. (See Chapter 6.)&quot;</td>
<td>DIMENSION A(10), B (5,15), CVAL (3,4,5)</td>
</tr>
</tbody>
</table>

The DIMENSION statement provides the information necessary to allocate storage in the object program for arrays.

Each variable which appears in subscripted form in a program or subprogram must appear in a DIMENSION statement of that program or subprogram, or as a dimensioned COMMON entry. (See COMMON.) The DIMENSION or COMMON statement must precede the first appearance of that variable. The DIMENSION or COMMON statement lists the maximum dimensions of arrays; in the object program, references to these arrays must never exceed the specified dimensions.

The above example indicates that B is a two-dimensional array for which the subscripts never exceed 5 and 15. If B is a floating point variable, the DIMENSION statement causes 75 (i.e., 5 \times 15) storage locations to be set aside for the array B. A single DIMENSION statement may specify the dimensions of any number of arrays. A program must not contain a DIMENSION statement which includes the name of the program itself, or any program which it calls.

Adjustable Dimensions

The name of an array, and the constants which are its dimensions, may be passed as arguments when calling a Function or Subroutine subprogram. By this means, a subprogram may perform calculations on one or more arrays whose size is not known within the subprogram until it is called.

The following illustrates how this effect is obtained within a subprogram:

```
SUBROUTINE MATMY (... R, L, M ...)
  ...
END
DIMENSION ... R(L, M) ...
  ...
DO 100 I = 1, L
  ...
END
```

Note that a special form of the DIMENSION statement is required. The dimensions listed within parentheses are symbolic; they are not constants. Variables may be used as dimensions of an array only in a FUNCTION or SUBROUTINE subprogram. For any such array, the array name and all variables used as its dimensions must appear as arguments in the FUNCTION or SUBROUTINE statement at the head of the program in which such a DIMENSION statement occurs.

A subprogram, handling arrays with adjustable dimensions, need not be recompiled for each set of dimension sizes.

EQUIVALENCE

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;EQUIVALENCE (a, b, c, ...), (d, e, f, ...), ...&quot; where a, b, c, d, e, f, ... are variables optionally followed by a single unsigned fixed point constant in parentheses.</td>
<td>EQUIVALENCE (A, B(1), C(5)), (D(17)), E(3))</td>
</tr>
</tbody>
</table>

The EQUIVALENCE statement provides the option of controlling the allocation of data storage in the object program. In particular, when the logic of the program permits it, the number of storage locations used can be reduced by causing locations to be shared by two or more variables.

An EQUIVALENCE statement may be placed anywhere in the source program. Each pair of parentheses of
the statement list encloses the names of two or more quantities which are to be stored in the same locations during execution of the object program; any number of equivalences (i.e., sets of parentheses) may be given.

Quantities or arrays which are not mentioned in an EQUIVALENCE statement will be assigned unique locations. Locations can be shared only among variables, not among constants. EQUIVALENCE should not be used to establish mathematical equivalence.

**COMMON**

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;COMMON a, b, c.../r/d, e,</td>
<td>COMMON A, B, C/X/Q, R</td>
</tr>
<tr>
<td>f.../s/g,h,j...&quot; where a, b</td>
<td>/Y/M, P, O</td>
</tr>
<tr>
<td>... are dimensioned or non-di-</td>
<td>COMMON/Z/G, H, J / / D</td>
</tr>
<tr>
<td>mensioned variable names of</td>
<td></td>
</tr>
<tr>
<td>any type. /r/,/s/are symbols</td>
<td></td>
</tr>
<tr>
<td>conforming to FORTRAN</td>
<td></td>
</tr>
<tr>
<td>naming conventions.</td>
<td></td>
</tr>
</tbody>
</table>

The COMMON statement permits data storage area to be shared between programs in a way analogous to that by which EQUIVALENCE permits data storage sharing within a single program. Where the logic of the programs permits, this can result in a large saving of storage space.

The programmer has complete control over the locations assigned to the variables appearing in COMMON. The locations are assigned in the sequence in which the variables appear in the COMMON statements, beginning with the first COMMON statement of the problem.

**ARGUMENTS IN COMMON STORAGE**

Because of the above, COMMON statements may be used to serve another important function: as a medium by which to transmit arguments from the calling program to the called Function or Subroutine subprogram. In this way, they are transmitted implicitly rather than explicitly by being listed in the parentheses following the subroutine name.

To obtain implicit arguments, it is necessary only to have the corresponding variables in the two programs occupy the same location.

This can be obtained by having them occupy corresponding positions in COMMON statements of the two programs.

All elements placed in COMMON may be placed in separate blocks of COMMON, each block of which is located separately at object time. The block name is carried symbolically at load time so that blocks with the same name from different programs may be given the same location. The block name must not be used in any other reference in the program.

**COMMON BLOCK**

The symbolic name of a block is placed at the head of the variable names belonging to the block. It is always enclosed between slashes. There are two types of COMMON blocks: blank and labeled.

a. Blank COMMON is indicated as in FORTRAN II (no slashes), or by preceding the blank COMMON variables by two consecutive slashes ("/**").

b. Labeled COMMON is indicated by preceding the labeled COMMON variables by the block name between two slashes.

**RULES**

1. The field of entries pertaining to a block name ends with a new block name or the end of the COMMON statement.

2. Block name entries are cumulative throughout a program. A block name may appear more than once with entries for previous appearances.

3. Blank COMMON may be of any length. Labeled COMMON must conform to this size requirement: all COMMON blocks of a given name should have the same length in programs which are to be operated together.

**DIMENSION AND COMMON**

Dimension information may be included in the COMMON statement. In this case, it need not be included in a DIMENSION statement. Example:

```
COMMON A, B(10, 15), C.
```

**EQUIVALENCE AND COMMON**

Equivalence will never re-order COMMON, but may lengthen COMMON.

**Note 1:** In order to force correspondence in storage locations between two variables which otherwise will occupy different relative positions in COMMON storage, it is valid to place dummy variable names in a COMMON statement. These dummy names, which may be dimensioned, will cause reservation of the space necessary to cause correspondence.

**Note 2:** While implicit arguments can take the place of all arguments in CALL-type subroutines, there must be at least one explicit argument in a FORTRAN function. Here, too, a dummy variable may be used for convenience. When a variable is made equivalent to a variable which appears in a COMMON statement, the first variable will also be located in COMMON storage.

Variables brought into a COMMON block through equivalence statements may increase the size of the block indicated by the COMMON statements as in the following example:

```
COMMON A, B, C
DIMENSION D(3)
EQUIVALENCE (B, D)
```
The layout of core storage indicated by this example is:

A
B, D(1)
C, D(2)
D(3)

However, a variable may not be made equivalent to an element of an array in such a way as to cause the array to extend below the beginning of the COMMON block. For example:

COMMON A, B, C
DIMENSION D(3)
EQUIVALENCE (B, D(3))

This would cause the following impossible situation:

D(1)
A, D(2)
B, D(3)
C

Type Statements

There are three Type statements: LOGICAL, INTEGER, and REAL.

LOGICAL, INTEGER, AND REAL

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;LOGICAL a, b, c,...&quot;</td>
<td>LOGICAL F, G, L</td>
</tr>
<tr>
<td>&quot;INTEGER a, b, c,...&quot;</td>
<td>INTEGER Q, NAME</td>
</tr>
<tr>
<td>&quot;REAL a, b, c,...&quot;</td>
<td>REAL J, S</td>
</tr>
</tbody>
</table>

where a, b, c, ... are variable and/or function names appearing within the program.

A variable or function name, once declared as a certain type, will have that type throughout the program. Real and integer variables take arithmetic values; logical variables take the values "TRUE" or "FALSE".

A type declaration overrides the implicit declaration of any variable. For example, IOTA, which is normally an integer because it begins with I, can be declared a floating point variable by using the type declaration REAL IOTA.

If a variable does not appear in a Type statement, the variable will retain its implicit definition. That is, ALPHA is a floating point variable if it is not declared otherwise.

A declarative on a name must precede the first usage of that name in any executable statement.

Note that array specifications may not be given in these statements.

FREQUENCY

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;FREQUENCY n (i, j,...), m(k, l,...)&quot;</td>
<td>FREQUENCY 30(1, 2, 1), m(4(11), 50(1, 7, 1, 1))</td>
</tr>
<tr>
<td>...are statement numbers and i, j, k, l,...are unsigned fixed point constants.</td>
<td></td>
</tr>
</tbody>
</table>

The FREQUENCY statement is accepted by the IBM 7030 FORTRAN IV, but it has absolutely no effect on compilation. The syntax of the statement is checked, but in all other respects, the statement is ignored.

Chapter 8. Specification Statements

DATA

By means of the DATA statement, information may be obtained in the data areas and/or COMMON block of the object program to be loaded with the program. The DATA statement specifies the information and the variables to which they belong.

<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;DATA List (d1, d2/k/d3,...), List (d4, d5,...),...&quot;</td>
<td>DATA A (3HEND), (B(1), I=1, 15) (/5/4, 23)</td>
</tr>
<tr>
<td>where k appears before d and means that the d field is to be repeated k times.</td>
<td></td>
</tr>
<tr>
<td>DATA A(1.2), B(3.6), C(7.9)</td>
<td></td>
</tr>
</tbody>
</table>

Rules for Constructing the List

Variable names in the list follow the same rules as for constructing i-o lists with the following exceptions:

1. An array name may not appear unsubscripted.
2. Compiling of data into an array area must be under control of no-implying parentheses and associated parameters. The no-defining parameters must be constants.
3. If a variable appears with a subscript, not under control of no-implying parentheses, the subscript must be a constant.

Rules for Literals

The d literals may take any of three forms:

1. Integers. These are listed in the standard FORTRAN manner.
2. Floating point numbers. These are listed in the standard FORTRAN manner.
3. Alphabetic characters. (These may extend over more than one word.) nH precedes the alphabetic characters which are treated in the standard 7030 FORTRAN manner (eight per word, left justified, filled out with blanks, etc.). A repeat field preceding an alphabetic field covers the entire field.

With regard to the data list and its associated data: if the list is shorter than the data, the redundant data is ignored and, conversely, if the list is longer than the data the remaining items in the list are ignored.

Data may not be in blank COMMON. Data which is to be in a labelled COMMON block may be specified in a special separate program. This special program may contain only the DATA and COMMON statements, as well as any required EQUIVALENCE, DIMENSION and Type statements, and must be headed by the BLOCK DATA statement.
<table>
<thead>
<tr>
<th>GENERAL FORM</th>
<th>EXAMPLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;BLOCK DATA&quot;</td>
<td>BLOCK DATA</td>
</tr>
</tbody>
</table>

A program headed by this statement may contain only `DIMENSION`, `COMMON`, `Type`, `EQUIVALENCE`, and `DATA` statements, plus one `END` statement. Example:

```
BLOCK DATA
COMMON/D/A, B, C
DIMENSION B(2, 2)
INTEGER C
DATA ((B(I, J), I+1, 2), J=1, 2) (1.1, 1.2, 1.3, 1.4),
     C(16)
END
```
Part II. FORTRAN Programming for the IBM 7030

Chapter 9. First Principles

SUBPROGRAMS

The existence in the FORTRAN language of the three statements CALL, SUBROUTINE and FUNCTION enables a programmer to construct programs in a hierarchical fashion; that is, to form sequences of FORTRAN statements, these sequences being given unique names by means of which they may be concisely referred to in the construction of yet higher level code sequences. For any given problem, the construction of these sequences — Subroutine, or Function subprograms — terminates with the construction of the highest level sequence of all, the main program.

DEFINITION OF A JOB

It is possible to code these subprograms, not only in FORTRAN, but in lower level languages (e.g., STRAP) provided certain rules are followed. Hereafter, unless otherwise stated, the term “subprogram” will be taken to include main programs. Within the total collection of subprograms forming a machine procedure, some will have been proven error-free (“debugged”), some will be partially debugged, and some will be completely untested. Therefore, the total card deck submitted by a programmer to a machine operator most often consists of an arbitrary number of subprogram sub-decks, in a mixture of source languages. If the aim is not merely to compile the statement of the problem, but also to execute the resultant machine code, then the program deck may be followed by input data cards. The total collection of subprograms, plus data (if present), plus appropriate preceding control cards, comprises a job, in the sense described in the IBM Reference Manual, 7030 Master Control Program (Form C22-6678). If execution is not to be attempted at a particular time, then the compile-only job deck need not contain all the subprograms making up the total program, nor any data.

MCP ENVIRONMENT

The compilation process, and the ultimate execution of the generated machine code, are both under control of MCP.

RELOCATION, LINKAGES, AND COMMON STORAGE

As each subprogram is independently processed by the compiler, the resultant object code is not precisely in the form it will assume at execution time. At that time, it will have been combined with the other subprograms of the job, each requiring a region of memory whose size cannot be known at the time of compilation of all the others. For this reason, every compiled subprogram is produced with a standard “origin” (initial cell occupied by the program) of 00000.0; it is the function of the loader to relocate each binary subprogram to its own unique region of memory, just prior to execution. As an aid to this function, the compiler produces a card containing certain descriptive information about each subprogram (length of program, its name or names, etc.). This card, known as a FORTRAN Program Card, precedes the cards containing the object code.

THE LOADER

The loader also:

1. Sets up the appropriate linkages between subprograms,
2. Sets aside the amount of common storage required (storage to which any or all subprograms may have access, to facilitate communication between routines),
3. Supplies requested subprograms from the System’s binary library, and finally
4. Initiates execution by transferring control to the entry point of the main program.

Chapter 10. Program Environment

PROGRAM TERMINATION

A FORTRAN job for the 7030 may, in general, be schematically represented as in Figure 3.

![Figure 3](image)

The procedure has a single starting point, and many possible finishing points, the particular one chosen for any given machine run depending upon the parameters characterizing that run. All possible end points may be categorized in two ways — “good stops” and “bad stops.”
"Good Stops"

A successful termination is characterized by the emptying-out of any I-O buffers containing as yet unwritten information, the closing-out of any files on which the last activity was a WRITE, the writing of a tape mark on the system output tape, the release of all I-O equipment utilized by the job, and the issuance of appropriate messages to the operator. A successful termination may be achieved by any of the following:

1. The execution of a RETURN statement, in a main program.
2. An attempt to read data from the system input tape, when all data from that source has been exhausted.
3. Operator action following the execution of a PAUSE statement.

"Bad Stops"

An unsuccessful finish point is characterized by all the actions of the preceding section, but in addition, a memory dump is supplied on the system output tape. An unsuccessful finish is defined by any of the following:

1. The execution of a STOP statement, in a subprogram at any level.
2. An erroneous, illegal, or undefined I-O operation.
3. Operator action following the execution of a PAUSE statement.

Note that neither a (main program) RETURN, nor any STOP, actually causes the IBM 7030 to halt its activities; in both cases, control is returned to MCP. The only instruction capable of suspending machine activity indefinitely is the PAUSE operation.

FORTRAN Interrupt Handling

Like all programs operational under MCP control, a FORTRAN-compiled code runs with the 7030 "enabled." It will be recalled that prior to execution of a problem program, MCP has placed the required (MCP) interrupt table address in cell 2, and loaded the standard MCP mask (all maskable interrupts except IF off) into cell 12. Furthermore, the refill field of SX15 has been set to zero, thus indicating the absence of a Program Table of Exits. With respect to I-O interrupts, MCP expects the address of an I-O Table of Exits to be punched on each I0D card loaded, or, at worst, set up by a SCHEX pseudo-op before any activity on an I-O device.

I-O Interrupts

The implication of these facts, for FORTRAN programs, is as follows: All object program input-output is handled through the Input-Output Control Subroutine (actually a whole family of library subroutines linked in a fairly complex fashion). This subroutine will perform the necessary SCHEX operation for any given non-defined device, at the time that the first operation for that device is encountered in the running of the object program. The I-O Table of Exits, whose address is supplied to MCP in this fashion, is located inside the I0CS Subroutine, with the result that the various actions appropriate to the different possible returns can be taken, independent of the main-stream of the problem program.

Maskable Interrupts

Some further initialization of the maskable interrupt mechanism is required for FORTRAN object programs. The Stretch Compiling System inserts a call to an initializing routine as the first executable statement of each main program compiled. This results in the routine being fetched from the library at load time, and control being given to it before the true problem program gets under way. The routine replaces the contents of the Mask Register with the required FORTRAN mask, and places the address of a Program Table of Exits (if required) in the refill field of SX15. This table is itself contained in the initializing routine, as are any correction routines accessed through the table. It is possible for an installation to change this routine, if necessary.

In summary, no interrupt indicator of any kind is accessible to the FORTRAN-language programmer. It is, however, entirely possible to strap-code routines that reference the indicators, and to CALL such routines in a FORTRAN-program.

Internal Representation

The IBM 7030 FORTRAN IV compiler handles three basic kinds of data, in two different ways. So-called real numbers and integers are operated on by the operations of addition, subtraction, multiplication and division, while "logical quantities" may be manipulated by the operations of "and-ing," "or-ing," and "negation."

Real Numbers

A real number is represented internally by a normalized floating-point binary number, that is, a full-word containing a 12-bit signed exponent, a 48-bit fraction, a fraction sign, and 3 (unused) flag bits. The conventional floating point instructions of the IBM 7030 are used to manipulate such data.
INTBERS

A FORTRAN integer is actually represented internally as an unnormalized floating point quantity, with a fixed exponent of $38_{10}$. This enables the floating point instruction set to be used in manipulating these quantities, resulting in an appreciably shorter running time.

LOGICAL QUANTITIES

A logical quantity, as manipulated by a FORTRAN code, consists of a single bit, stored in bit 59 of a full word. Such quantities are operated on principally by connect instructions.

ZERO

The FORTRAN programmer should be aware that in the manipulation of normalized floating point quantities (real numbers), a number having a zero fraction part and an arbitrary exponent may be generated as a result of add and/or subtract operations. It is necessary, therefore, that the generated code be such that two numbers with zero fractions and different exponents be found equal under comparison operations. In the case of the FORTRAN integers, however, the fixed exponent of $+38_{10}$ is always preserved, including the case of zero.

PRECISION

Note that even though FORTRAN arithmetic is single-precision in nature, the generated code contains some double precision operations. These are necessary for such cases as the multiplication of two FORTRAN integers; by reason of their internal format, a single precision multiplication operation would yield an incorrect result.

Chapter 11. IOD Card Preparation

All programs to be executed under MCP control are required to state their I-O requirements (other than the system I-O devices) in terms of IOD cards, placed at the front of a job input deck. This enables MCP to determine, prior to the execution of a program, the devices required by a job, so that these devices may be made ready, and any necessary messages in connection with tape reel mounting may be given to the operator.

For a FORTRAN job, two important preliminary statements may be made concerning IOD cards. First, unlike a STRAP program, IOD's need not be present for a "compile-only" job. This arises from the fact that in a FORTRAN program there are no direct references to the symbols punched in the IOD name field. For a compile-go job, or a go job, they are required. If the FORTRAN section of a job was processed in a compile-no-go operation, then the necessary IOD cards (prepared in the manner explained later in this chapter) must be passed through the compiling system by themselves, before they may be combined with the FORTRAN object deck and run into the machine as a go-job. (See Example 3, Chapter 4.)

Second, MCP requires all the IOD's of a job to be in one group, at the front of a job deck. Since a typical FORTRAN job consists arbitrarily of many subprograms, the I-O requirements must be considered for the job in toto, not merely for each individual subprogram.

It is now possible to examine the 11 I-O statements of the FORTRAN language individually, and to determine their implications in terms of required IOD cards. To this end, it is convenient to divide the statements into four groups.

Group 1: Statements Not Requiring IOD's

The three statements of Group 1 are unique in that their use does not require the manufacture of IOD cards. They are:

- READ f, list
- PRINT f, list
- PUNCH f, list

where f is a format specification, and list is a list of the quantities to be transmitted. These three statements refer to the System reader, the System printer and the System punch, respectively.

Group 2: Binary I-O

The two statements in this group are:

- READ (u) list
- WRITE (u) list

where u is a fixed point integer or variable indicating the particular unit referenced. These statements are for the transmission of binary information to or from magnetic tape, or disk, the actual device required being determined by the related IOD cards.

The FORTRAN language is defined such that each distinct value of u implies a distinct I-O device, or, in
the case of the disk, a distinct region (sometimes referred to as a symbolic file). Therefore, for each actual or possible value that \( u \) may assume at object time, it is necessary to prepare an \( \text{IOD} \) card. Before examining the two possible types of \( \text{IOD} \) (tape or disk), it may be stated that in both cases, the correspondence between the \text{FORTRAN} statement or statements, and the related \text{IOD} cards, is established by punching the appropriate decimal value of \( u \) in the \text{IOD-name} field. Thus, if a given job contains the following statements:

\[
\begin{align*}
2 & \text{ READ (3) list} \\
4 & \text{ WRITE (5) list} \\
& \text{ DO 10 I=2, 10, 3} \\
10 & \text{ WRITE (1) list etc.}
\end{align*}
\]

It will be necessary to prepare a total of four \text{IOD} cards, defining units 2, 3, 5 and 8. (If, however, the parameters characterizing a particular run are such that statement 4 in the example above will not be executed, then the \text{IOD} card defining unit 4 may be removed before loading the job deck, thus avoiding the scheduling of unnecessary equipment for this run.) This requirement, that the \text{IOD-name} field be numeric, is the only \text{required} difference between \text{FORTRAN}-oriented \text{IOD}'s and \text{STRAP}-oriented \text{IOD}'s.

**General Rules for Constructing IOD Cards**

1. A binary-type \text{IOD} (disk, or binary tape) should contain the decimal value of unit, followed by a B, punched in the name field.

2. A decimal-type \text{IOD} (printer, decimal tape, or system "pseudo-IOD’s") should contain the decimal value of unit, followed by a D, punched in the name field.

3. If an \text{IOD-name} is seen, of which the first character is alphabetic, the compiler will assume that this \text{IOD} is for the benefit of a \text{STRAP} subprogram wishing to perform its own i-o operations. While the \text{IOD} will be passed through the system (and thus be assigned an i-o reference number equal to its relative placement in the \text{IOD-deck} excluding "pseudo-IOD’s"), it will be ignored by the \text{FORTRAN I0CS} routines, except for the insertion of the letter "K" before the \text{IOD-name}.

It will be possible for the system to accept a \text{DISK IOD}, without the terminal B in the name field; similarly, a \text{PRINTER}, or "pseudo"-\text{IOD} can be handled without a terminal D.

If a \text{TAPE IOD} with a pure integer in the name field is encountered, it will be assumed to be binary and the appropriate control mechanisms set up. If, at object time, the first reference to this device is decimal in nature, then the appropriate corrections will be made in flight. However, if the first operation is non-decimal, binary usage will be confirmed, and a subsequent decimal operation will cause job rejection.

**Binary Tape IOD's**

The tape \text{IOD} card format is shown in Figure 4.

![Figure 4](image)

(See the \text{MCP} manual for a detailed description of each field indicated.) A right-to-left drop-out is permitted on the fields as shown. A tape \text{IOD} card prepared by a \text{FORTRAN} programmer \text{may} have every field shown above punched (although it will become apparent that punching in the exit field is meaningless). Anything acceptable to \text{MCP} is acceptable to the \text{FORTRAN} compiler.

1. The exit field: For \text{STRAP} programs, the contents of this field indicate to the assembler the address of the appropriate i-o Table of Exits for this symbolic unit. However, \text{FORTRAN} object programs communicate with \text{MCP} via the \text{I0CS} routine, and the necessary Table of Exits is embodied therein. It is impractical for the programmer to determine where this (relocated) table will be at execution time; therefore, \text{I0CS} is so constructed that it will dynamically perform a \text{SCHEX} pseudo-op, at object time, just before the first reference to the file. Even if there is punching in the exit field, the compiling system will delete the field, and by \text{STRAP} time, it will be void. Therefore, it is suggested that the \text{FORTRAN} programmer never fill in this field.

2. The Channel and Unit Fields: An attempt by a \text{FORTRAN} programmer to associate more than one \text{IOD} with a given physical tape unit is an error. Therefore, to simplify the preparation of tape \text{IOD}'s for \text{FORTRAN}, \text{MCP} has been designed to assign distinct tape units to each tape \text{IOD} with both channel and unit void. Note that if it is desired to force two or more tape \text{IOD}'s to reference a given tape unit, the standard \text{MCP} procedure of punching identical non-void symbols in these fields may be followed. However, this is \text{not} usual for \text{FORTRAN}; these fields will most often be blank.

3. The Mode and Density Fields: Binary tapes, which are automatically blocked in the \text{FORTRAN} system, must be written in \text{odd} parity. As this is the installation standard in the \text{MCP} system as distributed, it is not necessary for the programmer to specify this mode on the \text{IOD} card. Note, however, that as the compiling system is not aware of the use to which the
unit will be put, even parity specification will not be rejected. If a binary READ or WRITE references this 10 in at execution time, the operation will be rejected and the job terminated. Similarly, the density field may be left void; this will result in a labeled tape being read in the density indicated by the label, while a labeled tape will be written, and an unlabeled tape written or read, in installation standard (high density).

4. The Disposition Field: If the assumption is made that the most frequent use of binary tapes is to provide temporary external storage to be used for the duration of a job, but not preserved thereafter, it may be said that usually this field may also be left void. A void disposition field implies that the tape is, under no circumstances, to be saved (removed from the machine). Once again, the programmer is at liberty to fill in this field in any of the ways specified in the MCE manual.

In summary, many binary tape usages can be satisfied by the simplest possible tape 10 (Figure 5).

Disk IOD's
A disk 10, in the most general case, has the form shown in Figure 6.

For identical reasons to those given for Tape 10's, the exit and channel fields may be omitted. The number field may be omitted, if the job for which this 10 is prepared requires only one disk file. In this case all of disk is available to problem programs. Thus, the single disk 10 required is as shown in Figure 7.

For more than one disk file, each 10 has the form shown in Figure 8. Note that an attempt to read or write beyond the limits of the disk file will result in an abnormal termination of the object program.

Group 3: Decimal I-O Statements
The two statements of Group 3 are:

READ (u, f) list
WRITE (u, f) list

where u is a fixed point integer or variable denoting the unit, and f is a format specification. These statements are for the transmission of decimal information, to or from magnetic tape, non-system printer, or, under special circumstances, the system card-reader, card punch, and printer. The specific device actuated depends upon the associated 10 cards.

Decimal Tapes
Magnetic tapes may be considered first. As in the case of binary tapes, for each possible object-time value of "u," a tape 10 must be prepared, with the decimal value of u punched in the 10 name field. Again, the exit, channel and unit fields may be void. The mode and density field requirements, though, are slightly different. A decimal BCD tape written through the Iocs output routine is not blocked; each WRITE statement creates a series of records, from 1 to 17 words in length, which are not only logical, but also physical records on tape. No control information is generated, or written, by the Iocs routines. The use of a decimal READ or WRITE, as opposed to a binary READ or WRITE, actually implies two things to the Iocs routines: first, that a conversion process must be applied to the data transmitted, and second, that the tapes concerned are not to be blocked. There are no requirements on the mode or density of the tape to be written (or read). The mode and density, in fact, are determined by the use to which this tape is ultimately put. If it is to be used by subsequent FORTRAN codes, running on the IBM 7030, or by the off-line IBM 1401 list and/or punch routines, any mode and any
density are permitted. Therefore, in this case, the FOR-
TRAN programmer may very well leave both fields void.
If the tape is to be taken outside the IBM 7030/1401
environment, however, it is probable that definite re-
quirements exist for these variables. For example, if
the tape is to be listed on the IBM 720 printer, then
parity must be even, and density low.

In summary, the TAPE IOD associated with a decimal
READ or WRITE may often be as simple as in Figure 9.
No more complex form can cause job rejection, either
at compile or object time.

\[
\begin{array}{c|c|c|c}
12 & 9 & 10 & 62 \quad 80 \\
\hline
8 \quad u & \text{IOD, TAPE} \\
\hline
\end{array}
\]

\text{u is a fixed point number}

Figure 9

**Non-System Printer**

The programmer may wish to force a generalized
WRITE statement to output material on the on-line, non-
system printer. In this case, an IOD must be prepared,
with the chosen value of \( u \) in the IOD name field, in the
format shown in Figure 10. For the usual reasons, the
exit and channel fields may be omitted.

\[
\begin{array}{c|c|c|c}
12 & 9 & 10 & 62 \quad 80 \\
\hline
8 \quad u & \text{IOD, PRINTER, exit, channel} \\
\hline
\end{array}
\]

\text{u is a fixed point number}

Figure 10

**System Devices**

To reference the system reader, punch or printer, by
the statements of this group, rather than by the three
statements specifically designed for this purpose, READ,
PUNCH, and PRINT, pseudo-IOD cards must be prepared.
They must be linked to the READ or WRITE statement
by means of the \( u \)-value in the name field. The format
of the IOD's is shown in Figure 11.

An attempt to reference a value of \( u \), defined by an
IOD of this type, by a binary READ or WRITE will cause
the job to be abnormally terminated.

**Group 4: Manipulative Statements**

The final group of statements in this area consists of
the three so-called manipulative statements:

\[
\begin{align*}
\text{END FILE} & \quad u \\
\text{BACKSPACE} & \quad u \\
\text{REWIND} & \quad u \\
\end{align*}
\]

where \( u \), as before, is a fixed point variable or literal.
It is unlikely that a program will use any of these
statements to reference a device not elsewhere referred
to by at least one of the statements of Groups 2 or 3.
Therefore, when these statements are considered in
context of a given program, an IOD card defining \( u \)
has probably already been prepared. If such is not
the case, however, then an appropriate IOD must be
created. Clearly, a value of \( u \) referenced in a REWIND
statement, for example, should not be defined by an
IOD PRINTER card; this gives trouble at object time.
Note, though, that inasmuch as the disk is treated
much like a binary tape, the operations of END FILE,
BACKSPACE, and REWIND are defined for disk. They
result in the writing of a pseudo-tape mark, the loca-
tion of the start of the previous logical record, and
the location of the first arc of the given disk file,
respectively.

**Summary**

The programmer is required to consider the collective
I-O requirements of all subprograms of his job, and
prepare an appropriate set of IOD cards, belonging, in
some sense, to the entire job. This set is liable to change
if any part of any subprogram is changed.

**Chapter 12. Control Cards, and
Deck Preparation**

A job, as presented to MCP and the Compiling System
on the system input tape, is defined and delimited by
certain control cards. This chapter describes the neces-
sary control cards, and the construction of the input
deck.

**Control Cards for Entire Job**

First, control cards govern the entire job. MCP requires
that the very first card of each job be a job card (Fig-
ure 12).
**JOB Card**

The job card serves to delimit this job from its predecessor. The contents of the job identification field are printed on the system output tape, for logging purposes.

**TYPE Card**

The second card of every FORTRAN job deck must be a TYPE card, punched as shown in Figure 13.

![Figure 13](image)

**Control Cards for Each Subprogram**

As mentioned earlier, the typical FORTRAN job consists of many subprograms, in various levels of language (including binary, precompiled subprograms). Moreover, there will usually be a set of 100 cards relevant to the job as a whole, and sometimes a deck of punched data cards. For the purpose of describing subprogram control cards, it is convenient to treat both the 100-deck and the data-deck (if present) as two more subprograms. Therefore, if any subprogram of a job requires compilation, then every subprogram of that job must be immediately preceded by a SUBTYPE card, whose general form is shown in Figure 14.

![Figure 14](image)

**Deck Description**

The deck description field indicates the language level of the following subprogram deck, and must be one of the following:

- **FORTRAN** — FORTRAN IV source language
- **AI** — SMAC source statements
- **STRAP** — STRAP II code
- **BIN** — Pre-assembled relocatable binary deck(s)
- **DATA** — Data deck
- **FIOD** — Unprocessed FORTRAN-oriented IOD deck

An unrecognizable field or a null field will cause the processor to skip to the next SUBTYPE card, and to cancel any CO option selected by the TYPE card at the start of this job.

**List Option**

The list option field may be:

- **LSTRAP** — list-generated STRAP code
- **null** — list generated STRAP
- **NOLIST** — do not list generated STRAP

Clearly, LSTRAP in this field is meaningless if the deck description is BIN or DATA.

**Punch Option**

The punch option field may be either

- **PBIN** — punch binary cards
- **NOPUNCH** — do not punch binary cards
- **null** — equivalent to PBIN

Again, it is clear that PBIN (and null) are meaningless on a SUBTYPE card whose deck description field contains BIN or DATA.

**Exception for Consecutive Binary Decks**

The rule stated above—that every subprogram deck in a COMPIL or COMPILGO job must be preceded by an appropriate SUBTYPE card—may be relaxed in the case of a job containing two or more pre-compiled binary decks placed contiguously in the input deck. The second and subsequent binary decks may have their SUBTYPE's omitted. Thus, after the SUBTYPE preceding the first binary deck, the next SUBTYPE seen (if any) is that preceding the next non-binary section of the deck.

**IOD Subprogram First**

It was mentioned above that the entire collection of 100 cards for a given job (plus any related REEL cards) must be collected together, and treated as a subprogram. This concept is pursued to the extent that it is necessary to follow the collection with an END statement, and to precede it with a SUBTYPE card (deck description FIOD). *This subprogram must be the first of a job; there is no exception to this rule.* The END must be punched beginning in column 10.
SUBTYPE FOR PROCESSED IOD SUBPROGRAM

After a job containing such a subprogram has been compiled, note (provided binary decks have been requested from the system) that IOD's are still present, even though all other decks now appear as column binary cards. These IOD's now have two extra fields punched; this has been done by STRAP, as described in the MCP manual. The cards are at the front of the binary deck, and may not be removed from this position on any subsequent occasions in which the deck is presented to the IBM 7030. However, these processed IOD's are to be regarded as binary input — i.e., the preceding SUBTYPE card must have the deck-description field as BIN, if this binary deck is combined with some additional non-binary decks to form a (new) compile, or, more probably, a COMPILCO job.

DATA SUBPROGRAM

Any data to be input on the System input tape by the execution of a program must follow all subprogram decks. There is no exception to this rule. If the job is COMPILE, or COMPILCO, then the data deck must be preceded by a SUBTYPE card, with deck description data. If the job is a (FORTRAN) GO, then data must be separated from the last binary subprogram by a FORTRAN Branch card (see Appendix A).

GO JOBS

When a job requires no compilation whatsoever (i.e., all input consists of binary precompiled programs), then SUBTYPE cards are not needed. The presence of the TYPE, GO, FORTRAN cards will invoke the FORTRAN relocating loader. If input data is to follow, then the last subprogram must be followed by a FORTRAN Branch card.

Examples

A few sample input decks are illustrated below to indicate the various control cards required.

Example 1

A job consisting of two FORTRAN and one SMAC subprograms, to be compiled and executed, plus decimal data. No binary decks required, but STRAP listings requested.

T SUBTYPE, A1, LSTRAP, NOPUN
    ----        SMAC subprogram
    ----
END
T SUBTYPE, FORTRAN, LSTRAP, NOPUN
    ----        FORTRAN subprogram 2
    ----
END
T SUBTYPE, DATA
    ----        Data Cards
    ----

Example 2

A job consisting of a FORTRAN subprogram: two binary decks, a STRAP deck, and a processed IOD deck. (This implies the FORTRAN and STRAP subprograms do not have any additional 1-o requirements not known at the time the IOD deck was made up.) No STRAP listings are required, but binary decks are to be obtained. Execution is to be attempted.

B JOB, RESEARCH PROB. 0001. CIRCLE, (SQUARING OF)
B TYPE, COMPILCO, FORTRAN
T SUBTYPE, BIN
B IOD, -----
B IOD, -----

Binary deck produced by IOD compilation

T SUBTYPE, BIN
Binary Subprogram 1

T SUBTYPE, BIN
Binary Subprogram 2

T SUBTYPE, STRAP

STRAP subprogram

T SUBTYPE, FORTRAN
FORTRAN subprogram

END

Example 3

A job consisting of one binary subprogram, and an unprocessed IOD subprogram. No listings, but binary cards are required. Execution to be attempted with following data:

B JOB, STATISTICAL EVALUATION OF SWINE-FEVER INCIDENCE IN AKRON.
B TYPE, COMPILCO, FORTRAN
T SUBTYPE, FIOD
B IOD, -----
B IOD, -----

END
T SUBTYPE, BIN
Binary Subprogram

T SUBTYPE, DATA
Data deck

Example 4
A straight go job, with two previously compiled binary decks, and a processed IOD-deck.

B JOB, ORBITAL CALCULATION 376. (VULCAN)
B TYPE, GO, FORTRAN
B IOD ----
B IOD ----
-----
Binary deck produced by IOD compilation

Binary Subprogram
1

Binary Subprogram
2

FORTRAN Branch Card
Data deck

Chapter 13. STRAP Subprograms and Binary Decks

Provided certain rules are followed, it is possible to write subprograms (either Function-type, or Subroutine-type) in strap, that may be continued with FORTRAN-coded routines to form a job. This chapter explains these rules after first reviewing the constitution of a typical binary subprogram deck.

Binary Deck
The usual binary deck consists of the following cards, in the order given:
1. A FORTRAN Program card.
2. One or more Common Definition cards.
3. An arbitrary number of Relocatable Instruction cards, and/or Relocatable Data cards.

Program Card
The FORTRAN Program card provides certain descriptive information about the subprogram to the loader. After the routine has been loaded into memory and during execution, the Program card occupies no space. The information carried by this card is:
   - The number of full words of memory required by this routine.
   - The number of full words of memory required by this routine.
   - The “name” of this routine.
   - The entry point, relative to zero, associated with (c).

A STRAP subprogram, unlike a FORTRAN-coded routine, may have more than one entry point. For such routines, items (c) and (d) are repeated as many times as required.

Common Definition Card
A Common Definition card informs the loader of the names and sizes of the named common blocks utilized by this program, of which there may be arbitrarily many. For each such block, the information given is:
   (a) Common block name.
   (b) Total number of full words required for this block.

If a given subprogram references more common blocks than can be defined on one common definition card, then as many more additional cards of this type as are necessary may follow. There is no difference in formats between the first and any “continuation” common definition cards; however, it is vital that the original ordering of these cards (i.e., as produced by the Compiling System) be preserved.

Relocatable Instructions and Data Cards
Relocatable binary instruction and relocatable binary data cards contain the executable instructions of a subprogram and its associated data, respectively. The instruction card also contains a relative loading origin for the card, plus relocation bits, specifying how the individual instruction address references are to be relocated. The data card contains the data bits, a loading base (which specifies whether the data is to be loaded into the program region of memory, or into a named common block), and a relative loading origin (with respect to the loading base).

Compatibility Requirements
STRAP subprograms must observe certain rules in connection with the production of:
1. The FORTRAN Program Card.
2. Any necessary Common Definition cards.
Moreover, certain conventions must be followed in connection with the following additional points:
3. The calling of lower level routines (if any).
4. The acquisition of input arguments (if any).
5. Index register saving and restoring.
6. The production of output results (if any).
7. The returning of control to the calling routine.

Production of a Program Card
A FORTRAN Program Card may be obtained by the following sequence of instructions:

FOR SCR, S, C
Name1 ENTER, address1
Name2 ENTER, address2
-----
Name3 ENTER, addressn
where:

- **S** defines the number of full words occupied by this program (equal to the highest location relative to zero used, plus 1).
- **C** defines the number of full words of blank common used by this program.

**Name** is the *i*th entry point name to be placed on the program card.

**Address** defines the actual entry point to be associated with name.

If more entry points are defined by means of the **ENTER** statements than can be punched on one card, then **STRAP** will automatically produce as many additional program cards as are required.

**Production of Common Definition Cards**

Any Common Definition cards that may be required can be produced by the following sequence of **STRAP** statements:

```
PUNCD
Name1: COMBLOCK, Q1
Name2: COMBLOCK, Q2
NameN: COMBLOCK, Qn
```

where:

- **Name1** is the name of the *i*th common block defined for this program.
- **Q1** is the number of full words to be reserved for this common block.

**Calling Lower Level Routines; Transfer Vector**

Any lower level routines to be referenced by a given subprogram, must be invoked in a standard fashion. The formal instructions passing control must be:

```
SIC, $X15
B, name
```

where **name** is the address of a word in the transfer vector, at the start of the program. This transfer vector is a consecutive series of full words, containing the A-8 coded names of the lower level routines to be called, beginning in cell 1 of the program (relative to zero). The first 18 bits of cell zero must contain an integer, specifying the length of this transfer vector. Thus, to call a lower level routine named **GLOB**, the following code should be used:

```
PUNREL
SLC, 0.0
XW., ENDTV-BEGINTV
BEGIN TV
SYN, $  
(A*) DD(BU, 64,8), GLOB
```

**ENDTV**

```
SYN, $ Executable Code
```

SIC, $X15

```
B, GLOB
```

(Note that this example does not show the passing of arguments into **GLOB**.)

**Argument Lists**

The **STRAP** subprogrammer is concerned with argument lists in two connections. First, input arguments to a subprogram must be acquired by means of the argument list generated by the calling program; second, if a lower level routine is to be called, an appropriate argument list must be **constructed** in **STRAP**.

A subprogram list consists of a consecutive series of full words, one for each argument, immediately following the invoking "**sic; b**" instructions. Each word is in the form:

```
VF, a1; VF, a2
```

where:

- **a1** is the location of the argument value (in the calling program)
- **a2** is either identical to **a1**, if the argument is a single valued variable, or the first location of an array, a particular element of which is being specified by **a1**.

Thus, **a1** is the address to be used in the acquisition of arguments by the called program. If the argument specified in the calling program is the name of an array, without any appended subscript, then **a1** and **a2** are identical, the address given being that of the first cell of the array, or **ARRAY** *(1, 1, 1)*. If the argument is alphanumeric data, then both addresses given are equivalent, and specify the *first* location occupied by that data.

**Example of Argument List**

**FORTRAN** code

```
DIMENSION A(2, 3, 4), B(10, 10)
CALL JOE (6.7, Q, 3HEND, B, A (1, 2, 2))
```

Generated **STRAP** code (for **CALL** statement)

```
SIC, $X15
B, TVLOC
VF, L(6.7); VF, L(6.7)
VF, Q; VF, Q
VF, HOLARG; VF, HOLARG
VF, B; VF, B
VF, A+8.0; VF, A

...... return point
......

HOLARG (A*) DD(BU, 8), END
```

**Index Register Saving and Storing**

A **FORTRAN**-coded subprogram always saves all 16 index registers upon initial entry to that routine. At each possible exit point, arrangements are made to restore these registers. While it is not strictly necessary for the **STRAP** subprogrammer to save and restore all 16 index
registers, it is imperative that any registers liable to be changed should be preserved. It may be convenient for this to be done in a similar fashion to that in which FORTRAN-coded routines perform this action; here, 16 memory words are set aside, following the transfer vector (if any) for the preservation of the index register contents.

Results
A Function subprogram must always place its (single) result in the left part of the accumulator ($L$). A Subroutine subprogram must place each of its results in a memory location. (Note that it is not required that a Subroutine produce any results.) A result represented by the n-th argument of a CALL statement is to be stored in the location specified by the first 18 bits of location (n-1).0($X15$).

Return of Control
Control may be returned to the calling routine by:
1. Restoring any altered index register.

2. Branching to the location immediately following the last word in the argument list (if present).
The appropriate instruction for a routine with n arguments is:
B, n.0($X15$)

Array Storage
Note that data is stored in a forward direction in programs compiled by the IBM 7030 FORTRAN. Given an array A(2, 3, 4) the memory cells allocated are:

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>A(1, 1, 1)</td>
</tr>
<tr>
<td>X+1</td>
<td>A(2, 1, 1)</td>
</tr>
<tr>
<td>X+2</td>
<td>A(1, 2, 1)</td>
</tr>
<tr>
<td>X+3</td>
<td>A(2, 2, 1)</td>
</tr>
<tr>
<td>X+23</td>
<td>A(2, 3, 4)</td>
</tr>
</tbody>
</table>

An array is stored forward by columns. Furthermore, the various items in a common block (including blank common) are also stored in a forward direction. That is, given the statement
COMMON A, B, C, D
A occupies a lower memory address than B, C, or D.
Appendix A. Binary Card Formats

FORTRAN Program Card

<table>
<thead>
<tr>
<th>ID</th>
<th>SEQUENCE NUMBER</th>
<th>CHECKSUM</th>
<th>UNUSED</th>
<th>PROGRAM SIZE</th>
<th>AMOUNT OF BLANK</th>
<th>COMMON SIZE</th>
<th>FIRST POINTER NAME (1B - 64 BITS)</th>
<th>ENTRY NAME (12 BITS)</th>
<th>SECOND ENTRY NAME</th>
<th>ENTRY ADDRESS</th>
<th>ETC. ACROSS THE CARD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>13</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

Columns 6, 7 and 8 are blank on any subsequent continuation program cards.

Common Definition Card

<table>
<thead>
<tr>
<th>ID</th>
<th>SEQUENCE NUMBER</th>
<th>CHECKSUM</th>
<th>UNUSED</th>
<th>COMMON NAME</th>
<th>COMMON SIZE</th>
<th>ETC. ACROSS THE CARD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Relocatable Binary Data Card

<table>
<thead>
<tr>
<th>ID</th>
<th>SEQUENCE NUMBER</th>
<th>CHECKSUM</th>
<th>BIT COUNT</th>
<th>ORIGIN</th>
<th>SECONDARY BIT COUNT</th>
<th>LOADING BASE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

If the number of named commons is greater than that which will fit on a single card, additional cards will be punched in the same format.

Relocatable Binary Instruction Card

Relocation Bits (describing a half-word field)

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No relocation</td>
</tr>
<tr>
<td>1 . .</td>
<td>Relocation</td>
</tr>
<tr>
<td>1 0 0</td>
<td>First 18 bits (address)</td>
</tr>
<tr>
<td>1 0 1</td>
<td>Last 18 bits (refill)</td>
</tr>
</tbody>
</table>

No relocation bits on this card.

Loading Base (column 10)

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Program Data</td>
</tr>
<tr>
<td>1</td>
<td>1st Named Common</td>
</tr>
<tr>
<td>2</td>
<td>2nd Named Common, etc.</td>
</tr>
</tbody>
</table>

Secondary Bit Count (columns 9-10)

Bits to be zeroed/skipped before/after loading as determined from 5.0, 5.1:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.0</td>
<td>0-skip 1-zero</td>
</tr>
<tr>
<td>5.1</td>
<td>0-before 1-after</td>
</tr>
</tbody>
</table>
Appendix B: Principal Differences Between the IBM 709/90 FORTRAN II and the IBM 7030 FORTRAN IV

The differences between FORTRAN II and FORTRAN IV may be considered under two headings: changes to existing facilities, and addition of new features.

Changes

1. The IF statement has been redefined, to reflect a logical decision making process, rather than an arithmetic sign test, although the IBM 7030 FORTRAN IV compiler will accept the arithmetic-if without flow analysis.

2. The I-O tape-oriented statements have been changed, to make them more general.

3. The machine dependent statements referencing sense lights, sense switches, and machine triggers have been dropped.

4. The EQUIVALENCE statement is no longer permitted to re-order COMMON storage.

5. Function naming rules have been changed, to unify the various conventions applicable to the different kinds of functions.

6. The B and F modal punch conventions have been dropped.

7. Implicit multiplication is not accepted.

Additions

1. Three type declarations have been added to the language to enable the nature of variable and function names to be (optionally) declared rather than established by declension.

2. Logical and relational operators have been added, to facilitate decision making processes.

3. A special statement, EXTERNAL, has been provided to permit the passing of subprogram names as arguments.

4. The concept of "named" common storage has been introduced, together with the ability to specify dimension information in COMMON statements.

5. The ability to write subprograms operating upon arrays of varying size specified by means of a variable DIMENSION statement, has been added.

6. A new statement, DATA, has been added, to permit initialization of data objects at compile time.

Appendix C: A STRAP Coded Subprogram

Suppose the programmer wishes to code a STRAP subprogram analogous to the following FORTRAN program:

```fortran
SUBROUTINE STRAP
COMMON ANAME1(10)
COMMON/BLOCK1/ANAME2,
BLOCK2/ANAME3(10)
ANAME2 = SIN(0.0)
DO 1 I = 1, 10
ANAME3(I) = I
ANAME1(I) = COS(ANAME3(I))
1 CONTINUE
RETURN
END
```

then the STRAP program should be similar to the following:

```fortran
PUNREL
SEM, 92, 114, 116
PUNFPC, LAST, COMMON
ENTER, ENTER
PUNCDC

STRAP
BLOC1 COMBLOC, BLOCK1ND
BLOCK2 COMBLOC, BLOCK2ND
XW, 2
COS (* (A*) DD(BU), COS
SIN (* (A*) DD(BU), SIN
ENTER
SX, $0, SAVE
SX, $1, SAVE+1.0
SX, $15, SAVE+2.0
CNOIP
LVI, $15, $+1.0
B, SIN
VF, ZERO
VF, ZERO
ST(U), ANAME2
LVI, $1, $0
L(U), ONE
ST(U), I
LCI, $0, 10

BEGIN1
L(U), I
ST(U), ANAME3($1)
LVI, $0, $L+0.32
V+, $0, ARG1+0.32
SV, $0, ARG1
CNOIP
LVI, $15, $+1.0
B, COS
ARG1
VF, ANAME3
VF, ANAME3
ST(U), ANAME1($1)
V+I, $1, $1.0
L(U), ONE
M+U), I
CB, $0, BEGIN1
LX, $0, SAVE
LX, $1, SAVE+1.0
LX, $15, SAVE+2.0
```
Storage for a segmented program will be allocated in the following way: One segment, called the primary segment, will remain in storage at all times. This segment must contain the main subprogram, to which control will be initially transferred. All library subprograms which are called, as well as the link subprogram, will be assigned by ass to the primary segment. The user must assign the ion subprogram to the primary segment. Commons will be allocated space higher in storage than the program, but commons will be properly associated with the segments which reference them. Block data will be associated with the segment which contains that block data subprogram.

Appendix D: Segmentation on the IBM 7030

Segmentation is a method of overlay designed to handle the situation in which the entire problem program is too large to be held in internal storage at one time. As a part of the FORTRAN chain, the 7030 segmentation is implemented through the BSS-MCP loader. Internally, the segmentation portion of BSS will set up the relocation tables the information required by the MCP loader to load the segmented program in the desired fashion. A special link program operates during execution of the problem program to determine when a new portion of the program program is to be loaded and to transfer control to the MCP loader at that time.

Segmentation is used in the following manner: a segment is defined by placing in front of the binary deck for a group of subprograms a node control card, of the card format given below. No node card is needed for a program consisting of a single segment.

All of the subprograms defined as a segment by a node card will be loaded into storage whenever any part of that segment has to be loaded. Since a certain amount of time will be required during execution to bring in the various segments, the most efficient deck arrangement for segmentation will be that which requires the smallest number of transfers between segments.

Node Card Format

The node control card, which indicates to ass whether segmentation is to be used and how the segments are divided, has the following format:

```
col  1   T
col 10-19   NODE, XX, YY
```

XX is the two-digit segment number for the group of subprograms preceding the next node card. YY is a two-digit segment number of the segment to which the current segment, XX, is to be attached. The value of YY should be 00 on the node card for the primary segment.

Notes

1. The number of segments cannot exceed 99.
2. Care must be taken in setting up the segmentation deck. A call from one segment to another will be illegal if the call causes the originating segment to be overlaid.
3. Neither named common nor block data will be preserved when a segment associated with it is overlaid.
4. An error will be indicated if any group of segments which are to be in storage at the same time will exceed the amount of storage available.
5. The number of segments which are to be in storage at the same time cannot exceed eight.
6. FORTRAN subprograms must be compiled before node cards are inserted in the program deck.