File No. GENL-25
Form C24-3322-2

IBM Systems Reference Library

Fortran IV Language Specifications, Program Specifications, and Operating Procedures IBM 1401, 1440, and 1460

Programs: 1401-FO-051 (Disk Resident System)
1401-FO-052 (Tape Resident System)

This reference publication contains the language specifications, program specifications, and operating procedures for the Fortran IV Programming System.

The language specifications describe the Fortran IV language that is processed by the Fortran system. The language closely resembles the language of mathematics, and includes various types of arithmetic, control, input/output, and specification statements.

The program specifications describe the two programs, *System Control* and *Fortran Processor,* that make up the Fortran system. Logical files defined and used by the system, control cards, and results of processing operations are also included.

The operating procedures are divided into two parts. The first part describes compiling and executing object programs, changing logical-file assignments, and maintaining a Fortran library of subprograms. The second part describes building and updating a Fortran system.

A summary of processor jobs, control-card formats, phase descriptions, and a listing of a sample program make up the appendix of this publication. Also included in the appendix is a description of the procedures to be followed in building a system that contains both Fortran and Autocoder.

For a list of other publications and abstracts, see the IBM *1401 and 1460 Bibliography,* Form A24-1495, and the IBM *1440 Bibliography,* Form A24-3005.

Major Revision (April 1966)

This publication, C24-3322-2, is a major revision of, and obsoletes the previous publication, C24-3322-1, and its associated Technical Newsletter N21-0050. Changes have been made throughout this publication.

Copies of this and other IBM publications can be obtained through IBM Branch Offices.
Address comments concerning the content of this publication to:
IBM, Programming Publications Department, Rochester, Minn. 55901.

Contents

This publication contains the language specifications, program specifications, and operating procedures for the Fortran IV programming system for IBM 1401, 1440, and 1460. In this publication, the term *Fortran system* refers to *1401/1440/1460 Fortran IV*, program numbers 1401-FO-051 (Disk Resident System) or 1401-FO-052 (Tape Resident System).

This publication is divided into three major sections, *language specifications, program specifications,* and *operating procedures.*

The language specifications section describes the coding of a Fortran program. The content of this section is presented with the assumption that the programmer is familiar with the information in the *Fortran General Information Manual,* Form F28-8074.

The program specifications section describes the Fortran system. Included in the section are such topics as a description of the *System Control Program* (the controlling element of the Fortran system), a description of the *Fortran Processor Program,* and a detailed description of the results of system operations. Although this section is directed primarily to the programmer, the machine operator should review the section for an understanding of the system.

The third section, operating procedures, contains such topics as preparing processor jobs, changing file assignments for processor jobs, and running processor jobs. The last part of the section outlines the procedures to follow in building a Fortran system. For the convenience of both programmer and machine operator, all control cards are summarized in *Appendix I.*

While the third section is directed primarily to the machine operator, it is recommended that the programmer review the content of the complete section. The programmer should particularly note the parts of the section dealing with preparing processor jobs and changing file assignments.

Related Information

The following Systems Reference Library publications contain additional information relating to the use of the Fortran system. It is recommended that these publications be available to the user for reference purposes.

Fortran General Information Manual, Form F28-8074.

Disk Utility Programs Specifications for IBM 1401, 1440, and 1460 (with 1301 and 1311), Form C24-1484.

Disk Utility Programs Operating Procedures for IBM 1401 and 1460 (with 1301 and 1311), Form C24-3105, or *Disk Utility Programs Operating Procedures for IBM 1440 (with 1301 and 1311),* Form C24-3121.

Definitions of Key Terms

In order to clarify the meaning of special terms when used in this publication, the following definitions are given. Standard terms are defined in *Glossary of Information Processing,* Form C20-8089.

Card Boot. A card deck, supplied as part of the Fortran system program deck, that is used to start system operations.

Compiler. The program that translates Fortran symbolic statements directly into relocatable machine language. This process is called a *compilation.*

Job. An operation or sequence of operations that are to be performed by the Fortran system.

Logical Files. Input/output devices and/or areas that are used by the Fortran system.

Object-time. A term describing those elements or processes related to the execution of a machine-language object program.

Operation. A basic unit of work to be performed by one of the components of the system.

Stack. A set of one or more jobs that is to be processed during the same machine run.

System. The set of programs made up of the elements required for compiling and/or executing user-programs.

[ ] Brackets contain an option that may be chosen. Braces contain options, one of which must be chosen.

Machine Requirements

To process a Fortran source program, the following minimum machine configurations are specified.
An IBM 1401 system with:

12,000 positions of core storage
Advanced-Programming feature

High-Low-Equal-Compare feature

One IBM 1311 Disk Storage Drive, or four IBM 729 Magnetic Tape Units, or four IBM 7330 Magnetic Tape Units, or a combination of four IBM 729 Magnetic Tape Units and IBM 7330 Magnetic Tape Units

One IBM 1402 Card Read-Punch

One IBM 1403 Printer, Model 2

An IBM 1440 system with:

12,000 positions of core storage

Indexing-and-Store-Address-Register feature

One IBM 1301 Disk Storage, or one IBM 1311 Disk Storage Drive

One IBM 1442 Card Reader

One IBM 1443 Printer

An IBM 1460 system with:

12,000 positions of core storage

Indexing-and-Store-Address-Register feature

One IBM 1301 Disk Storage, or one IBM 1311 Disk Storage Drive, or four IBM 729 Magnetic Tape Units, or four IBM 7330 Magnetic Tape Units, or a combination of four IBM 729 Magnetic Tape Units and IBM 7330 Magnetic Tape Units

One IBM 1402 Card Read-Punch

One IBM 1403 Printer, Model 2

To load and execute object programs generated by the Fortran system, the following minimum machine requirements are specified.

An IBM 1401 system with:

12,000 positions of core storage (or more if required by the object program)

Advanced-Programming feature

High-Low-Equal-Compare feature

One IBM 1311 Disk Storage Drive, or one IBM 729 Magnetic Tape Unit, or one IBM 7330 Magnetic Tape Unit (for residence of the Fortran system, including the library of the relocatable subprograms)

One IBM 1402 Card Read-Punch

One IBM 1403 Printer, Model 2

Sense switches, if required by the object program

A program loading device, which could be an IBM 1402 Card Read-Punch (the same as previously required), an IBM 1311 Disk Storage Drive (may be the same as that required for system residence), an IBM 729 Magnetic Tape Unit (must *not* be the same unit as that required for system residence), or an IBM 7330 Magnetic Tape Unit (must *not* be the same unit as that required for system residence)

Additional input/output devices as required by the object program

An IBM 1440 system with:

12,000 positions of core storage (or more if required by the object program)

Indexing-and-Store-Address-Register feature

One IBM 1301 Disk Storage, or one IBM 1311 Disk Storage Drive (for residence of the Fortran system, including the library of the relocatable subprograms)

One IBM 1442 Card Reader

One IBM 1443 Printer

Sense switches, if required by the object program

A program loading device, which could be an IBM 1442 Card Reader (the same as previously required), an IBM 1301 Disk Storage or an IBM 1311 Disk Storage Drive (may be the same as that required for system residence)

Additional input/output devices as required by the object program

An IBM 1460 system with:

12,000 positions of core storage (or more if required by the object program)

Indexing-and-Store-Address-Register feature

One IBM 1301 Disk Storage, or one IBM 1311 Disk Storage Drive, or one IBM 729 Magnetic Tape Unit, or one IBM 7330 Magnetic Tape Unit (for residence of the Fortran system, including the library of the relocatable subprograms)

One IBM 1402 Card Read-Punch

One IBM 1403 Printer, Model 2

Sense switches, if required by the object program

A program loading device, which could be an IBM 1402 Card Read-Punch (the same as previously required), or an IBM 1301 Disk Storage or an IBM 1311 Disk Storage Drive (may be the same as that required for system residence), or an IBM 729 Magnetic Tape Unit (must *not* be the same unit as that required for system residence), or an IBM 7330 Magnetic Tape Unit (must *not* be the same unit as that required for system residence)

Additional input/output devices as required by the object program

The Fortran system can use the following devices, if available.

IBM 1444 Card Punch

IBM 1447 Console without a buffer feature

The 1401, 1440, and 1460 Fortran Programming System consists of a language and its associated processor program. The Fortran language enables the programmer to code programs that deal with problems that are primarily mathematical in nature. Problems containing formulas and variables can be dealt with easily by using the Fortran language. A facility is included with the disk resident Fortran system to process information from randomly accessible records.

In addition to the main program, the user can code and use subprograms. These subprograms can be called for and used by the main program and/or other subprograms.

The Fortran language comprises five general categories of statements.

Arithmetic Statements. The arithmetic statements define the value of a variable to be the result of a numerical or logical calculation.

Control Statements. The control statements govern the flow of control in the program.

Input/Output Statements. Input/output statements specify the transfer of information between the program environment, the main-computer storage, and the extra-program environment, input/output devices, such as a card reader, a card punch, a printer, a console printer, a magnetic tape unit, or a disk unit.

Subprogram Statements. Subprogram statements permit the programmer to define subprograms for subsequent use.

Specification Statements. Specification statements declare properties of names used in the program and permit the user to exert some control over the allocation of core storage for program variables.

Any of these statements can be assigned a statement number. To permit reference within one statement to another statement, the latter statement must be assigned a statement number. Superfluous statement numbers will adversely affect compiling time.

The Fortran processor translates (compiles) the programs written in the Fortran language into machine-language object programs in the relocatable format. Object programs are then executed under control of the Fortran system.

Constants, Variables, Subscripts, and Expressions

This section describes constants, variables, and subscripts used to express 1-, 2-, and 3-dimensional arrays of variables. Also included in this section is a discussion of expressions, the combinations of constants, variables, and function references.

Constants

Three types of constants are permitted in a Fortran source program: integer (fixed point), real (floating point), and logical.

Integer Constants

General Form. An integer constant consists of n decimal digits, where $1 \leq n \leq k$, written without a decimal point.

Examples.

1
2
524267

Value of k

The value of k (precision) can be indicated to the Fortran compiler through control information supplied by the user. If k is specified by the user, the value of k must be $1 \leq k \leq 20$. If k is not specified by the user, the compiler uses k equal to five digits. No more than k digits can be written.

The absolute value of an integer constant must be between 0 and $(10^k - 1)$.

Real Constants

General Form. A real constant consists of n decimal digits, where $1 \leq n \leq f$, written with a decimal point. A real constant can be followed by a decimal exponent written as the letter E followed by a (signed or unsigned) 1- or 2-digit integer constant.

Examples.

17.	
5.0	
.0003	
5.0E3	i.e., 5.0×10^{3}
5.0E+3	i.e., $5.0 \times 10^{+3}$
5.0E—3	i.e., 5.0×10^{-3}

Value of f

The value of f (precision) can be indicated to the Fortran compiler through control information supplied by the user. If f is specified by the user, the value of f must be $2 \leq f \leq 20$. If f is not specified by the user, the compiler uses f equal to eight digits. No more than f digits can be written.

The absolute value of a real constant must be between the limits 10^{-100} and $(1 - 10^{-f}) \times 10^{99}$, or be zero.

Within core storage, a real constant is stored in an exponential form occupying $n + 2$ digits ($n + 2$ core storage positions), where $n \leq f$. The first n digits contain the mantissa (the fraction part of the constant). A decimal point is understood to precede the high-order digit position. The last two positions contain the characteristic (exponent). For example, if f is defined as 18, a number in the source program having 18 or less significant digits results in a 20-digit real number, 18 for the mantissa and 2 for the characteristic.

Logical Constants

General Form. A logical constant can take either of the following forms.

.TRUE.
.FALSE.

Within core storage, a logical constant of .FALSE. is represented by the character zero (0). A logical constant of .TRUE. is represented by the character one (1).

Variables

A variable quantity is represented by a symbolic name, and is specified by its name and its type. The type of variable (integer, real, or logical) corresponds to the type (integer, real, or logical) of values that the variable assumes.

The initial value of a variable must be predefined before its use in a Fortran statement, including all arguments in a CALL statement. A variable can be defined by a READ statement, an arithmetic statement, or a DATA statement.

Variable Names

General Form. A variable name consists of one to six alphameric characters, the first of which must be alphabetic. Subroutines and functions are named in the same way as variables (see *Naming Subprograms*). Within the same program, a unique name must be used to represent a variable, a subroutine, and/or a function.

Examples.

L5
JOB1
BETATS
COST
K

Variable Types

The type of variable, real or integer, can be specified explicitly by a type statement or implicitly by name. Logical variables must have their type specified by a type statement.

Explicit Type Specification

Explicit type specification of a real or integer variable is made by the type statements, INTEGER, REAL, LOGICAL, and EXTERNAL. See *Type Statements.*

Implicit Type Specification

Implicit type specification of a real or integer variable is made in the following manner.

1. If the first character of the variable name is I, J, K, L, M, or N, it is an integer variable. For example, MAX, JOB, IDIST, and LESL are integer variables.
2. If the first character of the variable name is *not* I, J, K, L, M, or N, it is a real variable. For example, ALPHA, BMAX, Q, and WHIT are real variables.

Explicit type specification overrides implicit type specification. For example, if a variable name is JOB and a type statement specifies that this variable be real, the variable is treated as a real variable even though it implicitly has the form of an integer variable.

Subscripts

A variable can be made to represent any element of a 1-, 2-, or 3-dimensional array of quantities by appending one, two, or three subscripts, respectively, to the variable name. The variable is then called a subscripted variable. The subscripts are expressions of a special form whose value determines the element of the array to which reference is made.

Form of Subscripts

General Form. A subscript can take only one of the following forms:

V
C
V + C
V − C
C * V
C * V + C′
C * V − C′

where V represents any unsigned nonsubscripted integer variable, C and C′ represent any unsigned integer constant, and + denotes addition, — denotes subtraction, and * denotes multiplication.

A variable in a subscript cannot be subscripted.

Examples.

J
3
I + 3
K − 1
4 * L
2 * M + 5
3 * M − 4

Subscripted Variables

General Form. A subscripted variable consists of a variable name followed by parentheses enclosing one, two, or three subscripts separated by commas.

Examples.

A (I)
K (3)
BETA (8 * J + 2, K − 2, L)
MAX (I, J, K)

Subscripted variables must conform to the following.

1. Each variable that appears in subscripted form must have the size of the array specified preceding the first appearance of the subscripted variable in an executable statement or DATA statement. Array sizes are specified by using a DIMENSION statement or a COMMON statement.
2. The order and number of the subscript expressions must correspond to the order and number of the declared dimensions.
3. During execution, variable subscripts are evaluated so that the subscripted variable refers to a specific member of the array.

NOTE: A variable subscript reference of the form $A(C_1*V_1 \pm C_1', C_2*V_2 \pm C_2', C_3*V_3 \pm C_3')$, where A is defined in a DIMENSION or COMMON statement as $A(d_1,d_2,d_3)$, must conform to the following inequality:

$$(f + 2)(C_1V_1 + d_1C_2V_2 + d_1d_2C_3V_3) < 16{,}000$$

where f is the real size and C_1, V_1, and d_1 are zero if they are not applicable. The V_1's are the largest value expected in the subscripted variable reference.

Arrangement of Arrays in Core Storage

Arrays are placed in core storage in column order, in order of decreasing core-storage addresses.

One-dimensional arrays are stored sequentially.

Two-dimensional arrays are stored sequentially by column.

Three-dimensional arrays are stored sequentially by column from plane to plane. (The first subscript is cycled most rapidly, and the last least rapidly.)

For example, the array whose last element is A(3,5) appears in core storage as:

A(3,5),A(2,5),A(1,5),A(3,4), . . . , A(3,1),A(2,1),A(1,1)

Note that A(1,1) is in the high core-storage position, and A(3,5) is in the low core-storage position.

Expressions

The Fortran language includes two kinds of expressions, arithmetic expressions and logical expressions.

Arithmetic Expressions

An arithmetic expression consists of sequences of constants, subscripted or nonsubscripted variables, and arithmetic function references, separated by arithmetic symbols, commas, and parentheses. The arithmetic operation symbols and their meaning follow.

Symbol	*Meaning*
+	Addition
−	Subtraction
*	Multiplication
/	Division
**	Exponentiation

+, -, *, /	Real	Integer	Logical
Real	Valid	Invalid	Invalid
Integer	Invalid	Valid	Invalid
Logical	Invalid	Invalid	Invalid

Figure 1. Arithmetic Operators

**		Exponent		
		Real	Integer	Logical
Base	Real	Valid	Valid	Invalid
	Integer	Invalid	Valid	Invalid
	Logical	Invalid	Invalid	Invalid

Figure 2. Exponentiation

The following rules must be followed in constructing arithmetic expressions.

1. Figures 1 and 2 indicate which constants, variables, and functions can be combined by the arithmetic operators to form arithmetic expressions. Figure 1 gives the valid combinations with respect to the arithmetic operators +, —, *, and /. Figure 2 gives the valid combinations with respect to the arithmetic operator **.
2. Parentheses can be used, as in algebra, to group expressions, indicate the order of operations, and make interpretation easier for the user.
3. Expressions can be connected by arithmetic operation symbols to form other expressions, provided that:
   a. No two operators appear in sequence, and
   b. No operation symbol is assumed to be present. Parentheses are not assumed to be present. Multiplication cannot be implied. For example, the expression A**B**C is not permitted. It must be written as either A**(B**C) or (A**B)**C, whichever is intended.

Examples. In the following examples, implicit type specification is assumed.

A + B	(Valid)
A + 2	(Invalid — an integer constant cannot be added to a real variable)
A + 2.	(Valid)
A + 2.0	(Valid)
I * J	(Valid)
I * A	(Invalid — a real variable cannot be multiplied by an integer variable)
A * B	(Valid)
A * I	(Invalid — an integer variable cannot be multiplied by a real variable)
A ** 2.0	(Valid)
A ** 2	(Valid)
I ** 2.0	(Invalid — an integer variable cannot be raised to a real constant power)
I ** 2	(Valid)

Preceding an expression by a + or — sign does not affect the type of the expression.

Exponentiation is defined as follows:

1. Given: A**I, where A is an integer variable or a real variable and I is an integer variable.
   If $I > 0$, then A**I = A*A* . . . *A, (I factors). Result is the same type variable as A.
   If $I = 0$ and if $A \neq 0$, then A**I = 1. Result is the same type variable as A.
   If $I = 0$ and if $A = 0$, then A**I = undefined.
   If $I < 0$ and if $A \neq 0$, A**I =
   $$\frac{1}{A*A*\ldots*A, (|I| \text{ factors})}$$
   Result is a real variable.
   If $I < 0$ and if $A = 0$, then A**I = undefined.
2. Given: A**R, where A is a real variable and R is a real variable.
   If $A > 0$, then A**R = EXP(R*ALOG(A)). Result is a real variable.
   If $A = 0$ and if $R > 0$, then A**R = 0. Result is a real variable.
   If $A = 0$ and if $R \leq 0$, then A**R = undefined.
   If $A < 0$, then A**R = undefined.

Order of Operations

Parentheses can be used, as in algebra, in expressions to specify the order in which the expression is to be evaluated. Expressions are evaluated from left to right. Where parentheses are omitted, the order of computation is as follows.

1. Function computation and substitution.
2. Exponentiation
3. Multiplication and division.
4. Addition and subtraction.

For example, the expression A + B/C —D**E*F — G will be interpreted A + (B/C) — (D^E * F) — G and will be evaluated in order from left to right.

The expression —I**N, —A**B, and —A**N will be interpreted as (—I)**N, (—A)**B, and (—A)**N if preceded by a left parenthesis or an equal (replacement) sign.

Logical Expressions

A logical expression consists of certain sequences of logical constants, logical variables, references to logical functions, and arithmetic expressions separated by logical operators or relational operators. A logical expression always has the value .TRUE. or .FALSE. .

The logical operators (where *a* and *b* are logical expressions) are:

Operator	*Definition*
.NOT.a	This has the value .TRUE. only if *a* is .FALSE. ; it has the value .FALSE. only if *a* is .TRUE. .
a.AND.b	This has the value .TRUE. only if *a* and *b* are both .TRUE. ; it has the value .FALSE. if either *a* or *b* is .FALSE. .
a.OR.b	(Inclusive OR) This has the value .TRUE. if either *a* or *b* is .TRUE. ; it has the value .FALSE. only if both *a* and *b* are .FALSE. .

The logical operators .NOT. , .AND. , and .OR. must always include the preceding and following periods.

Examples.

```
Z .OR. X
X . AND. .NOT. Y
```

The relational operators are:

Operator	*Definition*
.GT.	Greater than (>)
.GE.	Greater than or equal to (>)
.LT.	Less than (<)
.LE.	Less than or equal to (<)
.EQ.	Equal to (=)
.NE.	Not equal to ($\neq$)

The relational operators must always include the preceding and following periods.

Examples.

```
.NOT. C .EQ. Z
A .GT. B .OR. C .LE. B
```

The following are the rules for constructing logical expressions:

1. Figure 3 indicates which constants, variables, and functions can be combined by the relational operators to form a valid logical expression. The logical expression will have the value .TRUE. if the condition expressed by the relational operator is met. Otherwise, the logical expression will have the value .FALSE. .
2. A logical expression may also consist of a single logical constant, a logical variable, or a reference to a logical function.
3. The logical operator .NOT. must be followed by a logical expression, and the logical operators .AND. and .OR. must be preceded and followed by logical expressions to form more complex logical expressions.
4. Parentheses may not be used in logical expressions.

Order of Operations

In logical expressions, the order of operations is understood to be as follows:

1. Function computation and substitution.
2. Exponentiation.
3. Multiplication and division.
4. Addition and subtraction.
5. .LT. , .LE. , .EQ. , .NE. , .GT. , .GE.
6. .NOT.
7. .AND.
8. .OR.

.GT., .GE., .LT., .LE., .EQ., .NE.	Real	Integer	Logical
Real	Valid	Invalid	Invalid
Integer	Invalid	Valid	Invalid
Logical	Invalid	Invalid	Invalid

Figure 3. Relational Operators

The Arithmetic Statement

The arithmetic statement defines a numerical or logical calculation. A Fortran arithmetic statement closely resembles a conventional arithmetic or algebraic formula, except that the equal sign specifies replacement rather than equality.

General Form. $a = b$

a is a real, integer, or logical variable that may be subscripted.

b is an expression.

Examples.

```
A = B + (C - 3.0) *D
A (I) = B (I) + SIN (C (I) )
V = .TRUE.
E = C.GT.D.AND.F.LE.G
```

Figure 4 indicates the types of expressions and variables that can be equated to result in valid arithmetic statements.

In the following examples of arithmetic statements, *I* is an integer variable, *A* and *B* are real variables, and *C*, *D*, and *E* are logical variables.

Statement	*Definition*
A = B	Replace A by the current value of B.
I = B	Truncate B to an integer, convert it to an integer constant, and store it in I.
A = I	Convert I to a real variable and store it in A.
I = I + 1	Add 1 to I, and store it in I.
A = 3 * B	*Not permitted.* The expression is mixed for multiplication; that is, it contains both a real variable and an integer constant.
C = .TRUE.	Store the logical constant .TRUE. in C.
D = .NOT. C	If C is .TRUE. , store the value .FALSE. in D. If C is .FALSE. , store the value .TRUE. in D.
D = I.GE.A	*Not permitted.* An integer and a real variable may not be joined by a relational operator.

Left Side of Equal Sign	Right Side of Equal Sign: Expression / Variable	Real	Integer	Logical
	Real	Valid	Valid	Invalid
	Integer	Valid	Valid	Invalid
	Logical	Invalid	Invalid	Valid

Figure 4. Valid Combinations of Variables and Expressions

C = D.OR. .NOT.E

D	E	~E	DV~E
T	T	F	T
T	F	T	T
F	T	F	F
F	F	T	T

where: ~ implies .NOT. , and V implies .OR. .

C = 3. .GT.B — C is .TRUE. if 3. is greater than B; C is .FALSE. otherwise.

The last two examples illustrate the following rules:

1. Two logical operators can appear in sequence only if the second logical operator is .NOT. .
2. Two periods may appear in succession as in C = D.OR..NOT.E or when one belongs to a constant and the other to a relational operator.

The Control Statements

The nine control statements enable the programmer to control and terminate the flow of his program. Transfer of control must be to an executable statement.

The Unconditional GO TO Statement

General Form. GO TO n

n is a statement number of an executable statement.

This statement causes control to be transferred to the statement numbered n.

Example.

```
GO TO 25
```

This statement causes control to be transferred to statement numbered 25.

The Computed GO TO Statement

General Form. GO TO $(n_1,n_2,\ldots,n_m),i$

$n_1, n_2, \ldots, n_m$ are statement numbers of executable statements.

The limits of the value m are $1 \leq m \leq 9$.

i is a nonsubscripted integer variable.

The limits of the value i are $1 \leq i \leq 9$.

This statement causes control to be transferred to the statement numbered $n_1,n_2,\ldots,n_m$ depending on whether the value of i is $1,2,\ldots,m$, respectively, at the time of execution.

Example.

```
GO TO (30, 45, 50, 9), K
```

In the example, if K is equal to 3 at the time of execution, the program will transfer control to the third statement in the list, statement 50.

The Logical IF Statement

General Form. IF *(t) s*

t is a logical expression.

s is any executable statement except DO, an arithmetic IF, or another logical IF.

If the logical expression *t* is true, statement *s* is executed. Control then transfers to the next sequential executable statement, unless *s* is a GO TO statement, in which case control is transferred as indicated.

If *t* is false, control transfers to the next sequential statement.

If *t* is true and *s* is a CALL statement, control transfers to the next sequential executable statement on return from the subprogram called.

Examples.

```
IF (A .AND. B) F = SIN (R)
IF (16 .GT. L) GO TO 24
IF (D .OR. X .LE. Y) GO TO (18, 20), I
IF (Q) CALL SUB (Q)
```

The Arithmetic IF Statement

General Form. IF *(a)* n_1, n_2, n_3

a is an arithmetic expression, type integer or type real.

n_1, n_2, n_3 are statement numbers of executable statements.

The arithmetic IF statement is used to transfer control to one of three specified statements depending on the value of an arithmetic expression. The arithmetic statement *a* is tested. If *a* is less than zero, control transfers to statement n_1. If *a* is equal to zero, (plus or minus), control transfers to statement n_2. If *a* is greater than zero, control transfers to n_3.

Examples.

```
IF (A(J, K) − B) 10, 4, 30
IF (D*E + BRN) 9, 9, 15
```

The DO Statement

General Form. DO *n i* = m_1,m_2,m_3

n is a statement number of an executable statement.

i is a nonsubscripted integer variable.

m_1,m_2,m_3 are either unsigned integer constants greater than zero or unsigned nonsubscripted integer variables whose value is greater than zero. The number of digits in the integer size (*k*) must be at least one greater than the greatest number of digits used for m_1,m_2, or m_3. In the example below, *k* must be greater than or equal to 3.

m_3 is optional. If m_3 is not stated, its value is assumed to be 1. If it is omitted, the preceding comma must also be omitted.

Examples.

```
DO 30 I = 1, M, 2
DO 24 I = 1, 10
```

The DO statement is an instruction to execute repeatedly the statements that follow, up to and including the statement numbered *n*. The first time the statements are executed, *i* has the value m_1 and each succeeding time *i* is increased by the value of m_3. After the statements have been executed with *i* equal to the highest value that does not exceed m_2, control passes to the executable statement following statement number *n*. This is called the normal exit from the DO statement.

The Range of the DO Statement. The range of the DO statement is that set of statements that will be executed repeatedly. That is, it is the sequence of statements immediately following the DO statement, up to and including the statement numbered *n*. After the last execution in the range, the DO is said to be satisfied.

The Index of the DO Statement. The index of the DO statement is the integer variable *i*. Throughout the range of the DO, the index is available for computation, either as an ordinary integer or as the variable of a subscript. After a normal exit from a DO, the index *i* must be redefined before it is used in computation. After exiting from a DO by transferring out of the range of the DO, the index *i* is available for computation and is equal to the last value it attained.

DO's Within DO's. A DO statement can be contained within another DO statement. This is called a nest of DO's. If the range of a DO contains another DO, then all statements in the range of the enclosed DO must be within the range of the enclosing DO. The maximum depth of nesting, not including implied DO's in I/O lists, is twelve. That is, a DO can contain a second DO, the second can contain a third, the third can contain a fourth, and so on up to twelve statements.

Transfer of Control. Control cannot be transferred into the range of a DO from outside its range. However, control can be transferred out of a DO range. In this case, the value of the index remains available for use. If the exit is transferred out of the range of a set of nested DO's, then the index of each DO is available.

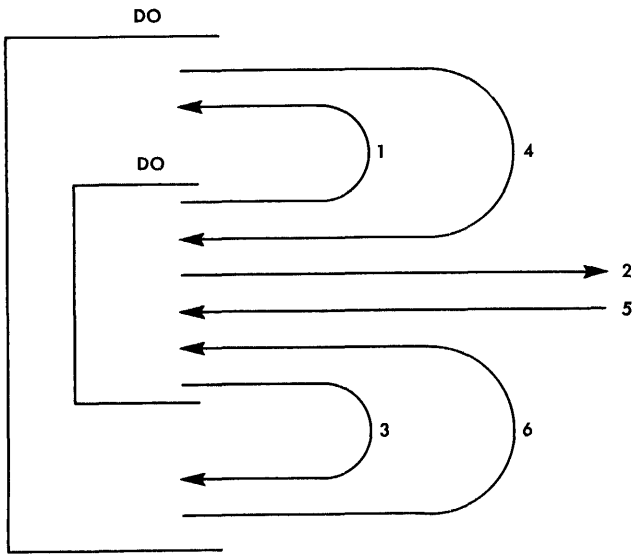


Figure 5. Nest of DO's

Figure 5 illustrates the possible transfers in an out of the range of a DO. In the figure, 1, 2, and 3 are permitted, but 4, 5, and 6 are not permitted.

Restrictions on Statements in the Range of a DO. Any statement that redefines the index or any of the indexing parameters (*m*'s) is not permitted in the range of a DO.

The range of a DO cannot end with a GO TO type statement or another DO. The range of a DO can end with a logical IF, in which case control is handled in the following manner. If the logical expression *t* is false, the DO is repeated. If the logical expression *t* is true, statement *s* is executed and then the DO is repeated. If *t* is true and *s* is a transfer type statement, control is transferred as indicated by *s*.

When a reference to a subprogram is made in the range of a DO, care must be taken that the called subprogram does not alter the index or any of the indexing parameters.

The CONTINUE Statement

General Form. CONTINUE

CONTINUE is a dummy statement that does not produce any executable instructions. It is most frequently used as the last statement in the range of a DO to produce a branch address for GO TO statements that are intended to begin another repetition of the DO range.

Example.

```
      .
      .
      .
      DO 20 I = 2, N
      IF (BIGA .LT. A (I) )BIGA = A (I)
20    CONTINUE
      .
      .
      .
```

The PAUSE Statement

General Form. PAUSE or PAUSE *n*

n is an unsigned integer constant of one to three digits.

The statement causes the machine to halt. The integer constant *n* is in the B-address register. If *n* is not specified, it is assumed to be zero. When the machine is restarted by pressing the start key, the next Fortran statement is executed.

The STOP Statement

General Form. STOP

The STOP statement terminates execution of the program. When the STOP statement is executed, control returns to the System Control Program. The program can have any number of STOP statements. The STOP statement must consist entirely of alphabetic characters.

The END Statement

General Form. END

The END statement defines the end of a program or a subprogram. Physically, it must be the last statement of each program or subprogram. As the END statement is not executable, it must not be encountered in the flow of the program.

The Specification Statements

The specifications statements provide information about storage allocation and the variables and constants used in the program.

The DIMENSION Statement

General Form. DIMENSION v_1 (i_1), v_2 (i_2), . . . , v_n (i_n)

$v_1, v_2, \ldots, v_n$ are the names of arrays.

$i_1, i_2, \ldots, i_n$ are each composed of 1, 2, or 3 unsigned integer constants and/or integer variables separated by commas. Each integer specifies the maximum value of the subscript. i can be an integer variable only when the DIMENSION statement appears in a subprogram.

Examples.

```
DIMENSION A(10), B(5, 15), C(L, M)
DIMENSION S(10), K(5,5,5), G(100)
```

The DIMENSION statement provides the information necessary to allocate storage for arrays in the object program. It defines the dimensionality and the maximum size of each array listed.

Each variable that appears in subscripted form in the source program must appear in a DIMENSION statement contained in the source program. However, if the dimension information for a variable is included in a COMMON statement in the source program, it must not be included in a DIMENSION statement.

A single DIMENSION statement can specify the dimensions of any number of arrays. The DIMENSION statement that specifies the array size must precede the first appearance of each subscripted variable in an executable or DATA statement. (See Figure 6).

Dimensions specified in a COMMON statement are subject to all the rules for the DIMENSION statement, except that adjustable dimensions are not permitted.

```
        DIMENSION A(5, 10)
              .
              .
              .
        CALL MAYMY (...,A,...)
              .
              .
              .
        SUBROUTINE MAYMY (...,R,...)
              .
              .
              .
        DIMENSION...,R(5, 10),...
              .
              .
              .
```

● Figure 6. Passing Array Names

The COMMON Statement

General Form. COMMON *a, b, c, . . .*

a, b, c, . . . are variable or array names that can be dimensioned

Example.

```
COMMON A, B, C (5, 10)
```

The COMMON statement refers to a common area in core storage. Variables or arrays that appear in main programs and subprograms can be made to share the same storage locations by using the COMMON statement. For example, if one program has the statement COMMON A and a second program has the statement COMMON B, the variables (or arrays) of A and B will occupy the same storage locations in the COMMON area. These variables (or arrays) appearing in COMMON statements are assigned locations relative to the beginning of the COMMON area.

Within a specific program or subprogram, variables and arrays are assigned core storage locations from the high core-storage addresses to lower core-storage addresses in the sequence in which their names appear in the COMMON statement. Subsequent sequential storage assignments within the same program or subprogram are made with additional COMMON statements.

For example, if the main program contains the statement

```
COMMON A, B, C
```

and a subprogram contains the statement

```
COMMON L, M, N
```

then A, B, and C are assigned sequential locations, as are L, M, and N. Further, A and L will occupy the same location, B and M will occupy the same location, and C and N will occupy the same location.

Variables declared in COMMON must agree, respectively, in type. In the preceding example, A and L are type real, as are B and M, and C and N. (L, M, and N must be declared as real in the subprogram.)

A dummy variable can be used in a COMMON statement to establish shared locations of variables that would otherwise occupy different locations. For example, the variable X can be assigned to the same location as the variable C of the previous example by using the following statement.

```
COMMON R, S, X
```

where R and S are dummy names that are not used elsewhere in the program.

Redundant entries are not permitted in a COMMON statement. For example, the following statement is invalid.

```
COMMON F, G, H, F
```

Variables brought into COMMON through EQUIVALENCE statements may increase the size of COMMON (see *The EQUIVALENCE Statement*).

Two variables in COMMON cannot be made equivalent to each other, either directly or indirectly.

The EQUIVALENCE Statement

General Form. EQUIVALENCE (*a, b, c, . . .*), (*d, e, f, . . .*), . . .

a, b, c, d, e, f, . . ., are variables which may be subscripted. Subscripted variables can have single or multiple subscripts. These subscripts must be integer constants.

Example.

```
EQUIVALENCE (A, B(1), C(5)), (D(17), E(3))
```

The EQUIVALENCE statement controls the allocation of core storage by causing two or more variables to share the same core storage location.

An EQUIVALENCE statement can be placed anywhere in the source program. Each pair of parentheses in the statement list encloses the names of two or more variables that are to be stored in the same location during execution of the object program. Any number of equivalences (sets of parentheses) can be given.

In an EQUIVALENCE statement, *C(p)* is defined as the location of the *pth* element in the array *C*. Thus, in the preceding example, the EQUIVALENCE statement indicates that A, and the B and C arrays are to be assigned storage locations so that the elements A, B(1), and C(5) are to occupy the same location. In addition, it specifies that D(17) and E(3) are to occupy the same location. This implies that D(15) and E(1) occupy the same location.

All variables that are to occupy the same location as a result of an EQUIVALENCE statement must be of the same type and must not be inconsistent in relative core-storage locations. For example, the statement

```
EQUIVALENCE (A(4), C(2), D(1)), (A(2), D(2))
```

is invalid. The equivalencing of A(4), C(2), and D(1) sets up an equivalence among elements of each row below.

```
A(1)
A(2)
A(3)    C(1)
A(4)    C(2)    D(1)
A(5)    C(3)    D(2)
 .       .       .
 .       .       .
 .       .       .
```

Thus, D(2) must not be equivalenced to A(2). EQUIVALENCE (A(3), A(4)) is also invalid.

Variables or arrays not mentioned in an EQUIVALENCE statement will be assigned unique locations. Locations can be shared only among variables, not among constants.

The sharing of storage locations requires a knowledge of which Fortran statements cause a new value to be stored in a location. There are four such statements.

1. Execution of an arithmetic statement stores a new value in the variable to the left of the equal sign.
2. Execution of a DO statement, the terminal statement of a DO, or an implied DO in an I/O list stores a new indexing value.
3. Execution of a READ statement stores new values at the locations specified by variable names in the input list.
4. Execution of a subroutine or function may store a new value in any of its actual arguments or any variables in COMMON.

Variables brought into COMMON through EQUIVALENCE statements can increase the size indicated by the COMMON statements, as in the following example.

```
COMMON A, B, C
DIMENSION D(3)
EQUIVALENCE (B, D(1))
```

The layout of core storage indicated by this example (extending from the lowest location of COMMON to the highest location of COMMON) is:

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

A variable cannot be made equivalent to an element of an array in such a way as to cause the array to extend beyond the beginning of COMMON. For example, the following coding is invalid.

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

This would force D(1) to precede A, as follows.

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

The Type Statements

General Forms.

INTEGER *a, b, c, . . .*
REAL *a, b, c, . . .*
LOGICAL *a, b, c, . . .*
EXTERNAL *x, y, z, . . .*

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

x, y, z, . . . are function or subroutine names appearing as actual arguments within the program.

Examples.

```
INTEGER BIXF, X, QF, LSL
REAL IMIN, LOG, GRN, KLW
LOGICAL F, G, LWG
EXTERNAL SIN, MATMPY, INVTRY
```

The variable or function names following the type (INTEGER, REAL, LOGICAL, or EXTERNAL) in the type statement are defined to be of that type, and remain that type throughout the program. The type cannot be changed.

In the examples, note that LSL and GRN need not appear in their respective type statements, for their type is implied by their first characters.

The type statement must precede the first use of a name in any executable statement or DATA statement in the program. The appearance of a name in any type statement except EXTERNAL overrides the implicit type assignment. A name can appear in two type statements only if one of them is EXTERNAL. A name declared to be of a given type can assume only values of the same type.

Subprogram names that are actual arguments of other subprograms must appear in EXTERNAL type statements. Dummy arguments in a SUBROUTINE or FUNCTION statement must also appear in an external statement if they are dummy subroutine names that are actual arguments of other subprograms. For example, assume both SOMEF and OTHER are subprograms. If A = SOMEF (OTHER, B, C) + B appears in a program, the type statement EXTERNAL OTHER is required in the program.

The DATA Statement

General Form. DATA *list*/d_1, $d_2, \ldots, d_n$/, *list*/d_1, d_2, k*$d_3, \ldots, d_m$/, . . . , *list*/d_x, d , . . . , d /

list contains the names of the variables being defined.
d_i is the information literal.
k is an integer constant.

Examples.

```
DATA R, Q/14.2, 3HEND/, B(2)/0./
DATA (B(I), C(I), I = 1, 40, 2) /2.0, 3.0, 38*100.0/
```

The DATA statement is used to compile data into the object program, initializing the values of the specified values prior to execution. For example, the result of the first DATA statement would be:

Variable	*Initialized Value*
R	14.2
Q	END (right justified)
B(2)	0.

The list can contain nonsubscripted variables, subscripted variables, or array names that imply the entire array is initialized. Subscripted variables must have constant subscripts or subscripts whose variables are under control of DO-implying parentheses and associated parameters. The DO-defining parameters must be integer constants.

A k appearing before a d-literal indicates that the field is to be repeated k times. The k must be a one to three digit integer. It must be separated from the field to be repeated by a times sign (*).

The d-literals can take any one of the following forms.

1. Integer constants and real constants.
2. Alphameric characters. (The variable name associated with the H-Conversion must conform to the normal rules for naming Fortran variables and must be type real.) The alphameric field is written nH followed by n alphameric characters. n should be less than or equal to $f + 2$. Each group of $f + 2$ characters forms a word. If $n < f + 2$, the characters are right-justified in the field and the word is filled out with blanks. If $n > f + 2$, only the leftmost $f + 2$ characters are stored in the word and the remaining characters are lost. Blanks are significant in alphameric fields. The variable can be referred to by the appropriate name in an I/O list or as an actual subprogram argument. It cannot be used in arithmetic or logical operations.
3. Logical constants. The logical constants can be .TRUE. , T, .FALSE. , or F.
4. Variable names appearing in DATA statements cannot appear in COMMON statements.

There must be a one-to-one correspondence between the list items and the data literals. For example, if it is desired to define 16 alphameric characters (say 8HDATAbTOb,8HBEbREADb) as a variable (say G) and if f is 6, then G must be dimensioned to contain at least two elements.

When DATA-defined literals are redefined during execution, these variables will assume their new values regardless of the DATA statement.

Input/Output Statements

The input/output (I/O) statements control the transmission of data between the computer and input/output devices, such as card readers, card punches, printers, magnetic tape units, and disk units. The I/O statements fall into one of the following general categories.

FORMAT Statements. FORMAT statements are non-executable statements that specify the external arrangement of the edited information to be transferred and the editing transformation between internal (core storage) and external forms of the information. FORMAT statements are used in conjunction with the general I/O statements.

General I/O Statements. The general I/O statements READ and WRITE cause the transmission of information between core storage and the logical file that is assigned to an input/output device. (For a description of logical files, see *Logical Files.*)

If these statements refer to FORMAT specification statements, then the information is edited and the I/O statements are called edited I/O statements. Otherwise, the I/O statements are called unedited I/O statements.

Manipulative I/O Statements. The statements END FILE, REWIND, FIND, and BACKSPACE manipulate the input/output device to which a specific logical file is assigned.

I/O Specification Statement. The DEFINE FILE statement defines the size and characteristics of logical files that are assigned to disk units to be accessed randomly.

List Specifications

The general I/O statements cause the transmission of data.

An I/O list is a series of list items that are separated by commas. A single list item can be a subscripted or non-subscripted variable. An I/O list is read from left to right.

An I/O list is ordered. The order must be the same as the order in which the input exists in the input medium, or in which the output is to exist in the output medium.

An I/O list can contain implied DO's. All items to be included in the range of the implied DO and the indexing information must be set off by parentheses. Implied DO's can be nested to a nesting depth of three by placing matching parentheses around the first and last items of each successive inner DO range. Redundant parentheses are not allowed.

Consider this I/O list.

```
M, (Q(I), I = 1, 10)
```

This implies that the information in the external input and/or output medium is arranged in the following order.

```
M, Q(1), Q(2), . . . , Q(10)
```

Consider this I/O list.

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

This implies that the information in the external input and/or output medium is arranged in the following order.

```
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)
```

An I/O list containing parentheses is executed in a manner similar to the execution of a DO loop. The left parenthesis (except subscripting parentheses) is treated as though it were a DO statement, with the indexing information given immediately before the matching right parenthesis. The rules for specifying $i = m_1,m_2,m_3$ are the same as those for the DO statement. The DO range extends up to the indexing information. The order of the I/O list shown in the first example can be considered equivalent to the following steps.

```
      M
      DO 1 I = 1, 10
1     Q (I)
```

The order of the I/O list shown in the second example can be considered equivalent to the following steps.

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

In the preceding paragraph, the list item (C(I), D(I,K),I = 1, 10) is an implied DO. It is evaluated as

shown. The range of an implied DO must be clearly defined by parentheses.

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

Any number of items can appear in a single list. If the list is in an edited statement, the format of each data value must be the same as specified in a corresponding FORMAT statement. Essentially, the I/O list controls the number of the data values read or written. If the corresponding FORMAT statement indicates more data values are to be transmitted than there are items on the list, only the number of data values indicated on the list are transmitted. All remaining data values are ignored. If a list contains more items than there are data values on one edited input record, additional records are read. When a read operation is performed, a list must not contain more items than data values in one unedited input record.

A list can include no items only if the data to be transferred is entirely specified in a FORMAT statement.

Reading or Writing Entire Arrays

When the reading or writing of an entire array is required, an abbreviated notation can be used in the list of the input/output statement. Only the name of the array need be given, and subscripts can be omitted.

If A has previously been listed in a DIMENSION or COMMON (with dimensions) statement, either the statement

READ (5, 10) A or READ (3) A

is sufficient to cause all the elements of array A to be read in the implied order of elements. If A is a 2×3 array, the elements are read into core storage in the following order.

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

The FORMAT Statement

General Form. FORMAT (S_1, S_2, . . . , S_n/S'_1, S'_2, . . . , $S'_n/\ldots$) Each field, S_i, is a format specification.

Example.

FORMAT (I2/(E12.4,F10.2))

The edited input/output statements require, in addition to a list of items to be transmitted, reference to a FORMAT statement that describes the edited data record and the type of conversion to be performed between the internal representation and the external representation for each item in the list.

FORMAT statements are not executed. They need not appear in any special place within the program. Each FORMAT statement must be given a statement number.

The FORMAT statement indicates the maximum size of each edited record to be transmitted. Except when a FORMAT statement consists entirely of H- or X-conversion fields, it is used in conjunction with the list of some particular input/output statement. 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).

Edited data records must consist of one of the following:

1. On tape, 200-character records are written. The data records, specified by a FORMAT statement, begin in character position one. All unused characters are left blank. On disk, either one or two sectors is used, depending on whether the record is defined to be greater or less than 100 characters.
2. A punched card record with a maximum of eighty characters.
3. A line to be printed with a maximum of 120 or 132 print positions, depending on the printer being used.

The first left parenthesis begins a record. In a read operation, the record is read. In a write operation, the output record is begun, but not written.

A slash ends the current record and begins a new record. In a read operation a slash means that no more information is obtained from the last record that was read. In a write operation, the output that has been developed is written, even though the output record is blank, as when two slashes are adjacent.

The final right parenthesis of the FORMAT statement terminates the current record in the same manner as a slash. If list items remain to be processed, it also begins a new record and repeats. A repeat starts with the last repetitive group including the repeater, if there is one. Otherwise it starts with the specification following the first parenthesis of the FORMAT statement.

During input/output operations, the object program scans the FORMAT statement to which a specific input/output statement refers. When a specification for a numerical, logical, or alphameric field is found and list items remain to be transmitted, editing takes place according to the specification, and scanning of the FORMAT statement resumes. If no list items remain when

one of the preceding specifications or the final right parenthesis is processed, the current record and the execution of that particular input/output statement are ended.

Format Specifications

Numeric Fields

Three types of specifications are available for numeric data.

Internal	*Conversion Code*	*External*
Real	E	Real with E exponent
Real	F	Real without exponent
Integer	I	Integer

These types of conversion are specified in the following forms.

E*w.d*
F*w.d*
I*w*

E, F, and I specify the type of conversion.

w is an unsigned integer constant specifying the field width of the data, including signs and the exponent part, if appropriate. This field width can be greater than that required for the actual digits to provide spacing preceding the number.

d is an unsigned integer constant or zero that represents the number of numerics in the field that appear to the right of the decimal point. *d* is ignored for input when a decimal point actually appears. The exponent part for E-conversion is not included in this number.

For example, the statement FORMAT (1H1, I3, E12.4, F10.4) causes the following line to print after a skip to channel 1 (when given in conjunction with a WRITE statement).

	+	−+	−−
Stored data	00027	9320963102	7634352602
Field specifications	I3	E12.4,	F10.4
Printed line	b27b-0.9321Eb02bbb-0.0076		

where *b* indicates a blank.

Specifications for successive fields are separated by commas. Specification of more characters than are permitted for the appropriate input/output record cannot be given. Thus, a format for a record to be printed should not provide for more characters (including blanks) than can be handled by the printer.

Information to be transmitted with E- and F-conversion must be of type real. Information to be transmitted with I-conversion must be of type integer.

The field width *w* for I-conversion output must include a space for the sign.

The field width *w* for F-conversion output must include a space for the sign, a space for the decimal point (optional for input), and a space for a possible zero that precedes the decimal if the absolute value of the number is less than one. Thus, $w \geqq d + 3$.

The field width *w* for E-conversion output must include a space for the sign, a space for the decimal point (optional for input), and a space for a possible zero that precedes the decimal if the absolute value of the number is less than one, and four spaces for the E, exponent sign, and exponent. Thus, $w \geqq d + 7$.

The exponent that can be used with E-conversion is the power of 10 by which the number must be multiplied to get its true value. The exponent is written with an E followed by a minus sign if the exponent is negative, or a plus sign or a blank if the exponent is positive, and then followed by the exponent. The exponent can be one of two numbers. For example, the value .002 can be written as .2E-02.

If a number converted by I-, E-, and F-conversion on output requires more spaces than are allowed by the field width *w*, an X is inserted in the low-order (rightmost) position. If the number requires less than *w* spaces, the high-order spaces are filled with blanks. A space preceding a number output under I-, E-, or F-conversion indicates a positive value. The plus sign is not included, but *w* must be sufficiently large to include the blank or a minus sign.

Logical Fields

The specification *Lw* is used to transfer logical variables. *w* is an unsigned integer constant that specifies the field width on the external medium.

For input fields, *T* or *F* as the leftmost nonblank character in the field results in a value of .TRUE. or .FALSE. for the logical variable. A blank field results in a value of .FALSE. .

For output fields, *T* or *F* will appear right-justified in the field when the logical variable is .TRUE. or .FALSE. , respectively.

Alphameric Fields

Fortran provides two ways by which alphameric information can be transmitted. The internal representation is the same for both specifications.

The specification *Aw* causes *w* characters to be read into or written from a core-storage location designated by a variable or an array name.

The specification *nH* specifies that alphabetic information is contained in a FORMAT statement.

The basic difference between A- and H-conversion is that alphameric information handled by A-conversion is given a variable name or an array name. Thus, it can be referred to by the appropriate name in an I/O

list, a data-name list, an actual subprogram argument, or a dummy subprogram argument. The associated I/O statement therefore requires a list when A-conversion is specified by the FORMAT statement.

Information handled by H-conversion is not labeled. It is a constant field and cannot be referred to or manipulated in core storage in any way.

A-Conversion

The variable name to be converted by A-conversion must conform to the normal rules for naming Fortran variables. The variable name must be of type real.

For input, *nAw* is interpreted to mean that the next *n* successive fields of *w* characters each are to be transmitted to core storage without conversion. If *w* is greater than $f + 2$ (*f* is the assigned real size), only the $f + 2$ leftmost characters are significant. If *w* is less than $f + 2$, the characters are right justified and the high-order positions are filled with blanks.

For output, *nAw* is interpreted to mean that the next *n* successive fields of *w* characters each are to be the result of transmission from core storage without conversion. If *w* is greater than $f + 2$ in each of the *n* fields, only $f + 2$ characters of output are transmitted, followed by $w - (f + 2)$ blanks. If *w* is less than $f + 2$, the *w* rightmost characters of the field are transmitted.

H-Conversion

The specification *nH* is followed by *n* alphameric characters in a FORMAT statement. A comma separates successive specifications, including the H-conversion, used in the FORMAT statement. The separating comma appears after the last alphameric character, which can be a blank. For example,

..., 32HbTHISbISbALPHAMERICbINFORMATIONb, ...

Note that blanks are considered alphameric characters and must be included as part of the count *n*.

The effect of *nH* depends on whether it is used with input or output. For input, *n* characters are extracted from the input record and replace the *n* characters of the appropriate source program statement.

For output, the *n* characters following the specification, or the characters that replaced them, are written as part of the output record.

Figure 7 shows an example of A- and H-conversion in a FORMAT statement. The statement FORMAT (4HbXY = ,F8.3,A8) could produce the lines shown in the figure. *b* indicates a blank character.

Blank Fields — X-Conversion

For input, *nX* causes *n* characters in the input record to be skipped, regardless of what they actually are.

For output, *nX* causes *n* blank characters to be introduced into the output record.

Repetition of Field Format

It may be desirable to transfer *n* successive fields within the same record with the same format specification. This is specified by placing a number *n*, an unsigned integer constant, before E, F, I, L, or A. Thus, the field specification 3E12.4 is the same as E12.4, E12.4, E12.4.

Repetition of Groups

A repetitive group is an integer constant of not more than three digits followed by a left parenthesis, a specification list, and a right parenthesis. Thus, the specification FORMAT (2(F10.6,E10.2),I4) has the same effect as FORMAT (F10.6,E10.2,F10.6,E10.2,I4).

Scale Factors — P-Conversion

To permit general use of E- and F-conversion, a scale factor *s* followed by the letter P can precede the specification.

The scale factor is defined for F-conversion input as follows:

$$10^{-s} \times \text{external quantity} = \text{internal quantity}$$

The scale factor is defined for E- and F-conversion output as follows.

$$\text{external quantity} = \text{internal quantity} \times 10^{s}$$

For input, scale factors have effect only on F-conversion. For example, if input data is in the form XX.XXXX and it is desired to use it internally in the form .XXXXXX, the FORMAT specification to make this change is 2PF7.4.

```
XY = b-93.210bbbbbbbb

XY = 9999.999bbSNSSW1

XY = bb28.768bbbbbbbb
```

Figure 7. Example of A- and H-Conversion

For output, scale factors can be used with both E- and F-conversion. For example, the statement FORMAT (I3,3F11.3) would give the following record.

b27bbbb-93.210bbbbb-0.008bbbbbb0.554

Using the same data and the statement FORMAT (I3,1P3F11.3) would give the following record.

b27bbb-932.096bbbbb-0.076bbbbbb5.536

Whereas, using the same data and the statement FORMAT (I3,-1P3F11.3) would give the following record.

b27bbbbb-9.321bbbbb-0.001bbbbbb0.055

A positive scale factor used for output with E-conversion increases the base and decreases the exponent. A negative scale factor used for output with E-conversion decreases the base and increases the exponent. Thus, using the same data and the statement FORMAT (I3,1P3E12.4) would give the following record.

b27b-9.3210Eb01b-7.6344E-03bb5.5536E-01

Whereas, using the same data and the statement FORMAT (I3,-1P3E12.4) would give the following record.

b27b-0.0932Eb03b-0.0763E-01bb0.0555Eb01

The scale factor is assumed to be zero if no value is given. However, once a value has been given, it holds for all E- and F-conversions following the scale factor within the same FORMAT statement. This applies to both single-record formats and multiple-record formats (see *Multiple-Record FORMAT Specifications).* Thus, the specification

1PE10.4,E12.5,F8.3

is equivalent to

1PE10.4,1PE12.5,1PF8.3

Once the scale factor is given, a subsequent scale factor of zero in the same FORMAT statement must be specified by 0P. Thus, if it is desired that only the first item in a specification be affected by P-conversion, the specification should be written

1PE10.4,0PE12.5,F8.3

Scale factors have no effect on I-, A-, and L-conversion.

Multiple-Record FORMAT Statements

To deal with many output records, a single FORMAT statement can have several single-record format specifications separated by a slash (/) to indicate the beginning of a new record. For example,

FORMAT(3F9.2,2F10.3/8E11.4)

transfers the first, third, fifth, . . . , records with the specification 3F9.2,2F10.3; and the second, fourth, sixth, . . . , records with the specification 8E11.4.

Two consecutive slashes (//) indicate a blank record. For example,

FORMAT(3F9.2,2F10.3//I2//)

transfers the first, sixth, eleventh, . . . , records with the specification 3F9.2,2F10.3; the second, seventh, twelfth, . . . , records are blank; the third, eighth, thirteenth, . . . , records with the specification I2; the fourth, ninth, fourteenth, . . . , records are blank; and the fifth, tenth, fifteenth, . . . , records are blank.

On input, the same format descriptions apply. However, on input two slashes (//) indicate a record to be ignored. (The record is read, but not processed.)

If a single multiple-record FORMAT statement is required in which, for example, the first two records are unique and all remaining records are to be transferred to the same specification, the specification of the remaining records should be defined as a repetitive group by enclosing it in parentheses. For example,

FORMAT(I2,3E12.4/2F10.3,3F9.4/(10F12.4))

would transfer the first record with the specification I2,3E12.4, the second record with the specification 2F10.3,3F9.4, and all remaining records with the specification 10F12.4. The repetition starts at the last left parenthesis, including a repeater, if present.

If data items remain to be transferred after the format specification has been completely interpreted, the specification repeats after the last left parenthesis. Group repetition applies again if it is present. Consider this example.

FORMAT(3E10.3,2(I2,2F12.4,F28.17))

If more items are to be transferred after this format specification has been completely used, the specification repeats with I2 after the last left parenthesis. The 2 preceding the parenthesis, indicating group repetition, applies again.

Carriage Control

Carriage control characters must appear in the first position of the output record if the record is to be printed. The control character does not appear in the printed record. The valid characters used to achieve the desired results follow.

Character	*Result*
blank	No space before printing, that is, single space printing
0	One space before printing, that is, double space printing
1-9	Skip to channel 1-9 before printing as indicated

FORMAT Statements Read In at Object Time

A FORMAT may be specified for an I/O list at object time. In order that this be accomplished, two factors must be taken into account.

First, the name of the variable which will contain the FORMAT specification must appear in a DIMENSION statement, even if the array size is only 1.

Second, the format read in at object time under A-conversion must take the same form as a source program FORMAT statement, except that the word FORMAT is omitted. The variable format begins with a left parenthesis and ends with a right parenthesis.

Consider this example. A, B, and array C are converted and stored according to the FORMAT specifications that are read into the array FMT at object time.

```
      DIMENSION FMT (12)
    1 FORMAT (8A10)
      READ (1, 1) (FMT (I), I = 1, 8)
      READ (1, FMT) A, B, (C (I), I = 1, 5)
```

Assume that the first data card containing the FORMAT statement is of the form

(b2F10.8/b(E10.2))bb . . .

This implies that (b2F10.8/b will be stored in FMT(1) and (E10.2))bb will be stored in FMT(2), assuming $f + 2 = 10$. The remaining characters will be stored in FMT(3) . . . . This further implies that A and B will be read according to the F10.8 FORMAT specification and C(1), . . . , C(5) will be read according to the E10.2 FORMAT specification.

Edited Input Data

Edited input data to an object program is contained in records that conform to the following specifications:

1. The data must correspond in order, type, and field width to the field specifications in the FORMAT statement. Reading of data starts with the first character position.
2. Plus signs are indicated by a blank or a preceding + (12-zone punch). Minus signs are indicated by a preceding — (11-zone punch).
3. Blanks in numeric fields are regarded as zeros.
4. Numbers for E- and F-conversion can contain any number of digits, but only the high-order f digits will be retained. The number is rounded to f digits of accuracy. The absolute value of the number must be between the limits 10^{-100} and $(1 - 10^{-f}) \times 10^{99}$, or be zero. Numbers for I-conversion must be right justified (trailing blanks are regarded as zeros).
5. Numbers for E-conversion need not have four columns devoted to the exponent field. The start of the exponent field must be marked by an E, or if the E is omitted, by a + or — (not a blank). Valid forms for the exponent field are:

   E+02, Eb02, Eb2, E+2, E2, +02, +2, E—22, E—2, —2.

6. Numbers for E- and F-conversion need not have the decimal point punched. The format specification will supply the required decimal point. 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 a decimal point is punched, its position overrides the position indicated in the FORMAT specification.
7. A 7-8 punch in column one indicates an immediate return to the System Control Program.

Unedited Data

Unedited data may be stored on a disk unit or on magnetic tape. The length of each complete unedited record is defined by the number of items in the I/O list. Unedited records that are read from either a disk unit or magnetic tape must have been written by a Fortran WRITE statement. That is, the lists for the READ and WRITE statements must be of the same length. Further, the types of the list items must be in a one-to-one correspondence. For example, if the list of the WRITE statement is in the form

A, B, C

and the list of the READ statement is in the form

X, Y, Z

the types of A and X must match, the types of B and Y must match, and the types of C and Z must match.

The information is read or written using the internal representation of data values with no conversion.

The Fortran object-time routines may physically segment complete records into partial records.

General Input/Output Statements

The input/output devices available on the object machine are the only devices that can be referenced in an I/O statement. In particular, a disk unit must be available in order to use the DEFINE FILE, FIND, and the random form of the READ and WRITE statements. The sequential READ and WRITE, ENDFILE, BACKSPACE and REWIND statements can reference a disk unit area only if the area has not been referenced in a DEFINE FILE statement. Sequential operations on disk and tape are effectively the same.

The READ Statement

The READ statement designates input. This statement is used to transfer data from input devices to the computer.

General Forms.

READ *(i, n) list*
READ *(i) list*
READ *(j ' e, n) list*
READ *(j ' e) list*

i is an unsigned one digit integer constant or an integer variable that specifies a specific logical file to be used for data input.

n is the statement number of a FORMAT statement or a real array name that describes the data to be transferred.

list is an input list.

j is an unsigned one digit integer constant or an integer variable that specifies a specific logical file on a disk unit whose data input is to be accessed randomly.

' is a 4-8 punch (equivalent to the @ symbol).

e is an unsigned integer constant, integer variable, or integer expression that specifies a specific record within logical file *j*.

Examples.

```
READ (5, 10) A, B, (D (J), J = 1, 10)
READ (N, 10) K, D (J)
READ (3) (A(J), J = 1, 10)
READ (N) (A(J), J = 1, 10)
READ (6 ' 55, 15)
READ (I ' J, 22)
READ (I ' J + 5) X, Y, Z
```

The READ *(i,n) list* statement is used when the logical file contains edited information and is assigned to a card reader, or to a console printer, or to a tape unit, or to a disk unit whose records are to be selected sequentially. The statement causes the edited information to be read from the logical file *i* according to FORMAT statement *n*. Successive records are read in accordance with the FORMAT statement *n* until all the data items in the I/O list have been read, converted, and stored in the location specified by the I/O list.

The READ *(i) list* statement is used when the logical file contains unedited information and is assigned to a tape unit or to a disk unit whose records are to be processed sequentially. The statement causes the unedited information to be read from the logical file *i* starting with the record at the current position of the device to which logical file *i* is assigned. Only one record is read. The record is read completely only if the list specifies as many variables as the number of values in the record. Unedited records that are to be read in by a Fortran program should have been written by a Fortran program that used the same degrees of precision; that is, the values of *k* are the same for both programs, and the values of *f* are the same for both programs. An unedited record to be read can be divided into several parts by a Fortran I/O routine.

The READ *(j ' e,n) list* statement is used when the logical file contains edited information and is assigned to a disk unit whose records may be processed randomly. The statement causes the edited information to be read according to FORMAT statement *n* starting with record *e* within logical file *j*. Successive records are read in accordance with the FORMAT statement *n* until all the data items in the I/O list have been read, converted, and stored in the location specified by the I/O list. Each record should have no more characters than the maximum specified in the DEFINE FILE statement for the logical file *j*.

The READ *(j ' e) list* statement is used when the logical file contains unedited information and is assigned to a disk unit whose records may be processed randomly. The statement causes the unedited information to be read from record *e* within logical file *j*. Only one record is read. The record is read completely only if the list specifies as many variables as the number of values in the record. Unedited records that are to be read in by a Fortran program should have been written by a Fortran program that used the same degrees of precision; that is, the values of *k* are the same for both programs, and the values of *f* are the same for both programs. To contain all of the list data, the record to be read can be divided into several parts by a Fortran I/O routine, consistent with the description of logical file *j* in a DEFINE FILE statement. Each record should have no more data values than the maximum specified in the DEFINE FILE statement for the logical file *j*.

The WRITE Statement

The WRITE statement designates output. This statement is used to transfer data from the computer to output devices.

General Forms.

WRITE *(i,n) list*
WRITE *(i) list*
WRITE *(j ' e, n) list*
WRITE *(j ' e) list*

i is an unsigned one digit integer constant or an integer variable that specifies a specific logical file to be used for data output.

n is the statement number of a FORMAT statement or a real array name that describes the data to be transferred.

list is an output list.

j is an unsigned one digit integer constant or an integer variable that specifies a specific logical file on a disk unit that is to be accessed randomly for data output.

' is a 4-8 punch (equivalent to the @ symbol).

e is an unsigned integer constant, integer variable, or integer expression that specifies a specific record within logical file *j*.

Examples.

```
WRITE (6, 10)A, B, (C(J), J = 1, 10)
WRITE (N, 11) K, D (J)
WRITE (2) (A(J), J = 1, 10)
WRITE (M) A, B, C
WRITE (9 ' 55, 15)
WRITE (I ' J, 22)
WRITE (I ' J + 5)
```

The WRITE *(i,n) list* statement is used when the logical file is to contain edited information and is assigned to a card punch, or to a console printer, or to a tape unit, or to a disk unit on which records are to be written sequentially. The statement causes the edited information to be output on logical file *i* according to FORMAT statement *n*. Successive records are output in accordance with the FORMAT statement *n* until all the data items in the I/O list have been converted and output.

The WRITE *(i) list* statement is used when the logical file is to contain unedited information and is assigned to a tape unit or to a disk unit on which records are to be written sequentially. The statement causes the unedited information to be written on logical file *i* starting with the record at the current position of the device to which logical file *i* is assigned. Only one record is written. The record is written completely only if the list specifies as many variables as the number of values in the record. The record to be written can be divided into several parts by a Fortran I/O routine.

The WRITE *(j ' e,n) list* statement is used when the logical file is to contain edited information and is assigned to a disk unit on which records are to be written randomly. The statement causes the edited information to be written according to FORMAT statement *n* starting with record *e* within logical file *j*. Successive records are written in accordance with the FORMAT statement *n* until all the data items in the I/O list have been converted and written. Each record should have no more characters than the maximum specified in the DEFINE FILE statement for the logical file *j*.

The WRITE *(j ' e) list* statement is used when the logical file is to contain unedited information and is assigned to a disk unit on which records are to be written randomly. The statement causes the unedited information to be written starting with record *e* within logical file *j*. Only one record is written. The record is written completely only if the list specifies as many variables as the number of values in the record. To contain all of the list data, the record to be written can be divided into several parts by a Fortran I/O routine, consistent with the description of logical file *j* in a DEFINE FILE statement. Each record should have no more data values than the maximum specified in the DEFINE FILE statement for file *j*.

Manipulative Input/Output Statements

The FIND, END FILE, REWIND, and BACKSPACE statements manipulate the logical files that are used by the object program. The END FILE, REWIND, and BACKSPACE statements must *not* be used to reference logical files that are referenced in a DEFINE FILE statement, or logical files that are assigned to a card reader, a card punch, or a printer.

The FIND Statement

The FIND statement is used to initiate the positioning of the access mechanism when the logical file is assigned to a disk unit and has been described in a DEFINE FILE statement.

General Form. FIND *(j ' e)*

j is an unsigned one digit integer constant or an integer variable that specifies a specific logical file on a disk unit that contains data input or output to be selected randomly.

' is a 4-8 punch (equivalent to the @ symbol).

e is an unsigned integer constant, integer variable, or integer expression that specifies a specific record within logical file *j*.

The purpose of the FIND statement is to enable the programmer to increase the speed at which the object program is executed. The FIND statement may substantially reduce the seek time required by the next READ or WRITE statement, provided that it references the same record *e* within logical file *j*. When the statement is used, it starts positioning the access mechanism to locate the record *e* within logical file *j* while permitting computation to proceed concurrently. The greater the separation between the FIND statement and the following READ or WRITE statement, the greater the concurrent processing time.

The END FILE Statement

General Form. END FILE *i*

i is an unsigned integer constant or an integer variable that refers to a specific logical file.

Examples.

```
END FILE 3
END FILE N
```

The END FILE i statement causes an end-of-file indication to be written on the logical file i. If the logical file is assigned to a tape unit, a tape mark is written. If the logical file is assigned to a disk unit, an end-of-file record, 1bEOF, is written. Either indication is recognized as an end-of-file condition when sensed by a READ statement, and can be tested by using the standard subprogram EOF.

The REWIND Statement

General Form. REWIND i

i is an unsigned integer constant or an integer variable that refers to a specific logical file.

Examples.

```
REWIND 3
REWIND N
```

The REWIND i statement causes logical file i to be initialized to its starting point. If the logical file is assigned to a tape unit, the tape will be rewound. If the logical file is assigned to a disk unit, the START address of the disk area is obtained.

The BACKSPACE Statement

General Form. BACKSPACE i

i is an unsigned integer constant or an integer variable that refers to a specific logical file.

Example.

```
BACKSPACE 3
```

The BACKSPACE i statement causes logical file i to "backspace" one complete record. If the logical file contains edited records, one record is a complete record. If the logical file contains unedited records, there can be more than one partial record making up a complete record.

If the logical file is assigned to a tape unit, the tape is physically backspaced. If the logical file is assigned to a disk unit, a disk-address is appropriately decreased.

Input/Output Specification Statement

The DEFINE FILE Statement

The DEFINE FILE statement is used for logical files that are assigned to disk units such that records may be accessed randomly. The tape resident Fortran system cannot use a disk unit. Specifically, the DEFINE FILE statement must be used when any of the following input/output statements are used in the program.

```
READ (j ' e,n) list
READ (j ' e) list
WRITE (j ' e,n) list
WRITE (j ' e) list
FIND (j ' e)
```

General Form. DEFINE FILE j_1 (m_1, l_1, f_1, v_1), j_2 $(m_2, l_2, f_2, v_2), \ldots, j_i$ (m_i, l_i, f_i, v_i)

j_i is an integer constant that refers to a specific logical file located on a disk unit and can be 1 through 9. This is the file name that is referenced in a corresponding READ or WRITE statement.

m_i is an integer constant that defines the maximum number of records in the logical file j_i.

l_i is an integer constant that defines the maximum length of each record in logical file j_i. If the records are edited (f_i is E), the length l_i is the maximum number of characters in each record, and can be equal to or less than 200. If the records are unedited (f_i is U), the length l_i is the maximum number of data values in each record. The data values can be type real, type integer, and type logical.

f_i is either E or U. E indicates that the data in logical file j_i is edited. When the data is edited, the data must be read or written with statements in the form READ *(j'e,n) list* or WRITE *(j'e,n) list*. U indicates that the data in logical file j_i is unedited. When the data is unedited, the data must be read or written with statements in the form READ *(j'e) list* or WRITE *(j'e) list*.

v_i is a nonsubscripted integer variable name. This variable is set at the end of each READ or WRITE statement that references logical file j_i. The value of the variable is set to the value of the next available record following those records read or written in logical file j_i. During the execution of a FIND statement, the value of the variable v_i is set equal to the value of the expression specifying the record number in the FIND statement. It must appear in COMMON or be passed as a subprogram parameter if it is to be referenced by a subprogram at object time.

Example.

```
DEFINE FILE 4(100, 120, E, INDEX4), 5(50, 10, U, INDEX5),
6(300, 150, U, INDEX6)
```

In the example, logical file 4 will consist of not more than 100 records, each of which is not more than 120 characters in length. Any information within the file is to be edited according to some FORMAT statement(s) when read or written. Further, INDEX4 (type integer) will be assigned a value reflecting the record number following the last processed after each READ or WRITE statement.

Note that two sectors of disk storage will be required for each record, implying that $100 \times 2 = 200$ sectors must be available to be assigned to this logical file. If each record had been not more than 100 characters in length, only one sector of disk storage would have been required for each record.

Logical file 5 will consist of not more than 50 records, each of which contains not more than 10 data values of type real, type integer, or type logical. The information within the logical file must be read or written with the unedited form of READ and WRITE statements. Further, INDEX5 (type integer) will be assigned a value reflecting the record number following the last processed after each READ or WRITE statement.

Note that two, or some higher integral multiple of two, sectors of disk storage will be used for each record. Within each 2-sector partial record, 190 data characters are available. Therefore, the number of data values that can be contained within each 2-sector partial record is the highest integer which, when multiplied by the maximum of $f + 2$ and k, will be not greater than 190. For logical file 5, assuming that $f + 2 = 10$ and $k = 5$, the ten data values of a record will be contained within two sectors, implying that $50 \times 2 = 100$ sectors must be available to be assigned to this logical file.

Logical file 6 will consist of not more than 300 records, each of which contains not more than 150 data values. Assuming that $f + 2 = 10$ and $k = 5$, it follows that 19 data values can be contained within each two-sector partial record, and eight partial records constitute a complete record. Thus, $300 \times 8 \times 2 = 4800$ sectors must be available to be assigned to this logical file.

DEFINE FILE statements can appear anywhere in the program. They are used in conjunction with the FIND and the random form of READ and WRITE statements. For example, with DEFINE FILE 4(100,120,E,INDEX4), the following statements might be coded to cause every eleventh record to be read.

```
DEFINE FILE 4(100, 120, E, INDEX4)
      .
      .
      .
IVAR = INDEX4 + 10
      .
      .
      .
READ (4 ' IVAR, n) list
```

When programs are to be executed together, all necessary DEFINE FILE statements must appear in the main program. These files that are defined can be referenced by file name in READ, WRITE, or FIND statements in the subprograms. DEFINE FILE statements that appear in subprograms are ignored; however, they can be included for documentary purposes.

A complete record can contain an integral multiple of partial records. The complete record begins with the first partial record. If the complete record does not use all the partial records, all unused parts are ignored.

Logical Files Used for Input/Output

A set of 14 logical files has been defined by the Fortran system. Ten of these logical files have been assigned a unique single-digit integer and are available to the Fortran programmer. It is by referencing these numbers that the Fortran input/output operations are accomplished. The correspondence between the logical file reference and the actual physical input/output unit is established by the Fortran system just prior to executing the object program.

A complete description of the logical files that are defined by the Fortran system is contained in the program specifications section of this publication.

Subprograms—Function and Subroutine Statements

The Fortran language defines four general types of subprograms: library functions, FUNCTION subprograms, library subroutines, and SUBROUTINE subprograms. Library functions and subroutines are predefined subprograms that are a part of the Fortran library. FUNCTION subprograms are functions defined by a FUNCTION statement and its associated subprogram. SUBROUTINE subprograms are subroutines defined by a SUBROUTINE statement and its associated subprogram.

Functions differ from subroutines in that functions always return a single result that is the value of the function to the calling program, whereas subroutines may return any number of values to the calling program, and do not have an actual value of their own.

Advantages of Subprograms

One of the advantages of using subprograms is that the main program and its several associated subprograms can be compiled separately. Thus, a program can consist of a short main program and any number of subprograms. Changes or error correction can then be made by recompiling only the affected program.

Other advantages are that any subprogram can be placed in the Fortran library for use with other programs. Subprograms also permit more than one programmer to be simultaneously writing a large program.

Naming Subprograms

A subprogram name consists of one to six alphameric characters. No special characters can appear in the name. The first character must be alphabetic.

The rules for specifying the type of the function value are the same as those for naming variables.

The type (real or integer) of a predefined library function is already specified (Figure 8) and need not be defined by the user.

The type of a FUNCTION subprogram can be indicated implicitly (real or integer) by the initial character of the name or explicitly (real, integer, or logical) by a type statement. In the latter case, the implicit type is overridden by the explicit specification.

The type of a SUBROUTINE subprogram is not defined since there is no actual value associated with a subroutine.

Predefined Subprograms

This section describes the predefined functions and subroutines that are incorporated within the Fortran library.

Library Functions

Incorporated within the Fortran library is a set of predefined functions. These functions are shown in Figure 8. Note that the type (real or integer) of each function and argument is predefined and cannot be changed by the user.

Library Subroutines

Incorporated within the Fortran library is a set of predefined subroutines. These subroutines are given in the following list. Included in the list are the machine indicator test subroutines. *i* is an integer expression, *j* is an integer variable, *m* is a logical variable, and *r* is a real variable. *sense light* refers to symbolic switches with the values *off* and *on.*

These subroutines are referenced by CALL statements.

General Form	*Function*
SLITE *(i)*	If $i = 0$, all sense lights are turned off. If $i = 1, 2, 3,$ or 4, the corresponding light is turned on.
SLITET *(i, j)*	Sense light *i* (1, 2, 3, or 4) is tested and turned off. The variable *j* is set to 1 if *i* was on or *j* is set to 2 if *i* was off.
SSWTCH *(i, j)*	Sense switch *i* is tested. The variable *j* is set to 1 if *i* is off or *j* is set to 2 if *i* is on. The sense switch feature is required on the system for execution. $i = 1$-6 corresponds to sense switches B-G.
LINK (3H*phase-name*) *or* LINK *(r)*	*phase-name* stands for a three-character phase-name that indicates the presence of a program in the absolute format stored as a phase on the SYSTEM file of the Fortran system. The phase (program) is loaded and execution of the program begins. *r*, a real variable, has an alphameric value whose rightmost three characters are used as the phase-name.
EOF *(m)*	*m*, a logical variable, is assigned a value of .TRUE. if the most recently executed READ statement sensed an end-of-file condition; otherwise, *m* is .FALSE.. An end-of-file condition is indicated by 1EOFb, a tape mark, or an empty card reader. If a READ statement is issued after the last-card condition, no halt occurs; rather, blank input is interpreted. The end-of-file condition must be tested before the next I/O statement is executed.

Defining Subprograms

This section describes the FUNCTION statement that is used to name FUNCTION subprograms, and the SUBROUTINE statement that is used to name SUBROUTINE subprograms.

The FUNCTION Statement

The FUNCTION statement is used to name FUNCTION subprograms and must be the first statement of a FUNCTION subprogram. It cannot appear anywhere else in the subprogram nor can it appear in a main program.

General Forms.

FUNCTION *name* $(a_1, a_2, \ldots, a_n)$
REAL FUNCTION *name* $(a_1, a_2, \ldots, a_n)$
INTEGER FUNCTION *name* $(a_1, a_2, \ldots, a_n)$
LOGICAL FUNCTION *name* $(a_1, a_2, \ldots, a_n)$

name is the symbolic name of the single-valued function subprogram.

$a_1, a_2, \ldots, a_n$ are the dummy arguments of the function and can be nonsubscripted variable names, or array names, or the dummy names of SUBROUTINE, or other FUNCTION subprograms. There must be at least one argument in a FUNCTION subprogram.

The type of function can be explicitly stated by the inclusion of the word REAL, INTEGER, or LOGICAL before the word FUNCTION, as shown in the preceding formats.

Examples.

```
FUNCTION ARCSIN (RADIAN)
REAL FUNCTION ROOT (A, B, D)
INTEGER FUNCTION CONST (ING, SG)
LOGICAL FUNCTION IFTRU (D, E, F)
```

Function	Definition	Number of Arguments	Name	Type of Argument	Type of Function
Exponential	$_{e}$Arg	1	EXP	Real	Real
Natural logarithm	$\log_e$(Arg)	1	ALOG	Real	Real
Common logarithm	$\log_{10}$(Arg)	1	ALOG10	Real	Real
Arctangent	arctan (Arg)	1	ATAN	Real	Real
Trigonometric sine	sin (Arg)	1	SIN	Real	Real
Trigonometric cosine	cos (Arg)	1	COS	Real	Real
Square root	(Arg) $^{1/2}$	1	SQRT	Real	Real
Absolute value	|Arg|	1	ABS IABS	Real Integer	Real Integer
Truncation	Sign of Arg applied to largest integer$\leq$|Arg|	1	AINT INT	Real Real	Real Integer
Remaindering (see note below)	Arg_1 (mod Arg_2)	2	AMOD MOD	Real Integer	Real Integer
Choosing largest algebraic value	Max (Arg_1, Arg_2, ...)	$\geq$2	AMAX0 AMAX1 MAX0 MAX1	Integer Real Integer Real	Real Real Integer Integer
Choosing smallest algebraic value	Min (Arg_1, Arg_2, ...)	$\geq$2	AMIN0 AMIN1 MIN0 MIN1	Integer Real Integer Real	Real Real Integer Integer
Float	Conversion from integer to real	1	FLOAT	Integer	Real
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of sign	Sign of Arg_2 applied to |Arg_1|	2	SIGN ISIGN	Real Integer	Real Integer
Positive difference	Arg_1-Min (Arg_1, Arg_2)	2	DIM IDIM	Real Integer	Real Integer

Note: The function MOD (Arg_1, Arg_2) is defined as $Arg_1 - \left[Arg_1/Arg_2\right] Arg_2$ where $\left[x\right]$ is the integral part of x.

Figure 8. Predefined Library Functions

The FUNCTION subprogram can contain any Fortran statement except a SUBROUTINE statement or another FUNCTION statement.

The name of the function must appear at least once as a variable on the left side of an arithmetic statement, or as an element of an input list.

Consider the following example.

```
FUNCTION CALC (A, B)
    .
    .
    .
CALC = Z + B
    .
    .
    .
RETURN
    .
    .
    .
END
```

In this example, the value of CALC is computed and its value is returned to the calling statement.

The FUNCTION subprograms are logically terminated during execution by a RETURN statement. They are physically terminated during compilation by an END statement.

The arguments in a FUNCTION statement can be considered as dummy variable names that are replaced at the time of execution by the actual arguments supplied in the function reference in the calling program. The actual arguments must correspond in number, order, and type with the dummy arguments, and can be constants, nonsubscripted variables, subscripted variables, expressions, array names, or other subprogram names.

If the value of a dummy argument is changed by the subprogram, then the actual argument in the calling program is also changed. In this case, the actual argument must be a nonsubscripted variable, a subscripted variable, or an array name. The actual argument cannot be a constant, an expression, or a subprogram name.

Dummy arguments cannot appear in an EQUIVALENCE statement in the FUNCTION subprogram.

Dummy arguments can be subprogram names. The corresponding actual arguments in the calling program must be names that have appeared in an EXTERNAL statement in the calling program. The dummy arguments can also appear in an EXTERNAL statement. In this way, a subprogram name used as an actual argument in a calling program can be passed to a subprogram, which in turn can pass it on to another subprogram.

If a dummy argument is an array name, a DIMENSION statement or a COMMON (with dimensions) statement for that array must appear in the FUNCTION subprogram. Further, the corresponding actual argument must be a dimensioned array name.

If a dummy array name appears in a COMMON statement, then the actual argument in the calling program must be an array name that appears in the identical place in the COMMON statement.

If a dummy variable name appears in a COMMON statement, then the value of the actual argument in the calling program replaces the value of the dummy name in COMMON immediately upon entry to the FUNCTION subprogram.

The SUBROUTINE Statement

The SUBROUTINE statement is used to name SUBROUTINE subprograms and must be the first statement of a SUBROUTINE subprogram. It cannot appear anywhere else in the subprogram nor can it appear in a main program.

General Form. SUBROUTINE *name* $(a_1, a_2, \ldots, a_n)$

name is the name of the subprogram.

a_1 $a_2, \ldots, a_n$ are the dummy arguments, and can be nonsubscripted variable names, or array names, or the dummy name of another SUBROUTINE or FUNCTION subprogram. There must be at least one argument in a SUBROUTINE subprogram.

Examples.

```
SUBROUTINE MATMPY (A, N, M, B, L, J)
SUBROUTINE QDRTIC (B, A, C, ROOT1, ROOT2)
```

The SUBROUTINE subprogram can contain any Fortran statement except a FUNCTION statement or another SUBROUTINE statement.

The SUBROUTINE subprograms are logically terminated during execution by a RETURN statement. They are physically terminated during compilation by an END statement.

The arguments in a SUBROUTINE statement can be considered as dummy variable names that are replaced at the time of execution by the actual arguments supplied in the CALL statement. The actual arguments must correspond in number, order, and type with the dummy arguments, and can be constants, nonsubscripted variables, subscripted variables, expressions, array names, other subprogram names or an alphameric field. An actual argument that is an alphameric field must correspond to a dummy argument that is a real variable.

If the value of a dummy argument is changed by the subprogram, then the actual argument in the calling program is also changed. In this case, the actual argument must be a nonsubscripted variable, a subscripted variable, or an array name. The actual argument cannot be a constant, an expression, a subprogram name, or an alphameric field.

Dummy arguments cannot appear in an EQUIVALENCE statement in the SUBROUTINE subprogram. When a dummy argument appears in a SUBROUTINE statement, it cannot also appear in a COMMON statement.

Dummy arguments can be subprogram names. The corresponding actual arguments in the calling program must be names that have appeared in an EXTERNAL statement in the calling program. The dummy arguments can also appear in an EXTERNAL statement. In this way, a subprogram name used as an actual argument in a calling program can be passed to a subprogram, which in turn can pass it on to another subprogram.

If a dummy argument is an array name, a DIMENSION statement or a COMMON (with dimensions) statement for that array must appear in the SUBROUTINE subprogram. Further, the corresponding actual argument must be a dimensioned array name.

If a dummy array name appears in the COMMON statement, then the actual argument in the calling program must be an array name that appears in the identical place in the COMMON statement.

If a dummy variable name appears in a COMMON statement, then the value of the actual argument in the calling program replaces the value of the dummy name in COMMON immediately upon entry to the SUBROUTINE subprogram.

The RETURN Statement

General Form. RETURN

This is the exit from any subprogram. It returns control to the calling program. The RETURN statement is the logical end of the subprogram. There can be one or more RETURN statements in the subprogram.

Subprogram Names as Arguments

FUNCTION and SUBROUTINE subprogram names can be used as the actual arguments in the calling program. In order to distinguish these subprogram names from ordinary variables when they appear in an argument list, their names must appear in an EXTERNAL statement, even if they are also used in actual references within the same program (see *The Type Statements*).

Examples.

```
EXTERNAL SIN
CALL SUBR (A, SIN, B)
```

Using Subprograms

This section describes the method for referencing functions and subroutines.

Using Functions

A function is referenced (or called) by using its name followed by its actual arguments in parentheses as an operand in an expression.

General Form. Name $(a_1, a_2, \ldots, a_n)$

Name is the name of the function. The predefined function names are shown in Figure 8.

$a_1, a_2, \ldots, a_n$ are the actual arguments of the function. The arguments can be arithmetic or logical expressions, constants, nonsubscripted variables, subscripted variables, array names, or other subprogram names. The number of arguments required for predefined functions is shown in Figure 8.

A program which references a logical function must have the name of that function in a logical type statement.

Each actual argument must have been assigned a value before the function reference.

Examples.

```
Z = SIN (X) + COS(Y)*Z1
T = 5.*ARBFNC (55.2/SQRT (10.3*R), ABS (3. +5))
```

In Figure 8 note that the type (real or integer) of each built-in function is predefined and cannot be changed by the user. Note also that the type of each argument is predefined.

Using Subroutines — The CALL Statement

The CALL statement is used only to call a SUBROUTINE subprogram.

General Form. CALL *name* $(a_1, a_2, \ldots, a_n)$

name is the symbolic name of a SUBROUTINE subprogram.

$a_1, a_2, \ldots, a_n$ are the actual arguments that are being supplied to the SUBROUTINE subprogram.

Each actual argument must have been assigned a value before the reference to the subroutine.

Examples.

```
CALL MATMPY (X, 5, 10, Y, 7, 2)
CALL QDRTIC (9. 732, Q/4. 536, R-S**2.0, X1, X2)
```

The CALL statement transfers control to the SUBROUTINE subprogram and replaces the dummy variables with the actual arguments that appear in the CALL statement.

The actual arguments in a CALL statement can be any one of the following.

1. Any type of constant.
2. Any type of subscripted or nonsubscripted variable.

3. An arithmetic or logical expression.
4. Any array name.
5. The name of a FUNCTION or SUBROUTINE subprogram. The name must also appear in an EXTERNAL type statement.
6. Alphameric characters (allowed only for subroutines). Such arguments must be preceded by *n*H, where *n* is the number of characters included in the argument and must be less than or equal to $f + 2$. When $n < f + 2$, the characters are right justified. For example, 9HENDbPOINT. Blank spaces and special characters are considered in the character count when used in alphameric fields.

The actual arguments in a CALL statement must agree in number, order, type, and array size (except as explained in *The DIMENSION Statement*) with the corresponding arguments in the FUNCTION or SUBROUTINE statement of the called subprogram.

Segmenting Programs

Large programs can be segmented into several phases, or overlays. Each phase must consist of a main program and any required subprograms.

Each phase is given a unique three-character phase name by using a compiler option control card when the main program is compiled. The phase is called by using a CALL LINK (3H*phase-name*) or CALL LINK *(r)* statement in the calling program.

Data values can be passed between phases by using logical files or the COMMON area. Therefore, the COMMON area used for passing data values between phases must be the same. The longest phase must not overlap this COMMON area.

In order to use the segmenting capability of the Fortran system, perform the following operations.

1. Compile each main program, specifying its unique program name. Use a compiler option control card to specify the unique program name. If appropriate, compile any required subprograms. See *FORTRAN RUN.*
2. Specify that an absolute deck be punched as a result of the LOADER RUN that loads the main program and any subprograms. Use a loader output option control card to get an absolute deck. See *LOADER RUN.*
3. Perform a user-update job to place the program on the SYSTEM file. Once the program is stored as a phase on the SYSTEM file, the phase can be called at any time. See *User-Update Jobs* and *PRODUCTION RUN.*
4. Use the CALL LINK (3H*phase-name*) or CALL LINK *(r)* statement to call the phase.

When the phase is called, it is read into core storage beginning at address 950. It extends upward in core storage to include the entire phase. The COMMON area defined in the program or a subprogram is not included as a part of the phase.

The same phase can be called more than one time. The sequence of phase calling does not necessarily have to correspond to the order in which the phases are placed on the SYSTEM file. However, on a tape system, having the phases placed on the SYSTEM file in the same order in which they are to be called will optimize phase-call time.

Fortran Source Program

Source Program Statements and Sequencing

The order in which the source program statements of a Fortran program are executed follows these rules:

1. Control originates at the first executable statement. The specification statements and the FORMAT, FUNCTION, SUBROUTINE, DEFINE FILE, and END statements are nonexecutable. In questions of sequencing, they can be ignored.
2. If control was with statement S, then control will pass to the statement indicated by the normal sequencing properties of S. If, however, S is the last statement in the range of one or more DO's that are not yet satisfied, then the normal sequencing of S is ignored and DO-sequencing occurs.

The normal sequencing properties of each Fortran statement follows:

Statement	*Normal Sequencing*
a = b	Next executable statement
BACKSPACE	Next executable statement
CALL	First statement of called subprogram
COMMON	Nonexecutable
CONTINUE	Next executable statement
DATA	Nonexecutable
DEFINE FILE	Nonexecutable
DIMENSION	Nonexecutable
DO	DO-sequencing, then the next executable statement
END	Terminates source program

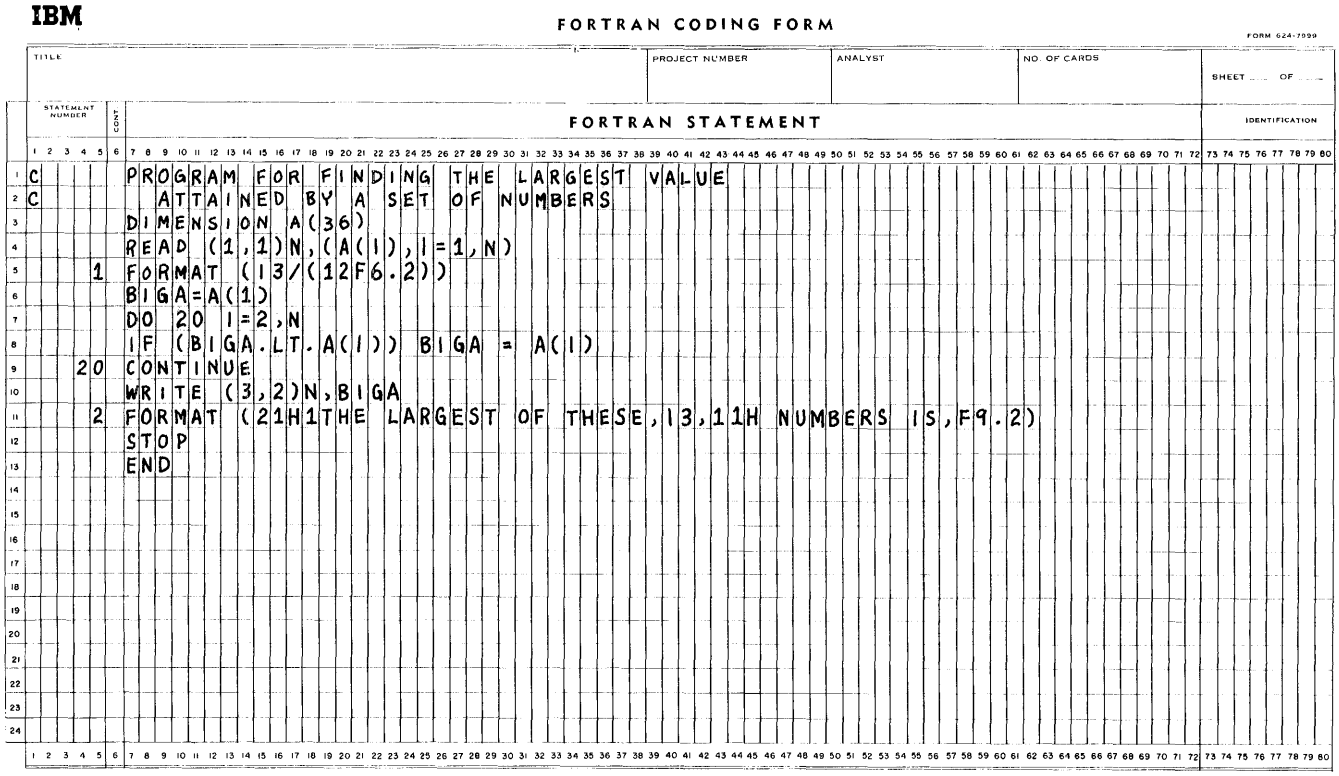


Figure 9. Fortran Source Program

Statement	*Normal Sequencing*
END FILE	Next executable statement
EQUIVALENCE	Nonexecutable
EXTERNAL	Nonexecutable
FIND	Next executable statement
FORMAT	Nonexecutable
FUNCTION	Nonexecutable
GO TO n	Statement n
GO TO (n_1, n_2, . . . , n_m), i	Statement n_i
IF (t) s	Statement s or next executable statement if t is true or false, respectively
INTEGER	Nonexecutable
LOGICAL	Nonexecutable
PAUSE	Next executable statement
READ	Next executable statement
REAL	Nonexecutable
RETURN	The first statement or part of a statement, following the reference to this program
REWIND	Next executable statement
STOP	Terminates object program
SUBROUTINE	Nonexecutable
WRITE	Next executable statement

Writing the Source Program

The statements of a Fortran source program are usually written on a standard *Fortran Coding Form,* Form 624-7999. A sample Fortran source program, coded on the standard coding form, is shown in Figure 9. This program selects the largest value from an array of numbers, identified by the variable name A.

Columns 1-5 of the first line of a statement may contain a statement number that identifies the statement. This number must be an unsigned integer constant less than 100,000. Blanks and leading zeros are ignored in these columns. For example, bbb50 is the same as b5bb0 and 5bbb0. A statement must not be numbered zero. All statement numbers must be unique. These statement numbers do not have to be in any sequence or order. For example, the first statement of a program may be given statement number 100 and the 50th statement in a source program may be given statement number 1. These statement numbers are used, for example, in DO loops to indicate the range of the DO loop, in the GO TO statement, and to refer to FORMAT statements. Superfluous statement numbers should be kept to a minimum.

A statement can be continued on as many as nine additional lines. Any line with a non-blank, non-zero column 6, is considered to be a continuation of the preceding line. The actual character used in column 6 does not have any significance. The first continuation card could have a 9 in column 6, the second card an A, the third a 2, and so on.

Columns 7-72 contain the actual Fortran statements. Blanks are ignored except in an H-field of a FORMAT statement, a DATA statement, or a CALL statement.

Statements with a C in column 1 are not processed by the Fortran compiler, but the statements appear in the source program listing as comments. If there is a C in column 1, columns 2-72 can be used for comments. Comment cards cannot appear between continuation cards of a statement. Comment cards cannot immediately precede the FUNCTION or SUBROUTINE statements.

Columns 73-80 are not processed and can be used for identification.

The order of execution of the source program statements is governed by the sequencing described in the preceding section.

Checking the Source Program

An early successful compilation of a Fortran source program is more likely if the coding is checked against the following list of commonly-made errors.

Item to Check	*Coding Error*
A-conversion	Field width, w, exceeds the word size, $f + 2$.
Arithmetic expressions	Real and integer numbers, both constants and variables, mixed in invalid combinations. Often, a real constant is written without a decimal point.
DO parameters	Subscripted integer variable or expression used as a parameter.
FORMAT statements	FORMAT specifications and I/O list not compatible.
Fortran language	Misspelled Fortran-language word such as EQUIVALENT instead of EQUIVALENCE.
H-conversion	Incorrect count for n of nH.
Program flow	Statement transfers into the range of a DO. Unreferenced statement after a GO TO, RETURN OR STOP. END statement encountered in program flow.
Statement numbers	Use of same statement number more than one time. Absence of a referenced statement number.
Subprograms	FUNCTION or SUBROUTINE statement missing at beginning of a subprogram; RETURN statement missing; END statement missing.
Subprogram names	Name is same as a variable name used in the program.
SUBROUTINE statement arguments	Dummy arguments that are subscripted or equivalenced variables.
Subscripted variables	Each subscripted variable, including those in lists, does not appear in a DIMENSION statement.
Variables	Variables must be defined in a read I/O list, on the left-hand side of an arithmetic statement, or in a DATA statement before being used on the right-hand side of an arithmetic statement, in a CALL statement, or in a WRITE I/O list.

Punching the Source Program

The Fortran statements, prepared as described in *Writing the Source Program,* are normally punched into the standard Fortran card, Form 888157, shown in Figure 10.

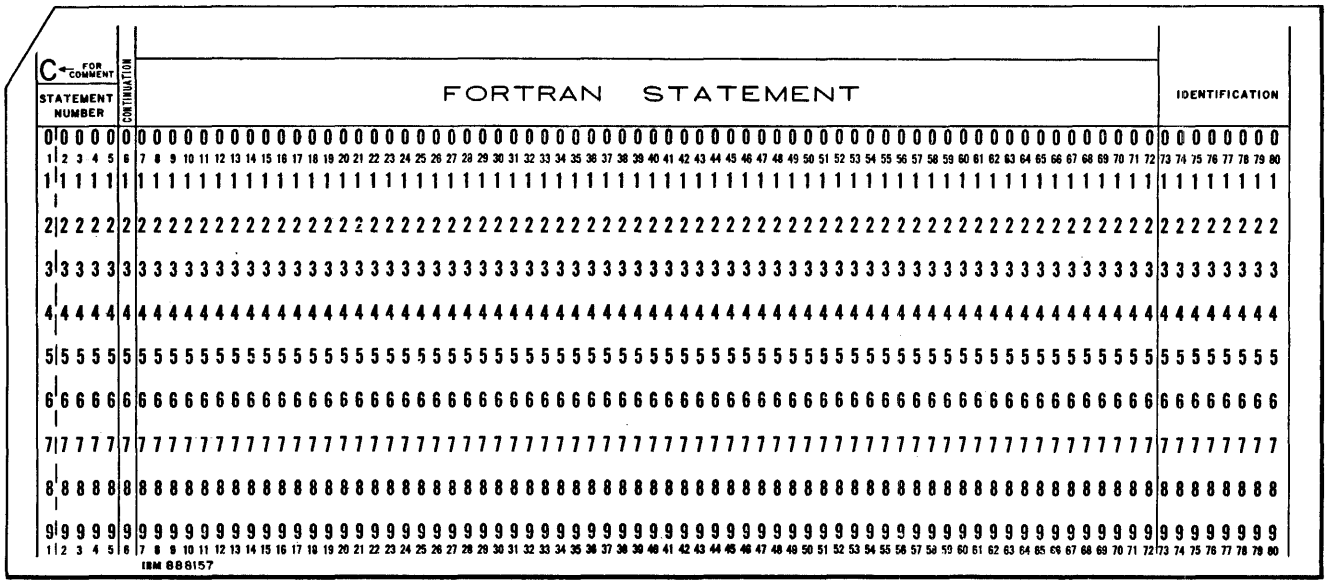


Figure 10. Fortran Source Card

Table of Source Program Characters

The following table indicates the list of characters in character Set H that can be used in a Fortran source program. The equivalent characters in character Set A are also given. The characters are shown in collating sequence.

Set H (Fortran Characters)	*Set A*	*Card Punches*
Blank	Blank	Blank
.	.	12-3-8
)	□	12-4-8
+	&	12
$	$	11-3-8
*	*	11-4-8
–	–	11
/	/	0-1
,	,	0-3-8
(	%	0-4-8
=	#	3-8
'	@	4-8
A	A	12-1
B	B	12-2
C	C	12-3
D	D	12-4
E	E	12-5
F	F	12-6
G	G	12-7
H	H	12-8
I	I	12-9
J	J	11-1
K	K	11-2
L	L	11-3
M	M	11-4
N	N	11-5
O	O	11-6
P	P	11-7
Q	Q	11-8
R	R	11-9
S	S	0-2
T	T	0-3
U	U	0-4
V	V	0-5
W	W	0-6
X	X	0-7
Y	Y	0-8
Z	Z	0-9
0	0	0
1	1	1
2	2	2
3	3	3
4	4	4
5	5	5
6	6	6
7	7	7
8	8	8
9	9	9

Note: The character $ can be used in Fortran only in an H-conversion field.

The Fortran Processor Program is a language processing system that operates entirely under control of the System Control Program. Together, the Fortran Processor Program and the System Control Program make up the Fortran system.

The Fortran system translates source program statements written in the Fortran language into machine-language instructions. Object-program execution is also under control of the Fortran system. In addition to these translation and execution functions, the Fortran system provides these additional features.

Expanding the Fortran Library. A Fortran library is defined by the system that contains commonly used subroutines and functions, such as sine, cosine, and logarithms. The Fortran system provides the capability of expanding the library to include additional user-supplied subroutines. In addition, if the system is disk-oriented, the user can relocate the library to an area of his choice in disk storage.

Changing Input/Output Devices. The Fortran system provides the user with the option of changing the form of input to and output from specific jobs. In order that the Fortran system operate at a machine-independent level, a set of logical files that are used for input/output operations has been defined. Although these logical files are assumed by the System Control Program to be assigned to a defined set of input/output devices, the user can change these assignments according to his particular needs.

Stacking of Jobs. Under control of the System Control Program, it is possible to process a sequence of jobs without regard to the type of processing that is being performed. For example, source programs can be compiled and object programs can be executed, all in one stack. However, stacking may only be accomplished on the card reader, card punch, or printer.

Building Object-Program Libraries in Mass Storage. By using a particular logical file (SYSTEM) defined by the Fortran system, it is possible to build an object-program library in mass storage (disk storage or magnetic tape). Because each of these object programs is identified by a unique three-character name, the user has immediate access to any of the programs. Object programs can be inserted or deleted from the library at the discretion of the user. If a program is to be used frequently, using the object-program library substantially reduces program load time (as opposed to loading from cards) and eliminates excessive handling of punched-card object decks. The use of the object-program library at execution time allows the user to segment a large program into sections or overlays.

The Fortran System

The Fortran system built by the user contains the System Control Program and the Fortran Processor Program.

System Control Program. The System Control Program is the controlling element of the system. Its main functions are to analyze control card information, transfer control to the appropriate portion of the system, and to perform actual input/output operations when a logical file is referenced, either during compilation or execution.

Fortran Processor Program. The Fortran Processor Program translates source programs, written in the Fortran language, into machine language object programs. Object programs are executed under control of the Fortran system.

System Control Program

System operations can be started by a deck of cards supplied by IBM. This deck, called the *card boot,* prepares core storage and reads in the first portion of the System Control Program from mass storage (disk or tape). Subsequently, the entire resident portion of the System Control Program is read into lower core storage.

System operations may also be initiated on the tape system by pressing TAPE LOAD and then START on the computer console when the SYSTEM file resides on tape unit 1.

All control functions for the system are accomplished by the System Control Program. These functions include:

Assigning Input/Output Devices. Each logical file is assigned to a corresponding user-specified physical input/output device.

Controlling Input/Output Devices. Physical manipulation of the corresponding input/output device occurs when the currently active program references a logical file.

Updating the System. Updating the system to the latest modification level or version is performed by the System Control Program.

Selecting Appropriate Processor Runs. From control cards supplied by the user, the System Control Program is able to determine the operations necessary for the completion of a job. For example, a source program is coded in the Fortran language and the user specifies the end result of processing to be a machine-language object program in relocatable format. This would require that processing be performed only by the Fortran compiler. The control card says in effect that the phases of the Fortran compiler are to be called by the System Control Program. The System Control Program reads the control card and calls the Fortran compiler. Processing takes place, and at completion, control reverts to the System Control Program which reads the control card for the next job.

The remainder of this section describes the logical files that are defined by the system, and the control cards required for system operations.

Logical Files

In order that input/output functions operate at a machine-independent level, a set of logical files has been defined by the Fortran system. These logical files are used for input/output operations. Each file has a specific function and is assigned by the System Control Program to a particular input/output device. The user can alter the file-assignments temporarily by using ASGN (assign) control cards.

The use of the logical files during compilation and execution times is described in this section.

Compilation Time

During compilation, the logical files can be thought of as falling into one of four categories. These categories are:

Residence File
Operation Files
External Files
Internal Files

The functions of the logical files and the devices to which they can be assigned are as follows.

Residence File

SYSTEM File. The SYSTEM file contains the System Control Program, the Fortran compiler, the Fortran loader, and the user's object-program library. It is assigned to a fixed area in a 1311 or 1301 disk unit, or to magnetic tape.

Operation Files

CONTROL File. The CONTROL file may be assigned to the card reader or to the console printer. When assigned to the card reader, the CONTROL file contains cards that send information to the System Control program or the Fortran loader. When assigned to the console printer, 80 character records must be used to send information to the System Control Program and 100 character records to send information to the Fortran loader.

MESSAGE File. The MESSAGE file contains information of primary interest to the machine operator. These messages are usually diagnostics relating to the operating procedures and/or instructions to the machine operator. It can be assigned to the printer, or to the console printer.

External Files

LIST File. The LIST file, generally associated with high-volume printed listings, contains information directed primarily toward the source programmer. It can be assigned to the printer, or to disk storage, or to magnetic tape, or it can be omitted. If the LIST file is assigned to a disk unit, the information is stored two sectors per printed line in the move mode.

INPUT File. The INPUT file contains source information to the compiler. It can be assigned to the card reader, or to any available area in disk storage, or to magnetic tape. If the file is assigned to a disk unit, the card images must be stored one card per sector in the move mode.

OUTPUT File. The OUTPUT file may contain the results of the operation specified in the RUN card. It can be assigned to the card punch, or to disk storage, or to magnetic tape, or it can be omitted. If the file is assigned to a disk unit, any card images will be stored one per sector in the move mode.

LIBRARY File. The LIBRARY file is a mass-storage file that supports the Fortran subprogram facility. The file contains standard Fortran functions and subroutines such as the sine and cosine functions. It is maintained by the Fortran librarian and used by the Fortran loader. The LIBRARY file can be assigned to any available area in disk storage, or to magnetic tape.

LOADER File. The LOADER file contains machine-language object programs in a relocatable format. It is built by the Fortran compiler, and used by the Fortran loader. The LOADER file can be assigned to any available area in disk storage, or to magnetic tape.

Internal Files

WORK1 and WORK2 Files. WORK1 and WORK2 are required files. They are used by the Fortran compiler for the GETEX and PUTEX functions that perform the large volume of data handling during compilation. They can be assigned to any available area in disk storage, or to magnetic tape.

WORK3 File. WORK3 is a required file. It can be assigned to any available area in disk storage, or to magnetic tape. The WORK3 file is used as an out-of-line file (PLACE) that bypasses data around major portions of the compiler.

WORK4, WORK5, and WORK6 Files. WORK4, WORK5, and WORK6 are not used by the Fortran compiler. They are defined for the user's input/output requirements during the execution of the object program produced by the compiler.

Execution Time

At execution time, the logical files may be thought of as falling into one of two categories. These categories are:

1. Files that are reserved for use by the Fortran system.
2. Files that are free to be used by the user's program.

The first category consists of four logical files. They are the SYSTEM, CONTROL, LIBRARY, and LOADER files. The function of these files is essentially the same during execution time as it was during compilation.

Fortran Numerical File Name	System Control Program File Name
0	MESSAGE
1	INPUT
2	OUTPUT
3	LIST
4	WORK1
5	WORK2
6	WORK3
7	WORK4
8	WORK5
9	WORK6

Figure 11. Correspondence between Fortran Numerical File Names and System Control Program File Names

The second category consists of the remaining ten logical files. The function of each of these files during execution time is determined by the user's program. These logical files have each been assigned a Fortran numerical file name. The correspondence between Fortran numerical file names and the System Control Program logical file names is shown in Figure 11.

The Fortran numerical files can be assigned to the same devices as the corresponding logical files. For example, the INPUT file can be assigned to the card reader, or to any available area in disk storage, or to magnetic tape. Likewise, numerical file 1 can be assigned to the card reader, or to any available area in disk storage, or to magnetic tape. Numerical files are assigned by using the corresponding logical file name.

Control Cards

The System Control Program recognizes eight types of control cards. They are:

RUN
INIT
ASGN
UPDAT
NOTE
PAUSE
COPY
HALT

Each type is punched in the Autocoder format. *Appendix I* contains a summary of all specific control cards that the System Control Program recognizes. Included in *Appendix I* is a detailed description of the manner of punching each specific control card and

valid entries for each of the general formats as discussed in the following sections. All control cards are printed on the MESSAGE file.

RUN Cards

The RUN card indicates the portion of the Fortran system that is to be selected by the System Control Program. A RUN card is required for four jobs to be performed. The four types of RUN cards are:

```
FORTRAN RUN
LOADER RUN
PRODUCTION RUN   three-character phase name
LIBRARY RUN
```

See *Preparing Jobs* for the specific RUN card format required for each job.

INIT Card

The INIT card is used to initialize the system. When the INIT card is sensed, the assumed logical file assignments become effective, and the LOADER file is initialized to accept a new batch of compiler output.

The general format of the INIT card is:

```
FORTRAN   INIT   [any message and/or comment]
```

ASGN Cards

An ASGN card indicates to the System Control Program that a logical file is to be assigned to a specific input/output device or area. An ASC card is used when the user wants a logical file assigned to an input/output device or area other than the assumed assignment of the System Control Program, or when the user wants to change an assignment that he has previously made.

The general format for an ASGN card is:

file-name ASGN { *device* / OMIT }

The *file-name* is the specific logical file; *device* is the input/output unit to which the logical file is to be assigned. Two examples for using an ASGN card follow.

The logical file, INPUT, is to be changed from the assumed device assignment (READER 1) of the System Control Program to an area in disk storage. This area is to be on 1311 unit 3, beginning at address 000600 and extending *to* (not through) 000900. Note that the END address to be punched is one more than the area actually used by the INPUT file. The ASGN card for this example is punched:

```
INPUT   ASGN   1311 UNIT 3, START 000600, END 000900
```

The second example is when a logical file is to be omitted. (This option is valid only in specific cases.) If the OUTPUT file is to be omitted, the ASGN card is punched:

```
OUTPUT   ASGN   OMIT
```

Blanks must be left between items in the operand field where indicated in the specific formats. For example, if the operand is READER 2, there must be a blank between READER and 2. Also, entries must be left-justified in the label, operation, and operand fields.

During a single stack of jobs, an assignment made by the user for a single logical file remains in effect until another ASGN card is sensed for that particular file, or until an INIT card is sensed, or until a HALT card is sensed. For example, an ASGN card that specifies the INPUT file to be assigned to READER 2 causes the assumed assignment, READER 1, to be altered. The System Control Program will select READER 2 during a single stack until another ASGN card for the INPUT file is encountered, or until an INIT card is sensed.

UPDAT Card

The UPDAT card is included in a package supplied by IBM, or supplied by the user for the purpose of amending the user's Fortran system. UPDAT cards supplied by IBM are prepunched in the following format:

{ *processor-name* / SYSTEM } UPDAT *phase-name,* { DELETE / INSERT }

This card (excluding DELETE) will be followed by the appropriate data cards.

NOTE Card

The NOTE card contains messages and/or instructions from the programmer to the machine operator. There is no interruption of processing when this control card is sensed by the System Control Program. The content of the NOTE card is printed on the MESSAGE file. The general format of the NOTE card is:

```
NOTE   any message and/or instruction
```

One application of the use of a NOTE card might be in the case where the programmer wants the program listing to include the date on which a particular job was compiled. The message that could be used for this purpose is:

```
NOTE   JOB NUMBER FOUR COMPILED 3/2/66
```

PAUSE Card

The PAUSE card contains messages and/or instructions from the programmer to the machine operator. When the PAUSE card is sensed, the content of the PAUSE card is printed on the MESSAGE file. Then, the System Con-

trol Program temporarily halts the system. Processing is resumed by pressing the start key. The general format for the PAUSE card is:

PAUSE *any message and/or instruction*

One application of the use of a PAUSE card might be in the case where the INPUT file for a job is located on disk unit 2. The programmer can inform the machine operator of this fact by using a PAUSE card, telling him to ready the drive. The message would be:

PAUSE READY THE PACK ON DISK DRIVE 2.

COPY Card

The COPY card is applicable only to tape-resident systems. It is used when the user wants to duplicate the system tape. The COPY option permits the user to duplicate the SYSTEM file, including the LIBRARY file if it resides on the same tape, on another tape (WORK1). The general format for the COPY card is:

COPY [*any message and/or identification*]

HALT Card

The HALT card indicates to the System Control Program that processing has been completed. It is the last card of a stack. The content of the HALT card is printed on the MESSAGE file. The general format for the HALT card is:

HALT [*any message and/or identification*]

Fortran Processor Program

The Fortran Processor Program is made up of the following:

Fortran compiler
Fortran loader
Fortran library

The following sections contain a description of the various components of the Fortran Processor Program, a description of the output from the components, and a description of any diagnostic messages that the user may receive as a result of processing operations.

Fortran Compiler

The Fortran compiler is the processing element of the processor program. It operates under control of the System Control Program. The compiler translates source program statements written in the Fortran language into machine-language and interpretative instructions in a relocatable format. These instructions are then acceptable to the Fortran loader.

Instructions in the relocatable format contain symbolic and relative addresses. These addresses are relative to a base address of 00. These symbolic and relative addresses are converted to actual addresses by the Fortran loader. The result of this conversion is an object program in the absolute format.

Relocatable formats permit inclusion of several relocatable programs at load time. Inter-program communication is accomplished by the use of symbolic names, whose corresponding addresses are substituted at load time by the Fortran loader.

Compiling Variables

Five compiling variables are available in the compiler. These variables include:

1. Integer size (the number of significant digits) to be used at object time.
2. Real size to be used at object time.
3. Object machine size.
4. Availability of the multiply/divide feature.
5. Main program name.

These variables can be specified by using compiler option control cards. A description of each of these variables follow.

Integer Size. The assumed integer size is 5. Preceding a compilation, the object-time integer size, k, can be specified to be any value from 1 through 20.

Real Size. The assumed real size is 8. Preceding a compilation, the object-time real size, f, can be specified to be any value from 2 through 20, where f is the mantissa length. The compiler will reserve $f + 2$ positions for each real variable to allow for a 2-digit exponent.

Object Machine Size. The assumed object machine size is 11999. If the object machine is greater than the assumed value, the highest core storage address available at object-time must be specified if maximum core storage usage is to be attained.

Multiply/Divide Feature. The multiply/divide feature is assumed to be available in the object machine. If the feature is not available, this fact can be specified by using a compiler control card.

Main Program Name. The main program (phase) name is assumed to be ///. By using a compiler control card, the user can specify a main program (phase) name that is three alphameric characters in length. At least one character of the three-alphameric character name must be alphabetic. The name appears on the first card, disk, or tape record of the

relocatable output generated by the compiler. If an absolute deck is specified to the loader, the three characters are included in the first card of the absolute deck. These three characters are used to identify the program if it is stored as a phase on the SYSTEM file.

If the program (phase) is to be stored on the SYSTEM file as a phase that can be called for execution at any time, the user *must* make sure that the three characters of the program (phase) name are not the same as the name of one of the phases of the System Control Program and/or the Fortran processor. The names of phases of the Fortran processor are in the form *nn*F, where *n* is numeric. The names of the phases of the System Control Program are in the form *xxy*, where *x* is alphabetic and *y* is alphameric. *Appendix II* contains the three-character names of the phases of both the System Control Program and the Fortran Processor Program. Consult the appendix to ascertain that there is no duplication of phase names.

Fortran Compiler Output

The output from the Fortran compiler is on the devices as specified in the ASGN cards. The LIST file output, with an assumed assignment to be the printer, is composed of a source program listing, a name dictionary, and a sequence number dictionary. Any or all of these types of output can be omitted by using an output option control card.

Source Program Listing. A listing of the Fortran source program is output by the Fortran compiler. The listing is made up of input card images and a compiler-generated sequence number. A sequence number appears in front of each card except for a comment card. An example of a source program listing is shown in the sample program included as *Appendix IV.*

Name Dictionary. A name dictionary is made up of the names of simple variables and/or arrays that are included in the source program. Associated with each variable and/or array is the corresponding object-time relocatable address. These addresses are relative to a base address of 001. An array-name address corresponds to the first element of the array. An example of a name dictionary is shown in the sample program included as *Appendix IV.*

Sequence Number Dictionary. The sequence number dictionary is made up of the compiler generated sequence numbers and the corresponding object-time relocatable addresses. These addresses are relative to a base address of 001. The address indicates the position of the first character of the transformed statement.

Sequence numbers do not necessarily appear in order in the dictionary. Further, a sequence number may appear twice in the dictionary. (This could occur in the case of a READ/WRITE statement. A sequence number would represent the actual input/output subroutine call; the same sequence number would represent the transformed I/O list.) Specification statement sequence number addresses appear in the dictionary with the addresses equivalent to another sequence number address.

Relocatable Punched Card Deck

In addition to output on the LIST file, the user can specify, by way of an OUTPUT ASGN card, that a punched-card deck in the relocatable format be produced by the compiler. This card deck is the same object program that is located on the LOADER file in mass storage. When a punched-card deck is specified, the user has the option of specifying that the relocatable object program on the LOADER file be omitted. (This is accomplished by using an ASGN OMIT card.) Although it is possible to have the object program in the relocatable format on both the LOADER file and the OUTPUT file, generally only one form of output is chosen.

Fortran Compiler Diagnostics

Diagnostic information is produced pertaining to the intelligibility and consistency of the source program as defined by the language specifications section of this publication. If an error is detected by the compiler, a message is printed on the LIST file informing the user of his error. The message printed on the LIST file can have a maximum of three parts.

The first part of the message is a flag that indicates the severity of the error. A single asterisk (*) indicates a diagnostic of a warning type. A single asterisk allows compilation to continue and relocatable output is produced.

Three asterisks (***) indicate a severe error, and compilation is suspended. When this type of error occurs, the message

*** COMPILATION SUSPENDED ***

is printed on the LIST file. In these instances, control is returned to the System Control Program, and a control card for the next job is read.

The second part of the message is the sequence number of the statement in error. In certain cases, the sequence number is not included as a part of the message. For example, if the name table is being searched and an error occurs, no sequence number is included in the message.

```
                          FORTRAN   RUN
      C     SAMPLE PROGRAM TO DEMONSTRATE TYPES OF COMPILER DIAGNOSTICS.      SAMPLE
001           DIMENSION A(5)                                                  SAMPLE
002           I=1                                                             SAMPLE
003         1 READ (5,2) A(I)                                                 SAMPLE
004         2 FORMAT (10X,2F10.3)                                             SAMPLE
005           WRITE (3,3) I                                                   SAMPLE
006         3 FORMAT (1X,14HRUN NUMBER IS I3,//17HOBEGIN PROCESSING ////)     SAMPLE
007           B(I)=SQRT(A(I))                                                 SAMPLE
      C     ADDITIONAL STATEMENTS FOR PROCESSING.                             SAMPLE
008           PAUSE                                                           SAMPLE
009           GO TO 1                                                         SAMPLE
010           END                                                             SAMPLE


NAME DICTIONARY
  00056 A       00061 I


                                  *** DIAGNOSTICS ***

FLAG      SEQ                          MESSAGE
***       007  ARITHMETIC STATEMENT,ARRAY NOT DIMENSIONED  (B)
*         006  FORMAT STATEMENT,IMPROPER ELEMENT SEQUENCING  (,)

                        ***COMPILATION SUSPENDED***
```

Figure 12. Erroneous Coding and Resulting Diagnostics

The third part of the message is the diagnostic itself. The diagnostic can include the type of statement in error and in some cases the portion of the statement being processed when the error occurred. The portion of the statement being processed is enclosed within parentheses.

Figure 12 is an example of erroneous coding and the diagnostic messages that result. The first error is the three-asterisks type that causes compilation to be suspended. In sequence number 007, array B, (enclosed within parentheses in the last part of the diagnostic), was not dimensioned in the source program.

The second error is the one-asterisk type that allows compilation to proceed. These are merely warnings to the programmer. In sequence number 006, a comma, (enclosed within parentheses in the last part of the diagnostic), incorrectly appeared in a FORMAT statement. In statement 006, the second comma must precede I3 rather than follow I3. Further, a comma should not separate the I3 and the / in a FORMAT specification.

An additional diagnostic message can appear in the sequence number dictionary. The diagnostic message

UNDEF STMT NO

will appear in the sequence number dictionary following the sequence number whose associated source statement contains a reference to an undefined statement number.

If an error is made when punching a compiler option control card, the content of the card is output on the LIST file beginning in position one, and the message

BAD CONTROL CARD

appears in positions 61-80.

Fortran Loader

The Fortran loader operates on the relocatable object programs that are produced by the Fortran compiler. It operates under control of the System Control Program. The function of the Fortran loader is to load object programs in the relocatable format into core storage. It relocates addresses and provides linkage among the programs, when appropriate. Optionally, the loader produces the relocated object program on the OUTPUT file in absolute format, or on the LIST file as a storage print, or starts the execution of the program.

Under normal operations, the relocatable object programs are present on the LOADER file in mass storage (disk or tape). Therefore, during a LOADER RUN, the LOADER file is always referenced by the Fortran loader.

When the LOADER file is assigned to mass storage, the relocatable programs are read into core storage when

a control card designating execution or no execution is sensed. At this point, the *entire* content of the LOADER file is read into core storage. Subprograms from the LIBRARY file are then included, if appropriate.

If the user has specified a punched-card deck as a result of the Fortran compiler processing, the LOADER file must then be assigned by the user to the card reader to which the CONTROL file is assigned. (All card input for the Fortran loader must be from the same device.)

When the LOADER file is assigned to a card reader, the relocatable programs must be read into core storage *before* the execution or no execution card is sensed. (The execution or no execution card is the last control card of the input for loader processing.) For this reason, an additional card, an INCLUDE card, is required preceding the relocatable programs to signal the Fortran loader that additional programs from the LOADER file (card reader) need not be included when the last control card is encountered. Instead, inclusion of required library routines takes place.

If additional programs are to be present during loader processing, the user can specify the location of these programs in mass storage by way of ASGN cards. These areas would be referred to as the logical files INPUT and WORK files. The user would have to supply an INCLUDE card for each of the logical files that is assigned. With the INCLUDE card, the user could specify the entire file be read into core storage along with the main program from the LOADER file. Further, if the entire file is not required, the user could specify by name a specific program to be read into core storage along with the main program. Subprograms from the LIBRARY file are then included.

If more than one set of relocatable programs is on the LOADER file, i.e., the programs are batched, the user normally would want to process all the programs at one time. If this is the case, the loader relocates and loads all programs encountered until the end of the file is sensed. If the file is a disk unit, a record with 1EOFb as the first five characters signifies the end of the file. A tape mark signifies the end of the file for tape files. The last card in a deck signifies the end of the file for card files. When batching is performed by the compiler on the LOADER file, an end-of-file is always defined.

When a program is to be executed after loader processing, the loader prepares core storage for execution. The loader overlays itself with a standard overlay package. This overlay package consists of an arithmetic interpreter and various input/output routines. The user's program is then executed.

If a program is not to be executed after loader processing, control returns to the System Control Program, which reads the next card in the CONTROL file.

Fortran Loader Output

The Fortran loader can produce three types of output. Unless specified to the contrary by an output option control card, a name map is output on the LIST file. By way of an output option control card, the user can specify that an absolute deck be produced on the OUTPUT file and a storage print be produced on the LIST file.

Name Map. Unless the user specifies otherwise, a name map is produced on the LIST file. The name map is a table that includes all external names assigned during the loading process. Each name is associated with its absolute address. The user can then determine all entry points for the subprograms within the object program.

Absolute Deck. By using an output option control card, the user can specify that an absolute deck be produced on the OUTPUT file. The absolute deck is made up of the entire relocated program, including the loader overlay package for the processor machine being used for the LOADER RUN. This deck can then be stored as a phase on the SYSTEM file for subsequent runs on the same processor machine used during the LOADER RUN. When this is done, the phase (absolute deck) can be selected from the SYSTEM file by using a three-character phase name. The three-character phase name is the three-character main program name specified to the compiler. The three-character phase name appears in columns 78-80 of each card of the absolute deck. It also appears in columns 21-23 of the first card of the absolute deck, which is an UPDAT INSERT card. The deck is sequenced in columns 73-75, beginning with 001.

Storage Print. By using an output option control card, the user can specify that a storage print be produced on the LIST file. The storage print shows the absolute locations of the programs that were loaded and relocated. The storage print does not include the overlay package.

Fortran Loader Diagnostics

Incorporated within the Fortran loader is the capability for recognizing error conditions that may occur during the loading process. These error conditions may prevent the successful completion of the user's program. The conditions are diagnosed by the loader, and an error indicator is output on the LIST file and a halt occurs. The error indicator, ERROR *n*, is in character positions 14 through 20.

Figure 13 shows the error conditions that may arise, and the action that is taken if the system is restarted.

Message	Meaning	Action if Restarted by Pressing START on the Console
ERROR 0	Two main programs have been encountered.	A storage print and a name map are output on the LIST file. Execution is suppressed, and control returns to the System Control Program, which reads the next card in the CONTROL file. Adjustment of the INPUT file may be necessary before pressing START.
ERROR 1	COMMON area of the program being loaded does not match a previously encountered COMMON area.	
ERROR 2	Integer size and/or real size is inconsistent.	
ERROR 3	Core storage is exceeded.	
ERROR 4	Loader control card was not recognized.	
ERROR 5	The subprogram whose name appears on the LIST file along with ERROR 5 was not found in the library.	
ERROR 6	The name that appears on the LIST file along with ERROR 6 was multiply defined.	
ERROR 7	Main program not included.	
ERROR 8	Name table overflow.	
ERROR 9	Library header record not present where the library is assigned.	

● Figure 13. Fortran Loader Diagnostic Messages

Object Time Diagnostics

Conditions may arise during the execution of the object program that the system recognizes as being erroneous. A message is printed unconditionally on the printer, starting at print position one. Messages are in the form

XXX YYY

where xxx is the error code for arithmetic errors and input/output routines, and YYY is the address of the location following the branch to the read/write subroutine or the interpreter. The remainder of the printed-line contains the current contents of a work area. The system does not halt when these errors occur.

Codes (xxx) for arithmetic errors, the meaning of the codes, and the values that are used by the object program follow.

XXX *Code*	*Meaning*	*Value used in Object Program*
NOF	Exponent overflow during normalization	±.99...E99
DZE	Attempt to divide by zero	±.99...E99
EOF	Exponential greater than 10^{99}	±.99...E99
LNZ	Logarithm of zero	±.99...E99
SCL	Sine or cosine argument too large	zero
SSE	Subscript error—index greater than 15999	A hard halt with 2002 in the A-address register.
LNN	Logarithm of negative number	ln | arg |
ZTZ	Zero raised to zero power	one
SQN	Square root of negative arguement	$\sqrt{\lvert \text{arg} \rvert}$

Codes (xxx) for input/output routine errors, the meaning of the codes, and the action that is taken in the object program as a result of the error follow.

XXX *Code*	*Meaning*	*Result*
E01	READ/WRITE statement with no list or FORMAT statement.	The next statement is processed.
E02	List exceeds the logical record on an unedited READ.	The next statement is processed.
E03	Variable type and conversion specification do not match.	The next statement is processed.
E04	Edited record is too long.	The next statement is processed.
E05	End of format detected twice with the same list element.	The next statement is processed.
E06	Edited record was exceeded while processing H-conversion field.	The next statement is processed.

xxx *Code*	*Meaning*	*Result*
E07	Unrecognizable character in E-, F-, or I-conversion input.	Assumed value is zero.
E08	Too many signs in E-, F-, or I-conversion input.	Assumed value is zero.
E09	Too many decimal points in E- or F-conversion input.	Assumed value is zero.
E10	Exponent is more than two positions in E-conversion input.	Rightmost position(s) are lost.
E11	No exponent after "E" in E-conversion input.	Assumed value is zero.
E12	Exponent specified in F-conversion input.	Treated as E-conversion input.
E13	Exponent overflow in E- or F-conversion input.	Assumed value is zero.
E14	Decimal point or exponent with I-conversion input.	Assumed value is zero.
E15	Input data field is longer than variable.	Assumed value is zero.
E16	L input, and first nonblank character not "T" or "F".	Assumed value is .FALSE. .
E17	BACKSPACE references file not previously referenced.	The next statement is processed.
E18	END FILE references device other than a mass-storage device.	The next statement is processed.
E19	FIND references disk unit with inoperative access mechanism.	Press START on the console to retry the disk I/O operation.
E20	Illegal variable file specification.	The next statement is processed.
E21	Illegal characters in object-time FORMAT.	The next statement is processed.
E22	Disk area exceeded, or end-of-file was sensed during WRITE operation.	The next statement is processed.
E23	Parenthesis mismatch in object-time FORMAT.	The next statement is processed.
E24	H-conversion field extended beyond the end of the record in object-time FORMAT.	The next statement is processed.
EOJ	A 7-8 punch in column 1 was detected during a READ operation.	Control is returned to the System Control Program, which reads the next control card.

Fortran Library

The Fortran library is made up of standard Fortran functions and subroutines. Collectively, the functions and subroutines are referred to as subprograms. These subprograms are included in the user's program at load time, just before execution.

The Fortran library is built and maintained by the librarian. The expansion capability of the system permits the user to code subroutines and place them in the library. The method for adding (or deleting) subroutines from the library is described in *Preparing Library Jobs.*

The standard subprograms supplied by IBM are defined in the language specifications section of this publication.

Object Programs

This section contains a description of the way object programs are contained in core storage at the time they are executed. Topics discussed in this section include storage allocation, the standard loader overlay, subprograms, statement expansions, and an explanation of the core-storage allocation for the sample program contained as *Appendix IV.*

Storage Allocation

In general, a Fortran source program is translated into interpretive strings representing arithmetic expressions or input/output lists, sequences of tests for logical expressions, and subroutine calls with necessary parameters.

Each Fortran main program or subprogram is relative to a relocatable base address of 001, and is ordered in the following manner.

1. Constants common to every program and subprogram
2. Arrays
3. Simple variables and constants
4. Executable statement expansions
5. FORMAT statement and input/output list expansions
6. Subprogram prologue and epilogue.

Each real variable requires $f + 2$ positions of core storage. Each integer variable requires k positions of core storage. Each logical variable requires one position of core storage.

The number of core-storage positions required for an array is determined by multiplying the array dimensions together and multiplying that product by the appropriate word size according to type (real, integer, or logical).

Integer constants appear in core storage with a length as written, but leading zeroes are dropped. A maximum of k positions is permitted; however, fewer may be used. Real constants always require a length of $f + 2$ positions in core storage. Logical constants require one position of core storage.

Arrays and simple variables in COMMON are assigned absolute (i.e., non-relocatable) addresses beginning with the high core-storage address and proceeding with decreasing addresses. No relocatable load cards are generated.

The Fortran loader normally loads a main program and possibly several subprograms in the relocatable format from the LOADER file. The first program read from the LOADER file is loaded into core storage beginning at address 5701. Subsequent loading follows the preceding program. Any required standard subprogram will then be relocated and loaded following the user-programs. A communication and system-constant area is prepared from core-storage address 950 to 1010.

When loading is complete, the loader prepares for object-program execution by calling a set of standard subprograms, called the standard loader overlay, always required by the object program. Execution then takes place.

Standard Loader Overlay

The standard-loader-overlay package occupies core-storage positions 1010 through 5700 during object-program execution. The overlay package includes routines such as the arithmetic interpreter, object-time error messages, and the general READ/WRITE service routines for initialization, file opening, end-of-file detection, buffer clearing, and unit record (card reader, card punch, printer, and console printer) input/output processing. Also included are the input/output list routine, and several formatting routines to handle FORMAT initialization, FORMAT-list interaction, the slash (record delimiter) element, and I, E, and F input and output conversions. Additional selectively included subroutines are normally required when using a FORMAT statement, as described with character counts in the following section.

Selectively Included Standard Subprograms

A list of standard subprograms that exist on the LIBRARY file in the relocatable format follows. Each is relocated and loaded by the loader, if required by the user-program(s). Note that some standard subprograms require other standard subprograms. The use of certain capabilities of the Fortran language as well as explicit subprogram references require the inclusion of standard subprograms. The external names that are associated with the subprograms appear in the name map when the corresponding subprogram is loaded.

Subprogram	*External Name*	*Character Count*
FORMAT left parenthesis	G.	143
FORMAT internal right parentheses	H.	39
FORMAT scale factor	L.	18
FORMAT H-specification	M.	153
FORMAT L-specification, requires P.	N.	135
FORMAT A-specification, requires P.	O.	191
FORMAT A-, L-, I-, F-, or E-specification	P.	230
FORMAT final right parenthesis	Q.	98
Random file processing for FIND or random READ/WRITE (disk-resident system only)	A. B. C. D. E. F.	217
Each logical file control word for file i, where $0 \leq i \leq 9$	*i	21
Unedited READ/WRITE	I. J. K.	373 (disk) 369 (disk)
BACKSPACE	R.	241 (disk) 171 (tape)
END FILE	S.	153 (disk) 90 (tape)
REWIND	T.	78 (disk) 47 (tape)
FIND (disk-resident system only)	U.	181
Variable file name search	V.	106
Address vector for random file definition vectors, requires V.	X.	33
Address vector for sequential file control words, requires V.	Y.	33
Object-time FORMAT, requires G., H., L., M., N., O., P., and Q.	W.	1061
Tape I/O with disk-resident system	Z.	171
Subscripting	)A)B U) V) Y) Z)	283
Multiply/divide subroutine	)C)D)E	471

Subprogram	External Name	Character Count
DO, or implied DO in I/O list	)F)G)H	173
Sense light	SSLITE W) X)	59
Sense light test, requires SSLITE	SLITET	82
Sense switch test	SSWTCH	83
End-of-file test	EOF	44 (disk) 44 (tape)
Phase linkage (also required by STOP statement)	LINK	22 (disk) 22 (tape)

Other uses of the Fortran language, as well as explicit references to standard Fortran functions, cause the loading of an 8-character BCE instruction for each function, plus the coding for the function itself. The external name associated with the BCE instruction is of the form ,*c* , where *c* represents the alphameric character unique to each standard function. The complete set of required BCE instructions for a particular program exists together in core storage after the program(s) have been loaded by the Fortran loader. The external name ,9 appears at the beginning of the BCE instructions.

Subprogram	External Name	Character Count
QUIT — leave interpretive coding for executable coding	,0	8
Relational expression testing, including arithmetic IF	,0	8
	,1	8
	,2	8
	,3	8
	,4	8
	,5	8
	(0 (1 (2 (3 (4 (5	134
Absolute value — ABS	,A	8
— IABS	,B	8
	(A	11
Positive difference — DIM	,D	8
— IDIM	,V	8
	(D	64
Extreme value — AMAX0, requires ,R and (R	,E	8
— AMAX1	,F	8
— MAX0	,G	8
— MAX1, requires ,I and (I	,H	8
— AMIN0, requires ,R and (R	,J	8
— AMIN1	,K	8
— MIN1, requires ,I and (I	,O	8
— MIN0	,Y	8
	(E (F (G (H (J (K (O (Y	248
Truncation — IFIX	,I	8
	(I	129
Float — FLOAT	,R	8
	(R	67
Remaindering — MOD	,M	8
	(M	88
— AMOD, requires ,R and (R and ,I and (I	,P	8
	(P	208
Transfer of sign — ISIGN	,U	8
— SIGN	,W	8
	(W	69
Truncation to real — AINT, requires (I and (R	,Z	8
	(Z	99
Complement compression, required by a subprogram with adjustable dimensions — CMCM, requires subscripting	,8	8
	(8	18
Integer exponentiation	,7	8
	(7	319
Power — logarithmic part of real exponentiation, requires A) and (L	,6	8
Cosine — COS, requires A) and (S	,C	8
Sine — SIN, requires A)	,S	8
	(C (S	441
Common logarithm — ALOG10, requires A)	,L	8
	(L	370
Natural logarithm — ALOG, requires A) and (L	,N	8

Subprogram	*External Name*	*Character Count*
Square root — SQRT, requires A)	,Q	8
	(Q	228
Arctangent — ATAN, requires A)	,T	8
	(T	503
Exponential — EXP, requires A)	,X	8
	(X	262
Power series subroutine required by various transcendental functions	A)	291
	B)	
	C)	
	D)	
	E)	
	F)	
	G)	
	H)	
	I)	
	J)	
	K)	
	L)	
	M)	
	N)	
	O)	
	P)	

Statement Expansions

The COMMON, EQUIVALENCE, and type statements do not generate object-time characters. These statements serve only as information to the compiler.

The DATA statement information literals, converted to internal notation, appear in the space allocated for their respective variables or array elements. Either $f + 2$, k, or one position is used.

An arithmetic statement defining a numerical value is translated into a character string consisting primarily of one-character operators and three-character (address) operands. References to standard functions require one character of core storage, plus the inclusion of a BCE for the function and the function itself. Subscripted variable references normally require five characters plus six for each dimension and the inclusion of the subscript subroutine.

An arithmetic statement defining a logical value is normally translated into a sequence of eight character tests on the various logical variables. Arithmetic strings, as described in the preceding paragraph, followed by a four-character relational test (one-character function reference plus a three-character parameter) represent a relation.

A GO TO statement results in a four-character branch instruction. A computed GO TO results in as many eight-character branch instructions as there are statement numbers in the computed GO TO.

A logical or arithmetic IF results in very nearly the same object expansion as an arithmetic statement, and can be approximated in essentially the same fashion.

The DO statement generates 23 characters, and causes the inclusion of the DO subroutine. Four characters are required after the final executable statement in the range of the DO.

The CONTINUE statement generates no object characters.

The PAUSE statement generates either one character or five characters, depending upon the absence or presence of the optional integer constant.

A STOP statement generates ten object-time characters and causes the inclusion of the LINK subroutine.

The END statement generates ten object-time characters.

Input/output lists are translated into a sequence of subroutine calls and interpretive strings. The sample program shown in *Appendix IV* gives an example of input/output lists.

A FORMAT statement is translated into a sequence of subroutine calls with parameters corresponding to the FORMAT elements used. Additional selectively included standard subprograms are normally required.

A READ or WRITE statement, or one of the manipulative input/output statements is translated to a subroutine call with required parameters. Corresponding lengths are:

READ/WRITE	19
FIND	10
END FILE	7
REWIND	7
BACKSPACE	7

A standard subprogram is required for each type of manipulative input/output statement used. An expression consisting of something more complex than a single non-subscripted variable or constant in a FIND statement, or in the random form of a READ or WRITE statement, causes the generation of additional characters, similar to the arithmetic statement. If a variable file name is used, ten additional characters are required for each reference, as well as two or three standard subprograms.

The DEFINE FILE statement causes the generation of seventeen characters for each file defined. The external names $\$i$, where $0 \leq i \leq 9$, are also generated by the DEFINE FILE statement.

The FUNCTION and SUBROUTINE statements cause generation of a prologue for the evaluation of parameters and array dimension calculations, if required, and an epilogue to reset any values required and return control to the proper place.

The RETURN statement generates four characters.

The CALL statement generates four characters, plus three characters for each actual argument. If the actual arguments consist of more complicated expressions than a single non-subscripted variable, constant, or alphameric field, then an expression-evaluation string for either arithmetic or logical expressions is also generated.

It was found that twenty-three typical programs had statement expansions averaging approximately thirty characters per statement. A program containing numerous long FORMAT statements had a significantly higher average, whereas a program almost entirely composed of simple arithmetic statements had a somewhat lower average. In all cases, additional storage was required for arrays, variables and constants, and required subprograms. Thus, the total core-storage requirement above the base loading point of 5700 varied considerably, ranging from about 500 characters to about 10,000 characters.

Core-Storage Allocation for the Sample Program

The discussion in this section refers directly to the sample program that is shown as *Appendix IV*. Core storage has been allocated for the object program in the following manner.

Area	*Content*
0-949	Resident System Control Program functions
950-1009	Communication area, system constants
1010-5700	Standard loader overlay
5701-6205	Main program, relocated
5701-5850	Constants, arrays, variables
5851-5991	Executable and END statements
5992-6205	FORMAT's, input/output lists
6206-8045	Selectively included standard subprograms
6206-6227	LINK subroutine, required by STOP statement
6228-6510	Subscripting subroutine
6511-6981	Multiply/divide subroutine
6982-7154	DO subroutine
7155-7206	Function branches for relational functions, required by logical IF statement
7207-7340	Relational expression testing
7341-7483	FORMAT left parenthesis
7484-7522	FORMAT internal right parenthesis
7523-7675	FORMAT H-specification
7676-7905	FORMAT A-, L-, I-, F-, or E-specification
7906-8003	FORMAT final right parenthesis
8004-8024	File control word for logical file 1
8025-8045	File control word for logical file 3

This accounts for all external names appearing in the name map, recognizing that the main program name is the assumed value ///.

The statement expansion area occupies 355 characters, averaging approximately 35 characters per statement. This average includes the END statement, but excludes the DIMENSION statement. Note that the FORMAT expansion required considerably more than the average number of characters, whereas the arithmetic statement, sequence number 004, required less.

The location and length of each statement expansion can be determined by adding the program base loading point to the relative addresses shown in the sequence number dictionary. The program base loading point for the first program that is loaded is 5700. Subsequent programs are loaded immediately following the first program. Consequently, their base loading points are higher than 5700.

Jobs

The Fortran system performs two major operations.

1. Translates source programs into object programs.
2. Starts the execution of object programs.

Because these operations are performed by the Fortran processor part of the system, the operations are called processor jobs.

Two other operations, maintaining the Fortran subprogram library and updating the Fortran system, are also considered jobs. Maintaining the Fortran subprogram library is called a library job. Updating the Fortran system is called an update job. Update jobs are described in *Updating a Fortran System.*

Under control of the System Control Program, it is possible to perform one or more jobs without operator intervention. This process is called stack processing. If the system resides on disk, or if the system resides on tape unit 2, 3, 4, 5, or 6, a stack is *always* made up of the *card-boot* deck, a SYSTEM ASGN card, the particular job(s) to be performed, and a HALT card. If the system resides on tape unit 1, a stack is made up of the particular job(s) to be performed and a HALT card. (Pressing the tape load key serves the same function as the *card boot* and the SYSTEM ASGN card when the system tape is on unit 1.)

In performing a job, the following factors must be taken into consideration.

1. The kind of input for the job.
2. The use of the logical files.
3. The machine-operator procedures to be followed.

The kinds of input for processor jobs and library jobs are discussed in the following sections (*Preparing Processor Jobs* and *Preparing Library Jobs*).

The general use of logical files is discussed in *Logical Files.*

In most cases, the user does not need to be concerned about the logical files because the Fortran system defines the files and assigns them to specific input/output devices. In the description of preparing processor jobs that follows, any file assignment that the user must make is explained.

The machine-operator procedures to be followed are described in *Performing Jobs.*

Preparing Processor Jobs

This section describes each processor job. They are:

FORTRAN RUN
LOADER RUN
PRODUCTION RUN

Each processor job description includes:

1. *Assumed input device.* This entry refers to the device on which the input is assumed to be located. For the 1402, READER 1 means that the cards are selected into stacker 1. For the 1442, READER 1 means unit 1.
2. *Input.* This entry refers to the type of input for the job.
3. *Assumed output devices.* This entry refers to the device(s) on which the output is assumed to be located. For the 1403, PRINTER 2 means that 132 print positions are available. For the 1443, PRINTER 2 means that 144 print positions are available. For the 1402, PUNCH 4 means that the cards are selected into stacker 4. For the 1442, PUNCH 1 means unit 1.
4. *Output.* This entry refers to the type of output that the user *always* gets as a result of the job, unless specified otherwise.
5. *Output options available.* This entry refers to the type of output the user can get by using output option cards.
6. *Required user assignments.* This entry describes any additional logical file assignments that the user must make to perform the job.
7. *Control cards.* This entry describes the method of punching any required control cards and output option cards.
8. *Arrangement.* This entry references a figure that shows the manner of arranging card input for the job.

NOTES.
1. Any logical file assumed assignment can be changed. If the user wishes, he can change the assignment by using an ASGN card.
2. NOTE and PAUSE cards can be placed between, but not within, job decks.

FORTRAN RUN

This is the type of run that translates a source program written in the Fortran language into an object program in the relocatable format. The output for this run is then ready for processing by the Fortran loader.

Assumed Input Device. INPUT file on READER 1.

Input. Source program.

Assumed Output Devices. MESSAGE file on PRINTER 2; LIST file on PRINTER 2; LOADER file on 1311 or 1301 UNIT 0, START 010400, END 012000 or TAPE UNIT 3.

Output.

1. Source-statement diagnostics on the LIST file, if errors are sensed.

2. Source program listing on the LIST file, unless an output option control card specifies that the listing be omitted.

3. Name dictionary on the LIST file, unless an output option control card specifies that the name dictionary be omitted.

4. Sequence number dictionary on the LIST file, unless an output option control card specifies that the sequence number dictionary be omitted.

5. Object program in the relocatable format on the LOADER file, unless the LOADER file has been omitted.

Output Options Available. Object program in the relocatable format on the OUTPUT file.

Required User Assignments. If the object program is to be written on the LOADER file, no file assignments are required. If the user wants the object program on the OUTPUT file, an OUTPUT ASGN card is required.

Control Cards.

1. The RUN card is the only required control card. Punch the RUN card in the following manner.

Columns	*Contents*
6-12	FORTRAN
16-18	RUN

2. If the object program is to be on the OUTPUT file, an OUTPUT ASGN card is required. This card must precede the RUN card. Generally, the only time that the user would want to use the OUTPUT file is when a punched card deck in the relocatable format is desired. If this is the case, punch the OUTPUT ASGN card in the following manner:

Columns	*Contents*
6-11	OUTPUT
16-19	ASGN
21-27	PUNCH *n*

If the OUTPUT file is assigned to PUNCH *n*, *n* can be 0, 4, or 8 for 1402, 1 or 2 for 1442, 3 for 1444.

If the OUTPUT file is assigned, the user may wish to omit the LOADER file. If this be the case, a LOADER ASGN card is required and must precede the RUN card. Punch the LOADER ASGN card in the following manner.

Columns	*Contents*
6-11	LOADER
16-19	ASGN
21-24	OMIT

3. The following cards are output option control cards and compiler option control cards. They are used when any of the options are desired. The option cards immediately precede the Fortran source program in the INPUT file and can be in any order. Option control cards are output on the LIST file.

a. If the integer size is to differ from the assumed value of 5, punch the following card. *Note:* $01 \leq nn \leq 20$.

Columns	*Contents*
1-8	$INTEGER
10-13	SIZE
15	=
17-18	*nn*

b. If the real size is to differ from the assumed value of 8, punch the following card. *Note:* $02 \leq nn \leq 20$.

Columns	*Contents*
1-5	$REAL
7-10	SIZE
12	=
14-15	*nn*

c. If the object machine differs from the assumed value of 07999, punch the following card.

Columns	*Contents*
1-7	$OBJECT
9-15	MACHINE
17-20	SIZE
22	=
24-28	15999

d. If the multiply/divide feature is not present in the object machine, punch the following card. (The multiply/divide feature is assumed by the Fortran compiler to be available.)

Columns	*Contents*
1-3	$NO
5-12	MULTIPLY
14-19	DIVIDE

e. If the main program (phase)name differs from ///, punch the following card. Note: *name* is three alphameric characters in length. At least one character of the three alphameric character name must be alphabetic. If the program is to be stored as a phase on the SYSTEM file, the arrangement of the three characters must be unique, i.e., the arrangement must differ from:

1. Any phase name of the System Control Program
2. Any phase name of the Fortran Processor Program
3. Any phase name that the user previously may have added to the SYSTEM file.

Columns	*Contents*
1-6	$PHASE
8-11	NAME
13	=
15-17	*name*

f. If no source program listing is desired, punch the following card.

Columns	*Contents*
1-3	$NO
5-8	LIST

g. If no name dictionary listing is desired, punch the following card.

Columns	*Contents*
1-3	$NO
5-8	NAME
10-19	DICTIONARY

h. If no sequence number dictionary is desired, punch the following card.

Columns	*Contents*
1-3	$NO
5-12	SEQUENCE
14-19	NUMBER
21-30	DICTIONARY

i. If neither the name dictionary nor the sequence number dictionary is desired, punch the following card.

Columns	*Contents*
1-3	$NO
5-14	DICTIONARY

Arrangement. The arrangement of input cards for a FORTRAN RUN is shown in Figures 14 and 15. The output option cards must be in the INPUT file and can be in any order.

LOADER RUN

This is the type of run that changes object programs in the relocatable format into object programs in the absolute format, establishing interprogram communication and including required subprograms from the subprogram library. These object programs can be executed at the completion of the LOADER RUN, or at a later time, depending on the wishes of the user.

Assumed Input Device. LOADER file on 1311 or 1301 UNIT 0, START 010400, END 012000 or TAPE UNIT 3.

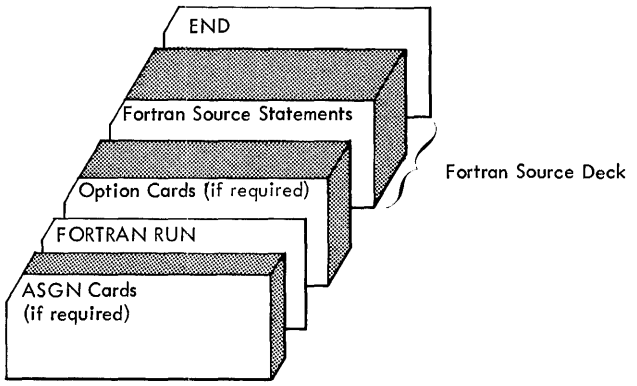


Figure 14. FORTRAN RUN with CONTROL and INPUT Files Assigned to the Same Device

Input. Object program(s) in the relocatable format.

Assumed Output Devices. LIST file on PRINTER 2, OUTPUT file on PUNCH 4 (1401 or 1460 systems) or PUNCH 1 (1440 systems).

Output.

1. Loader diagnostic messages on the LIST file, if errors are sensed.
2. Name map on the LIST file, unless an output option control card specifies that the name map be omitted.

Output Options Available.

1. Storage print on the LIST file, if an output option card is included that specifies a storage print.

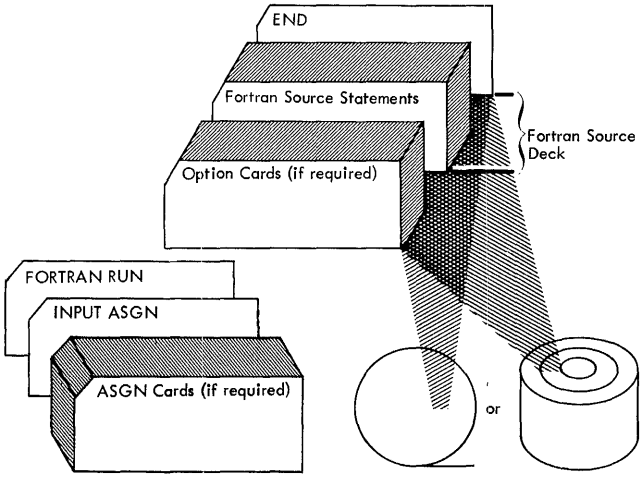


Figure 15. FORTRAN RUN with CONTROL and INPUT Files Assigned to Different Devices

2. Object program card deck in the absolute format on the OUTPUT file, if an output option card is included that specifies an absolute deck.

Additional Results. At the completion of a LOADER RUN, the program is ready for execution.

Required User Assignments. If the object program in relocatable format is on the LOADER file, no user-assignment is required. If the object program in relocatable format is on any other file (INPUT, WORK1-WORK6), an ASGN card is required designating this file. In this case, an associated output option card is required.

Control Cards.

The first card of a LOADER RUN job must be the LOADER RUN control card which is punched in the following manner:

Columns	*Contents*
6-11	LOADER
16-18	RUN

The last card of a LOADER RUN job must either be a $EXECUTION control card or a $NO EXECUTION control card.

a. If execution is desired, punch the following card:

Columns	*Contents*
1-10	$EXECUTION

b. If execution is not desired, punch the following card:

Columns	*Contents*
1-3	$NO
5-13	EXECUTION

Depending on the user's requirements, any or all of the following Loader control cards may be present between the LOADER RUN card and the $EXECUTION or $NO EXECUTION card. They may appear in any order.

a. The $INCLUDE card is punched in the following manner:

Columns	*Contents*
1-8	$INCLUDE
16-18	*three-character file name*
21-23	*three-character main program name*
or	or
21-26	*six-character subprogram name*

File Name	Three-Character File Name
LOADER	LDR
MESSAGE	MSG
INPUT	INP
OUTPUT	OUT
LIST	LST
WORK1	WK1
WORK2	WK2
WORK3	WK3
WORK4	WK4
WORK5	WK5
WORK6	WK6

● Figure 16. Equivalence Between Logical File Names and Three-Character File Names

This card causes the Fortran Loader to search the file specified in columns 16-18 for the relocatable main program specified in columns 21-23, or the relocatable subprogram specified in columns 21-26. When the relocatable program is found, it is loaded into core storage and the next control card is read. When columns 21-26 are blank, the entire file of relocatable programs is loaded. The valid file names which may appear in columns 16-18 are; LDR, INP, WK1, WK2, WK3, WK4, WK5, and WK6. The correspondence between the three-character file names and the actual file names is shown in Figure 16.

b. If an absolute deck is desired, punch the following card. (Unless otherwise specified in columns 21-23, the deck will be on the OUTPUT file.)

Columns	*Contents*
1-9	$ABSOLUTE
11-14	DECK
21-23	*three-character file name*

c. If a storage print is desired, punch the following card. (Unless otherwise specified in columns 21-23, the storage print will be on the LIST file.)

Columns	*Contents*
1-8	$STORAGE
10-14	PRINT
21-23	*three-character file name*

d. If no name map is desired, punch the following card:

Columns	*Contents*
1-3	$NO
5-8	NAME
10-12	MAP

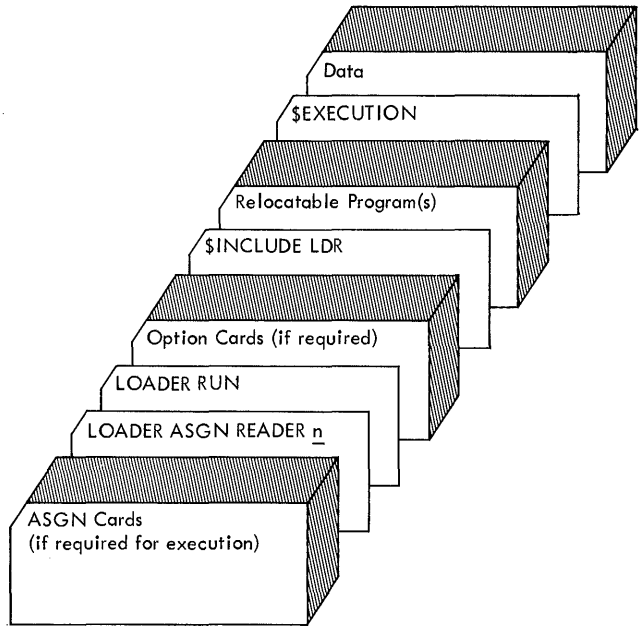


Figure 17. LOADER RUN with CONTROL and LOADER Files Assigned to the Same Device

Arrangement. The arrangement of input cards for a LOADER RUN is shown in Figures 17 and 18. The output option cards must be in the CONTROL file and can be in any order.

NOTE: If execution is to follow immediately after the LOADER RUN, indicated by a $EXECUTION card, the user must make sure that any Fortran numerical files referenced in the program have been assigned to the correct input/output devices. If ASGN cards are required to change file assignments, the cards precede the LOADER RUN card. Further, the user must make sure that the files referenced during a LOADER RUN do not conflict with files referenced in the object program.

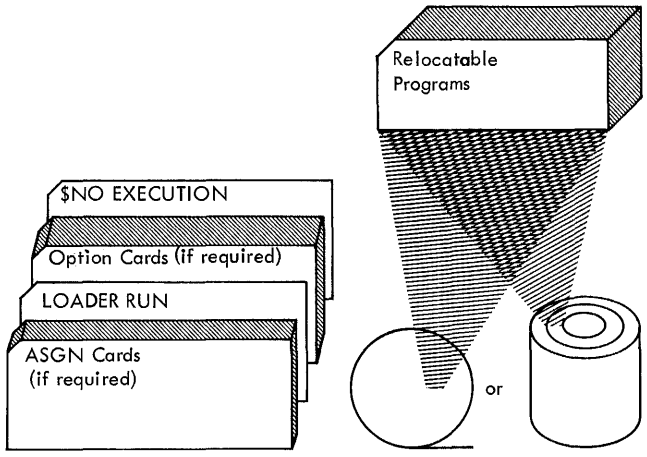


Figure 18. LOADER RUN with CONTROL and LOADER Files Assigned to Different Devices

Fortran Loader Operation

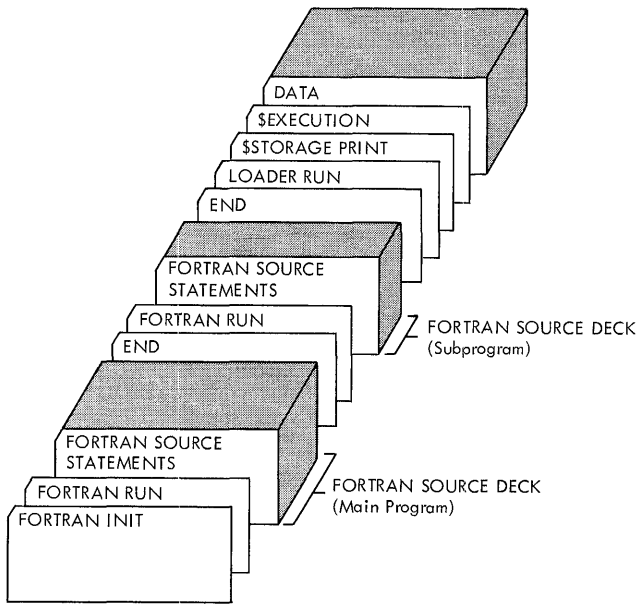
The Fortran LOADER is called by the System Control Program when the LOADER RUN card is read from the CONTROL file. All cards after the LOADER RUN card (up to and including the $EXECUTION or $NO EXECUTION card) on the CONTROL file are read by the Fortran LOADER. Relocatable programs may be loaded through the use of $INCLUDE cards as previously described. In addition, the entire LOADER file, which is developed by one or more Fortran runs, is loaded when the $EXECUTION or $NO EXECUTION card is read. The user should be aware that the LOADER file is not referenced if it has been referred to in a $INCLUDE card in the same LOADER run, or if it has been omitted with an ASGN card.

The LIBRARY file is always referenced and is the last file referenced in a LOADER run. Any subprograms, either supplied by IBM or entered on the LIBRARY file by the user, which are needed by the main program and/or subprograms already loaded, are extracted from the LIBRARY file.

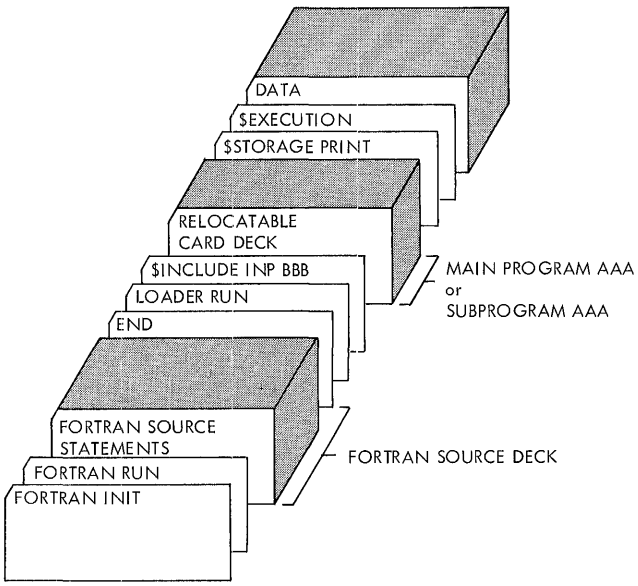
Following are three possible executions of the LOADER RUN job:

1. If the user has written a main program and an associated subprogram and wishes to compile and execute them, the card deck sequence of Figure 19 could be used.

   The two Fortran run jobs will place their respective relocatable decks on the LOADER file. Since there is no $INCLUDE card, at LOADER run time the entire LOADER file will be loaded when the $EXECUTION card is sensed.



● Figure 19. Fortran runs followed by *LOADER RUN*



● Figure 20. Main Program with Subprogram in Relocatable Card Form

2. If the user wishes to compile a Fortran program and then execute this program along with another program in relocatable card form, he may use the card deck sequence shown in Figure 20. One of the programs must be a main program and the other a subprogram. A program in the relocatable card form may be obtained from a Fortran run by assigning the OUTPUT file to the card punch.

   The Fortran run will place a relocatable deck on the LOADER file. When the $INCLUDE INP AAA card is read at LOADER RUN time, the Fortran loader loads program AAA from the INPUT file. Since this example uses the assumed logical file assignments, the INPUT file is on the same card reader as the CONTROL file. Therefore, the relocatable program AAA must immediately follow the $INCLUDE card.

   When the $EXECUTION card is read, the LOADER file, containing the program just compiled, will be loaded.

3. If the user wishes to load and execute one or more programs all of which are in the relocatable card form, he may use the card deck sequence shown in Figure 21.

   One of the three relocatable programs must be a main program, and the other two, subprograms.

   In this example, the CONTROL file, the INPUT file, and the LOADER file are all assigned to the same card reader. When each $INCLUDE card is read, the Fortran loader loads the specified relocatable deck from the LOADER file. When the $EXECUTION card is sensed, the Fortran loader will not reference the LOADER file again since it was referenced previously in a $INCLUDE card.

PRODUCTION RUN

This is the type of run that executes an object program in the absolute format. In order to perform a PRODUCTION RUN, the object program must be in the SYSTEM file, stored as a phase of the Fortran system. The method for placing the object program in the SYSTEM file is described in the following section, *Preparing User-Update Jobs.*

Assumed Input Device. For 1301, the SYSTEM file is on UNIT 0. For 1311, and for tape, the unit is user-assigned.

Input. Object program in the absolute format, stored as a phase of the Fortran system.

Assumed Output Devices. Not applicable.

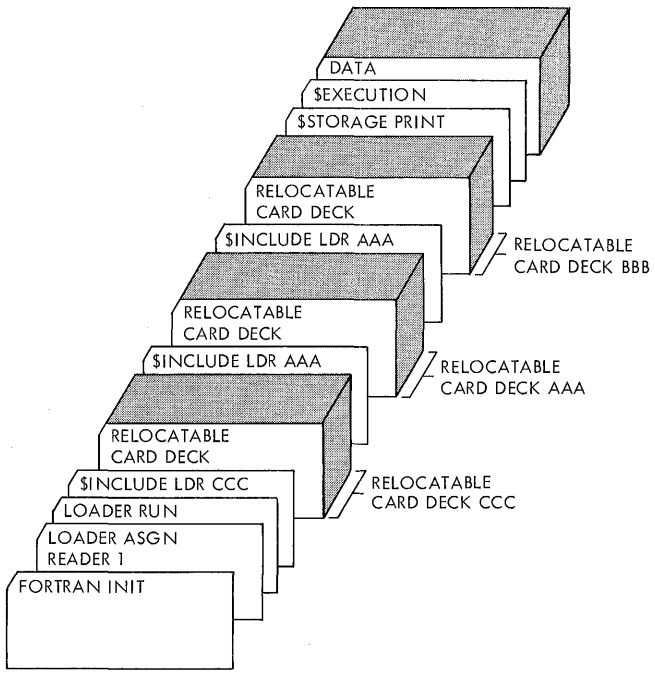
Output. Not applicable.

Required User Assignments. The unit(s) referenced by the source program.

Control Cards. The required RUN card is punched in the following manner.

Columns	*Contents*
6-15	PRODUCTION
16-18	RUN
21-23	*three-character phase name*

The *three-character phase name* is the three-character name assigned to the program before it was compiled. It is the same three-character name that appeared in columns 21-23 of the first card of the



● Figure 21. Main Program with Two Subprograms in Relocatable Card Form

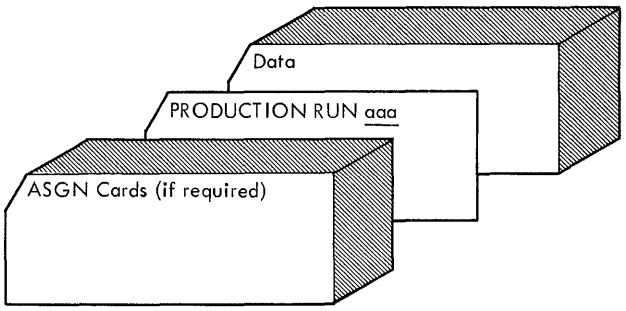


Figure 22. PRODUCTION RUN with CONTROL and INPUT (Equivalent to Fortran File 1) Files Assigned to the Same Device

absolute deck that was used to insert the program as a phase on the SYSTEM file.

Arrangement. The arrangement of input cards for a PRODUCTION RUN is shown in Figures 22 and 23.

Preparing User-Update Jobs

In order to perform a PRODUCTION RUN job, it is necessary that the object program in the absolute format be present in the SYSTEM file. This object program deck is output by a LOADER RUN, when specified by the user. Object programs can be inserted on or deleted from the SYSTEM file according to the needs of the user. If the system resides on a disk unit, these user-inserted phases (object programs in the absolute format) reside within the file-protected limits of the SYSTEM file.

On a tape-oriented system, the WORK1 file is used in conjunction with the SYSTEM file when a user-update job is performed. After a new phase is inserted (or deleted), the new SYSTEM file is present on WORK1. If the LIBRARY file followed the old SYSTEM file, it is copied following the new SYSTEM file on WORK1. According to the needs of the user, the WORK1 file can be transferred back to the master tape by performing a system-tape copy job. See *Duplicating the System Tape.* Since a system-tape copy job copies *from* the SYSTEM file *to* the WORK1 file, the user must make sure that the SYSTEM and WORK1 file assignments used for the user-update job are interchanged for the system-tape copy job. (Inserting a PAUSE card immediately after the user-update job and immediately preceding the system-tape copy job provides a temporary halt in the system that allows the user to interchange the two file assignments.)

The tape user is advised that when performing an UPDAT INSERT job, the phase after which the new phase is to be inserted must be present on the original SYSTEM file. The order of insertions (or deletions) must be the same as on the SYSTEM file.

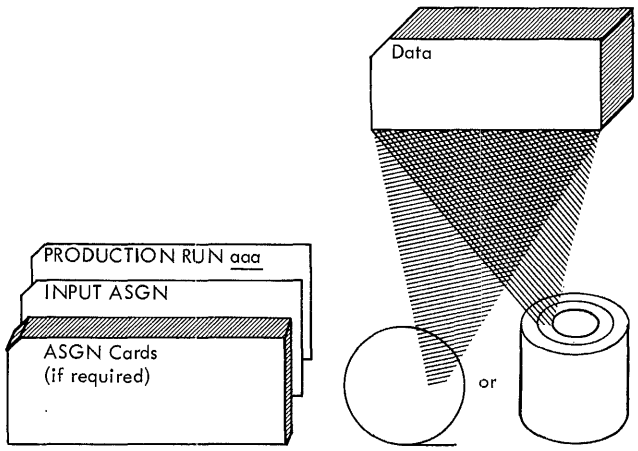


Figure 23. PRODUCTION RUN with CONTROL and INPUT (Equivalent to Fortran File 1) Files Assigned to Different Devices

Assumed Input Device. CONTROL file on READER 1.

Input. Object deck in the absolute format. This deck is obtained by selecting the absolute deck option in a LOADER RUN.

Assumed Output Devices. Not applicable.

Output. Not applicable.

Required User Assignments. None.

Control Cards.

1. If a phase is to be inserted on the SYSTEM file, the required UPDAT control card is generated by the Fortran loader. It is the first card of the absolute deck when the user specifies that an absolute deck be punched as a result of a LOADER RUN.

NOTE. If the SYSTEM file resides on tape, the user must specify that the phase be inserted after a particular phase on the SYSTEM file. In order that this be accomplished, punch the three-character phase name after which the phase is to be inserted in columns 21-23 of the UPDAT control card, the first card of the absolute deck. (Columns 21-23 of the UPDAT card punched by the LOADER contains the name of the phase that is to be inserted and must be changed.) 79F is the name of the last phase of the Fortran system. When deleting a phase and inserting another phase in its place, it is recommended that these two user-update jobs be performed in two separate stacks.

2. If a phase is to be deleted from the SYSTEM file, punch the following card.

Columns	*Contents*
6-15	[*any user comments*]
16-20	UPDAT
21-23	*three-character phase name*
24	comma
25-30	DELETE

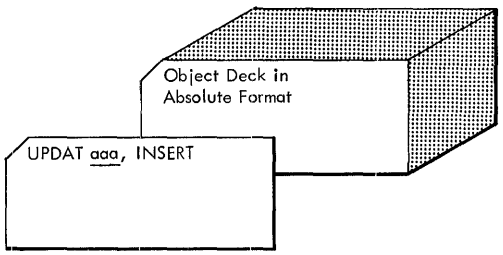


Figure 24. User-Update Job

In the control card, *three-character phase name* refers to the three-character main program name.

Arrangement. The arrangement of input cards for a user-update job is shown in Figure 24.

NOTE TO DISK USERS: Attempts to add a phase to the system may result in a halt (A-Address Reg.—088) indicating that the phase has not been inserted due to a lack of space in the system area. The following message will be printed before the halt occurs: SYSTEM AREA MUST BE OPTIMIZED BEFORE PHASE *name* CAN BE INSERTED. PRESS START TO OPTIMIZE. Pressing START will cause the system area to be scanned for all unused sectors. If unused sectors are found (resulting from prior deletions), the system area will be rearranged so that all unused sectors become available for phase insertion. Upon completion of this compression, another attempt will be made by the system to insert the phase. If the phase will not fit after compression, a hard halt will occur (A-Address Reg.—099). The following message will be printed before the halt occurs: NO ROOM IN SYSTEM AREA FOR PHASE *name-number* MORE SECTORS REQUIRED. Phases on the system that are no longer used may be deleted in order to make room for the phase to be inserted. A table containing the name and disk address of every phase in the system area is located in the file-protected area. A list of this table may be obtained by printing sectors 262583 through 262615 using a seek address of 002583 through 002615 (Load mode).

Preparing Library Jobs

Library jobs are associated with the maintenance of the Fortran library. The Fortran library is a mass-storage file that supports the Fortran loader. The file contains a library table (disk-resident systems only) and subprograms, such as standard Fortran functions and subroutines.

The three standard library jobs are:

1. Library build that enables the user to *define* a LIBRARY file. A library-build job, performed when a disk-resident Fortran system is built, defines a LIBRARY file on the same disk unit as the SYSTEM file. The limits of this LIBRARY file are 012000 *through* 013899. Thus, the assumed assignment is 1311 or 1301 UNIT 0, START 012000, END 013900.

   After the library-build job has been performed, the LIBRARY file contains the library table. The library table is thirty sectors in length. The user can enlarge the name table according to his needs.

   If the system is tape resident, the library can be built on the same tape unit with the SYSTEM file, or on another tape unit, if specified by a LIBRARY ASGN card. As a result, the tape user need only be concerned with library changes and library listings.
2. Library listing that enables the user to get a list of the library subprograms and/or the names of the subprograms that are in the LIBRARY file.
3. Library change that enables the user to modify the content of a LIBRARY file. A library-change job, first performed when the disk-resident system is built, transfers the subprograms to the LIBRARY file after the file has been defined.

A library job begins with a LIBRARY RUN card and terminates upon encountering the END card. Only four types of control cards can appear between the LIBRARY RUN and the END card. They are BUILD, LIST, INSER, and DELET.

At the completion of a library job (LIBRARY RUN) on a disk-oriented system, three messages are printed on the LIST file. The messages are:

END OF LIBRARY RUN
LIBRARY ASSIGNED *nnnnnn* TO *nnnnnn*
REMAINING SECTORS *nnnnnn* TO *nnnnnn*

In the message, *nnnnnn* is a disk address. From these messages, the user is able to determine the size of the present library, and the number of sectors available for any additional subprograms that may be added.

Any subprogram is stored in disk storage one card per sector in the move mode. As the input for a LIBRARY RUN must be in card form, the user can determine whether a subprogram will fit in the library by merely counting the cards in the deck output from a FORTRAN RUN.

If the LIBRARY file is full and the user wants to add a new subprogram, one of two steps can be followed.

1. The user can define and build a new LIBRARY.
2. The user can delete an existing subprogram from the LIBRARY file and insert the new subprogram. This can be done if the new subprogram will occupy the same number or fewer sectors than the old subprogram.

For tape-oriented systems, at the completion of a LIBRARY RUN, the message,

END OF LIBRARY RUN

is printed on the LIST file.

Library Build

Library-build jobs apply only to disk-resident systems. Each library-build job defines a LIBRARY file that contains a name table 30 sectors in length. If a table of more (or less) than 30 sectors is required, specify the sector number desired in the control card.

The control cards required for a library-build job are:

1. A LIBRARY ASGN card is required if the assignment of the LIBRARY file differs from that assumed by the System Control Program. Punch the ASGN card in the following manner:

Columns	*Contents*
6-12	LIBRARY
16-19	ASGN
21-57	1311 UNIT *n*, START *nnnnnn*, END *nnnnnn* *or* 1301 UNIT *n*, START *nnnnnn*, END *nnnnnn*

   For disk, the value *n* indicates the number of the disk unit, and can be 0, 1, 2, 3, or 4; *nnnnnn* represents a disk address. The limits of the library must be supplied.

2. Punch the required RUN card in the following manner:

Columns	*Contents*
6-12	LIBRARY
16-18	RUN

3. Punch the library-build card in the following manner:

Columns	*Contents*
16-20	BUILD
21-23	[*nnn*]

   The value *nnn* is used only when the name table is to differ from 030 sectors.

4. The END card must be the last card of a library-build job. Punch the END card in the following manner:

Columns	*Contents*
16-18	END

The arrangement of control cards for a library build job is shown in Figure 25. The cards are read from the CONTROL file.

Figure 25. Library Build

Library Listing

The user can request three types of library listings.

1. A listing of the names or headers of all the subprograms in the Fortran library.
2. A listing of the entries in a specific subprogram.
3. A listing of the entries in every subprogram.

The control cards required for a library-listing job are:

1. A LIBRARY ASGN card is required if the assignment of the library file differs from that assumed by the System Control Program. See *Library Build* for the format of the ASGN card.

2. The required RUN card is punched in the following manner.

Columns	*Contents*
6-12	LIBRARY
16-18	RUN

3. One of the following three cards are required for the library-listing job. The one that is selected depends upon the type of listing that is required.

   a. If a listing of the headers of all the tape subprograms is required, punch the following card. If a listing of the names and disk addresses of all the disk subprograms is required, punch the following card. The listing is output on the LIST file.

Columns	*Contents*
16-19	LIST
21-27	HEADERS

   b. If a listing of the entries in a specific subprogram is required, punch the following card. The listing is output on the LIST file.

Columns	*Contents*
6-11	*name*
16-19	LIST

   name is the six-character name of the specific subprogram entries that are required.

   c. If a listing of the entries in every subprogram is required, punch the following card. The listing is output on the LIST file.

Columns	*Contents*
16-19	LIST

4. The END card must be the last card of a library-listing job. Punch the END card in the following manner.

Columns	*Contents*
16-18	END

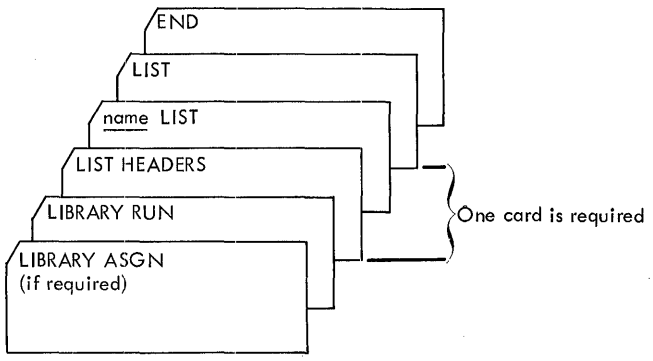


Figure 26. Library Listing

The arrangement of control cards for a library-listing job is shown in Figure 26. The cards are read from the CONTROL file.

Library Change

Fortran subprograms, supplied by IBM or developed by the user, can be added, modified, or deleted.

IBM provides a change deck whenever IBM-supplied standard subprograms should be modified. The change deck includes a LIBRARY RUN card, INSER and/or DELET cards, an END card, and cards containing the changes to be made.

Programs already in the library, if being replaced, should be deleted before new programs are inserted. In addition, all deletions should be performed before the first insertion. This becomes more significant from an efficiency standpoint as the number of programs being replaced increases.

User-change cards must be in the relocatable format, the result of processing by a FORTRAN RUN. In addition to the change cards, the following control cards are required for a library change.

1. A LIBRARY ASGN is required if the assignment of the LIBRARY file differs from that assumed by the System Control Program. See *Library Build* for the format of the ASGN card.
2. The required RUN card is punched in the following manner.

Columns	*Contents*
6-12	LIBRARY
16-18	RUN

3. If a subprogram is to be inserted, punch the following card.

Columns	*Contents*
6-11	*name*
16-20	INSER

The INSER card must immediately precede the relocatable deck. Each card of a relocatable deck is identified in column 72. The first card of this deck must contain an "H" in column 72 and the last card must contain a "." in column 72. The insertion of the program begins after reading of the "H" card and terminates after reading of the "." card. Termination of an insertion procedure by any other means, such as detecting the last card in the reader, must be avoided. Cards appearing between the first and the last card must have one of the characters "C", "K", "E", "R", or "O" in column 72.

When the LIBRARY file resides on disk, the new subprogram is inserted after the last subprogram in the LIBRARY file. When the LIBRARY file resides on tape, the new subprogram is inserted before the first subprogram.

If a subprogram by the same name already exists in the LIBRARY file, it is deleted before the new subprogram is inserted.

When the system is tape-oriented, the WORK1 file is used in conjunction with the LIBRARY file only when the library is to be changed. Whenever a subprogram is inserted, replaced or deleted, the new library will appear on the WORK1 file. The original library will not be changed. Perform a library-copy job to transfer the library from the WORK1 file to the LIBRARY file. *See Library Copy.*

4. If a subprogram is to be deleted, punch the following card.

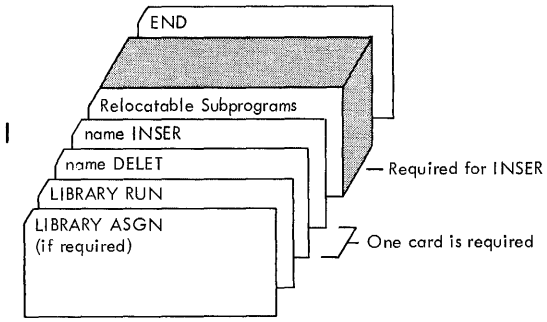
Columns	*Contents*
6-11	*name*
16-20	DELET

name is the six-character name of the subprogram to be deleted.

All DELET cards must precede all INSER cards. When the system is tape oriented, the WORK1 file is used in conjunction with the LIBRARY file when a subprogram is deleted. Perform a library-copy job to transfer the library from the WORK1 file to the LIBRARY file. See *Library Copy.*

5. The END card must be the last card of a library-change job. Punch the END card in the following manner.

Columns	*Contents*
16-18	END



● Figure 27. Library Change

The arrangement of the control cards and the input for a library change is shown in Figure 27. The control cards and the input cards are read from the CONTROL file.

If the input for a library-change job contains a card that is not recognized by the system, a halt occurs. The message

CARD NOT RECOGNIZED-BYPASS-CONTINUE INSERTION

is printed on the LIST file. In order to continue processing, press START.

Library Copy

The library-copy job is applicable only when the LIBRARY file resides on tape. This job is normally performed immediately after a LIBRARY RUN.

When a subprogram is inserted in place of a subprogram having the same name on the LIBRARY file, or when a subprogram is deleted from the existing library, the WORK1 file is used in conjunction with the LIBRARY file. At the completion of the insertion or deletion, the new or revised library is present on the WORK1 file.

To transfer the library to the LIBRARY file from WORK1, perform a library-copy job.

The following cards are required for a library-copy job.

1. A LIBRARY ASGN card is required if the assignment of the LIBRARY file differs from that assumed by the System Control Program. See *Library Build* for the format of the ASGN card.
2. The required RUN card is punched in the following manner.

Columns	*Contents*
6-12	LIBRARY
16-18	RUN

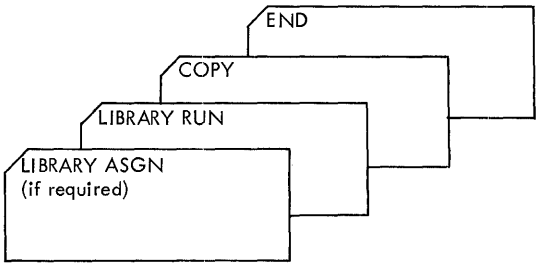


Figure 28. Library Copy

3. The required COPY card is punched in the following manner.

Columns	*Contents*
16-19	COPY

4. The END card must be the last card of a library-copy job. Punch the END card in the following manner.

Columns	*Contents*
16-18	END

The arrangement of the control cards is shown in Figure 28. The cards are read from the CONTROL file.

Changing File Assignments

Each logical file defined by the Fortran system, with the exception of the SYSTEM file, is assigned to a specific input/output device by the System Control Program. One set of logical file assignments applies to compilation (FORTRAN RUN). A second set of logical file assignments applies to execution (LOADER RUN and PRODUCTION RUN). These assignments can be temporarily changed by using ASGN cards.

Any assignment made by the user remains in effect until:

1. An ASGN card is sensed for the particular logical file, or
2. An INIT (initialize System Control Program assignments) card is sensed, or
3. A HALT card is sensed, signifying the end of a stack.

Preparing ASGN Cards

ASGN cards enable the user to change file assignments for one or more jobs in a stack. The general format for an ASGN card is:

file-name ASGN $\left\{ \begin{matrix} \textit{device} \\ \text{OMIT} \end{matrix} \right\}$

The *file-name* is the specific logical file; *device* is the input/output unit and/or area to which the logical file is assigned.

The assumed file assignments and ASGN card formats relating to specific files are shown in Figure 29. Valid device entries are shown in Figure 30.

ASGN cards are coded in the Autocoder format. When coding ASGN cards, the user must:

1. Leave blanks between items in the operand field as shown in Figure 29. If, for example, the OUTPUT file is to be assigned to disk area 004000 through 004799 on 1301 unit 1, the user would code the ASGN card for punching as shown in Figure 31. The END address that is coded is the address of the next available sector, not the address of the last sector to be used.
2. Left-justify entries in the label, operation, and operand fields, as shown in Figure 31.

File Considerations

SYSTEM File. If the SYSTEM file resides on 1311, drive 0 should be on-line because the System Control Program's assumed assignments are on drive 0. If drive 0 is not on-line, the user must use ASGN cards to change the assumed assignments for the LIBRARY, LOADER, WORK1, WORK2, and WORK3 files.

CONTROL and INPUT Files. If both the CONTROL and INPUT files are assigned to a reader, the assignments must be identical. For example, if the system is a 1440 and the CONTROL file is assigned to READER 1, the INPUT file must also be assigned to READER 1.

MESSAGE and LIST Files. If both the MESSAGE and LIST files are assigned to a printer, the assignment must be identical. For example, if the system is a 1401 and the MESSAGE file is assigned to PRINTER 2, the LIST file must also be assigned to PRINTER 2.

WORK1, WORK2, and WORK3 Disk Files. With disk systems, seek time is the most important factor affecting input/output operations. Therefore, it would be expedient for the user with a multi-unit system to distribute the Fortran files to all of the units, thus making a significant reduction in seek time.

Because the WORK1, WORK2, and WORK3 files handle large amounts of data, inefficient use of the files results in an increase in Fortran time requirements. For the benefit of the single disk unit user, WORK1, WORK2, and WORK3 are handled in a special way. The special way is to assign them to the same area of the disk unit as shown in Figure 32. When this is done, the System Control Program "splits" each cylinder, causing WORK1 to occupy the upper half of each cylinder and WORK2 to occupy the lower half of each cylinder. WORK3 initially occupies the upper half, then is effectively "flipped" back and forth as the compilation progresses. A programmed false cylinder overflow is forced as each half cylinder is operated upon and the next upper or lower cylinder is used.

WORK1, WORK2, and WORK3 Tape Files. As with disk files, it would be to the user's advantage to distribute the WORK files to separate tape units. However, if the system contains only the minimum number of units, it is possible to assign WORK1 and WORK3 to the same unit, thus saving the fourth unit for other purposes, such as a LOADER file for batched output from the compiler.

Use of Logical Files at Object-time. At object-time, the user is free to use any of the logical files for input/output operations except the SYSTEM, CONTROL, LIBRARY, and LOADER files. In addition, the LIST file is used by the Fortran loader for diagnostic messages and the LIST and OUTPUT files are normally used for the output options that are specified by the user. Therefore, if the LIST file (Fortran numerical file name 3) is used, it would normally be assigned to the printer so as to be aware of diagnostics, should any occur.

If the Fortran system resides on 1311 or 1301, tape input and/or output is permitted during object-time (LOADER RUN or PRODUCTION RUN). Care must be exercised that no logical file referenced by the disk loader be assigned to tape.

NOTE. If a single IBM 1442 Card Read-Punch is being used for punch and read operations, the user is advised that if a punch operation follows a read operation, the last card that was read will be punched.

Using ASGN Cards

At the beginning of stack processing, the System Control Program causes assumed assignments from the SYSTEM file to become effective. Each assumed assignment remains in effect until an ASGN card for that file is sensed. Any changed file assignment remains in effect until the next ASGN card for that file, or until an INIT card, or until a HALT card is sensed. (An INIT card causes all assumed assignments to become effective.)

If a file-assignment change is applicable for an entire stack, place the ASGN card immediately ahead of the first RUN card.

If a file-assignment change is applicable only to a specific job, place the ASGN card immediately ahead of the RUN card for that job. To change the single file assignment back to the assumed assignment or to a

ASGN Card Formats			Assumed Assignment		Remarks
Label Field (Columns 6-15)	Operation Field (Columns 16-20)	Operand Field (Columns 21-72)	Compilation (FORTRAN RUN)	Execution (LOADER RUN or PRODUCTION RUN)	
SYSTEM	ASGN	{1311 UNIT n / 1301 UNIT 0 / TAPE UNIT n}	1311 unit: user-assigned 1301 unit: must be assigned to UNIT 0 Tape unit: user-assigned	1311 unit: user-assigned 1301 unit: must be assigned to UNIT 0 Tape unit: user-assigned	If the system residence is 1311 or 1301, the SYSTEM ASGN card is the only required ASGN card. It must follow the Card Boot in a stack of jobs. Any other SYSTEM ASGN cards in the stack are flagged and bypassed. If the user desires that the Fortran system use less than the number of core storage positions available in the processor machine, punch a comma in column 32,and 12K or 16K beginning in column 34. If the system residence is tape and the tape unit is 1, neither the Card Boot nor the SYSTEM ASGN card is required. (Pressing the TAPE LOAD key achieves the same purpose as the Card Boot and the SYSTEM ASGN card.) If the unit is 2, 3, 4, 5, or 6, both the Card Boot and the SYSTEM ASGN card are required to start systems operations.
CONTROL	ASGN	{READER n / CONSOLE PRINTER}	READER 1	READER 1	If the CONTROL file and the INPUT file are assigned to the card reader, the assignment must be to the same card reader.
MESSAGE	ASGN	{PRINTER n / CONSOLE PRINTER}	PRINTER 2	PRINTER 2	When the MESSAGE file is assigned to the CONSOLE PRINTER, carriage control characters used with the 1403 or 1443 printer may appear in the message. If the MESSAGE file and the LIST file are assigned to the printer, the assignment must be to the same printer. The MESSAGE file is equivalent to Fortran file 0.
LIST	ASGN	{PRINTER n / 1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n / OMIT}	PRINTER 2	PRINTER 2	If the LIST file and the MESSAGE file are assigned to the printer, the assignment must be to the same printer. The LIST file is equivalent to Fortran file 3.
INPUT	ASGN	{READER n / 1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n}	READER 1	READER 1	If the INPUT file and the CONTROL file are assigned to the reader, the assignment must be to the same reader. The INPUT file is equivalent to Fortran file 1.
OUTPUT	ASGN	{PUNCH n / 1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n / OMIT}	OMIT	PUNCH 4 (1401 and 1460 systems) PUNCH 1 (1440 systems)	The OUTPUT file is equivalent to Fortran file 2.
LIBRARY	ASGN	{1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n}	1311 UNIT 0, START 012000, END 013900 1301 UNIT 0, START 012000, END 013900 TAPE UNIT 1	1311 UNIT 0, START 012000, END 013900 1301 UNIT 0, START 012000, END 013900 TAPE UNIT 1	1311 is assumed if the SYSTEM file is assigned to 1311. 1301 is assumed if the SYSTEM file is assigned to 1301. Tape is assumed if the SYSTEM file is assigned to tape. At execution time (LOADER RUN), the LOADER file can be assigned to READER n. If the MESSAGE, LIST, and WORK5 files are assigned to a printer, the assignment must be to the same printer. The WORK1 file is equivalent to Fortran file 4. The WORK2 file is equivalent to Fortran file 5. The WORK3 file is equivalent to Fortran file 6. The WORK4 file is equivalent to Fortran file 7. The WORK5 file is equivalent to Fortran file 8. The WORK6 file is equivalent to Fortran file 9.
LOADER	ASGN	{1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n / READER n / OMIT}	1311 UNIT 0, START 010400, END 012000 1301 UNIT 0, START 010400, END 012000 TAPE UNIT 3	1311 UNIT 0, START 010400, END 012000 1301 UNIT 0, START 010400, END 012000 TAPE UNIT 3	
WORK1	ASGN	{1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n}	1311 UNIT 0, START 007200, END 010400 1301 UNIT 0, START 007200, END 010400 TAPE UNIT 4	1311 UNIT 0, START 007200, END 007800 1301 UNIT 0, START 007200, END 007800 TAPE UNIT 4	
WORK2	ASGN	{1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n}	1311 UNIT 0, START 007200, END 010400 1301 UNIT 0, START 007200, END 010400 TAPE UNIT 5	1311 UNIT 0, START 007800, END 008400 1301 UNIT 0, START 007800, END 008400 TAPE UNIT 5	
WORK3	ASGN	{1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n}	1311 UNIT 0, START 007200, END 010400 1301 UNIT 0, START 007200, END 010400 TAPE UNIT 4	1311 UNIT 0, START 008400, END 008900 1301 UNIT 0, START 008400, END 008900 TAPE UNIT 6	
WORK4	ASGN	{1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n / OMIT}	OMIT	1311 UNIT 0, START 008900, END 009400 1301 UNIT 0, START 008900, END 009400 OMIT for tape systems	
WORK5	ASGN	{1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n / PRINTER n / OMIT}	OMIT	1311 UNIT 0, START 009400, END 009900 1301 UNIT 0, START 009400, END 009900 OMIT for tape systems	
WORK6	ASGN	{1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n / OMIT}	OMIT	1311 UNIT 0, START 009900, END 010400 1301 UNIT 0, START 009900, END 010400 OMIT for tape systems	

● Figure 29. ASGN Card Formats and Assumed Assignments

Device Entry and Values of n and nnnnnn	Remarks
{1311 / 1301} UNIT n, START nnnnnn, END nnnnnn n is the number of the disk unit, and can be 0, 1, 2, 3, or 4. nnnnnn is a disk address.	The END address is the address of the next available sector. The values of nnnnnn must adhere to the following rules: 1. WORK1 and WORK2 files. If the disk unit is a 1311, the START address must be a multiple of 200. If the disk unit is a 1301, the START address must be a multiple of 800. The END address (1311 and 1301) must be a multiple of 40. 2. WORK3, WORK4, WORK5, and WORK6 files. The START address (1311 and 1301) must be a multiple of 100. The END address of each file must be a multiple of 10. In addition, WORK3 must be at least 300 sectors long. 3. LIBRARY File. For both 1311 and 1301, the START and END addresses must be multiples of 20. If these rules are violated, the system automatically narrows in the disk area to an area that does adhere to these rules.
TAPE UNIT n n is the number of the tape unit, and can be 1, 2, 3, 4, 5, or 6.	
READER n For 1402, n can be 0, 1, or 2. For 1442, n can be 1 or 2.	For 1402, n represents the pocket into which the cards are stacked. For 1442 or 1444, n represents the number of the unit.
PUNCH n For 1402, n can be 0, 4, or 8. For 1442, n can be 1 or 2. For 1444, n must be 3.	
PRINTER n n can be 1 or 2.	n represents the number of print positions available on the 1403 or 1443. For 1403, a 1 indicates 100 positions and a 2 indicates 132 positions. For 1443, a 1 indicates 120 positions and a 2 indicates 144 positions.
CONSOLE PRINTER	The console printer must be an IBM 1447 without a buffer feature.
OMIT	Select this option when the file is not to be used by the Fortran system. The LIST, OUTPUT, LOADER, WORK4, WORK5, and WORK6 files can be omitted.

● Figure 30. Valid Device Entries

different assignment, place the ASGN card immediately ahead of the RUN card for the next job that requires the effective file assignment to be changed. If all effective file assignments are to be changed back to the original assumptions of the system, place an INIT card after the last job that is to use the effective assignments.

Example. Figure 33 is an illustration that shows the use of ASGN cards. Assume that:

1. The stack consists of compiling and executing Job 1.
2. The stack is to be processed on an IBM 1401 system with IBM 1311 Disk Storage Drives.
3. The SYSTEM and LIBRARY files are located on drive 0.
4. Drive 1, drive 2, drive 3, and drive 4 are on-line.
5. ASGN card A specifies SYSTEM ASGN 1311 UNIT 0. A SYSTEM ASGN card is required for each stack of jobs when the system resides on a disk unit.
6. ASGN card B specifies WORK1 ASGN 1311 UNIT 1, START 007200, END 010400. ASGN card C specifies WORK2 ASGN 1311 UNIT 2, START 007200, END 010400. ASGN

Line	Label	Operation	OPERAND
0 1	OUTPUT	ASGN	1301,UNIT,1,,START,004000,,END,004800
0 2			
0 3			

Figure 31. Coding for an OUTPUT ASGN Card

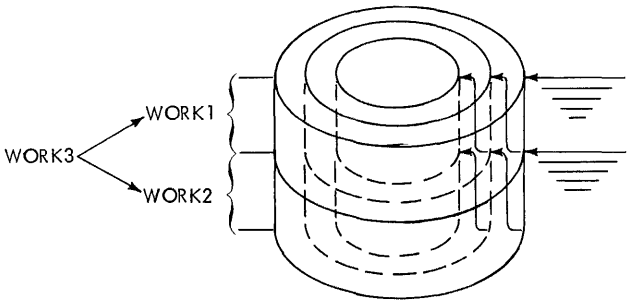


Figure 32. WORK1, WORK2, and WORK3 Assigned to the Same Disk Area

card D specifies WORK3 ASGN 1311 UNIT 3, START 007200, END 010400. ASGN card E specifies LOADER ASGN 1311 UNIT 4, START 010400, END 012000. Assigning disk files to separate disk units can make a substantial reduction in seek time.

7. ASGN card F specifies WORK5 ASGN PRINTER 2. WORK5 is equivalent to the Fortran numerical file name 8, which is to contain the results from Job 1.

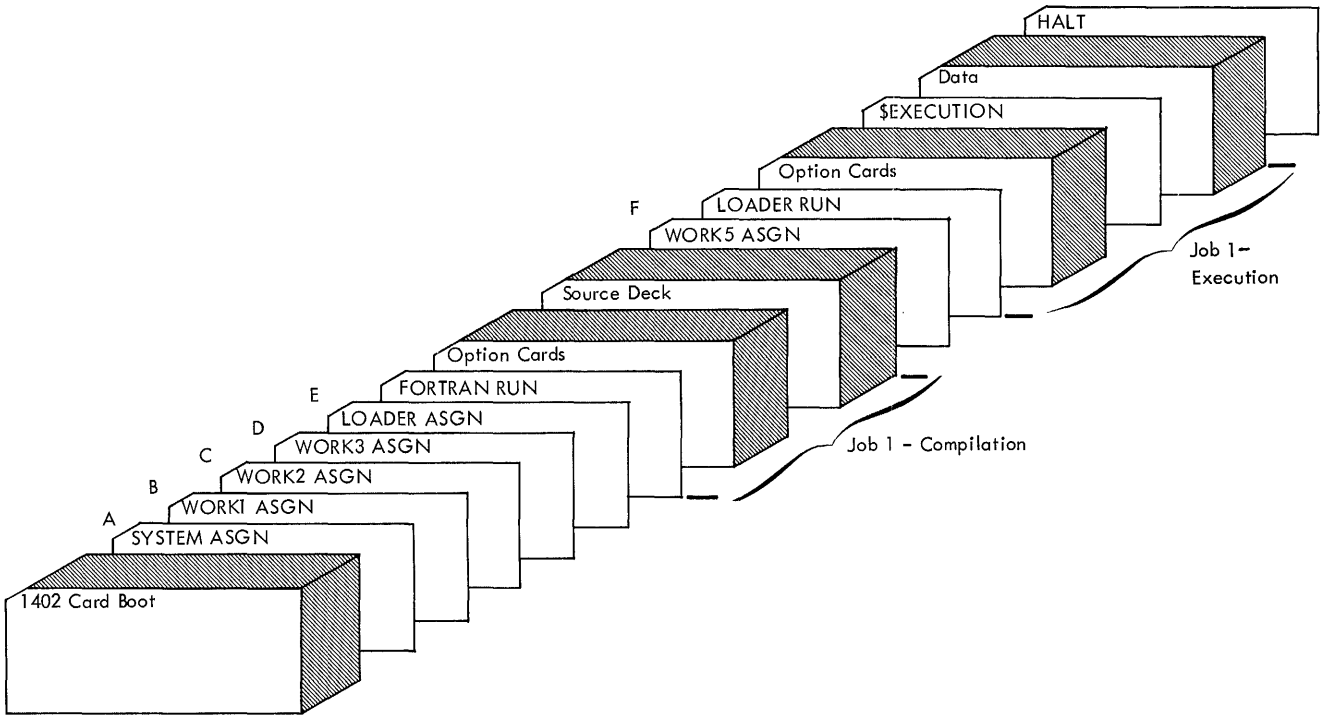


Figure 33. Using ASGN Cards

Performing Jobs

Under control of the System Control Program, it is possible to process one or more jobs without operator intervention. In order that this stack processing be accomplished, each separate job must be called for by the necessary control cards. A list of the operations that can be performed in a stack follows.

Logical File Assignments. Assign decks are made up of one or more ASGN control cards specifying input/output devices that differ from the effective devices of the System Control Program. With the exception of the SYSTEM ASGN card, logical-file ASGN control cards can appear as frequently within the stack as the user wishes. If the SYSTEM file resides on disk, or on any tape unit other than unit 1, the SYSTEM ASGN card appears once in a stack and immediately follows the *card-boot* deck. The user is reminded that no file assignment is reinstated to the original assumption unless specifically called for by an ASGN card, or until an INIT card is sensed.

When an INIT card is sensed, all logical file assignments revert to the assumed assignments of the System Control Program. At this time, the LOADER file is initialized for a new FORTRAN RUN job. Thus, by using an INIT card, the user can guarantee that his sequence of jobs will operate independently from any preceding jobs in the stack.

Library Maintenance. The composition of a library deck depends upon the nature of the library job. However, a LIBRARY RUN card and an END card are always required.

System Updating. Update decks as supplied by IBM or the user are read by the System Control Program and must be available to the system on the device to which the CONTROL file is assigned. An update deck consists of one or more control cards, followed by any appropriate data cards.

Processor Runs. Runs are dependent upon a RUN card and the input to the processors. If the INPUT file is assigned to the same device as the CONTROL file, i.e., the card reader, each source deck must be placed behind its respective RUN control card. If the input to the processor or programs is written in disk storage or on magnetic tape, an INPUT ASGN card is required designating the location of the source material.

Communicating with the Operator. NOTE control cards and PAUSE control cards can appear anywhere in a stack between jobs. A HALT card must be the last card of a stack.

Preparing a Stack

For a disk-resident system, the *card-boot* deck, a SYSTEM ASGN card, and a HALT card are always required. For a tape-resident system, the *card-boot* deck and the SYSTEM ASGN card are optional; the HALT card is required. The formats of the SYSTEM ASGN and HALT cards are shown in *Appendix I.*

The input cards for a stack are arranged in this order.

1. The 1402 or 1442 *card-boot* deck, which is optional if the tape-resident system is on unit 1.
2. The SYSTEM ASGN card, which is optional if the tape-resident system is on unit 1.
3. Job decks, to include *assign* card(s), *library* deck(s), *update* deck(s), and *processor* deck(s). Job decks can be in any order.
4. The HALT card.

This stack is placed in the card reader, and is read by the System Control Program from the CONTROL file.

Figure 34 shows a stack with CONTROL and INPUT files assigned to the same device.

Running a Stack

To perform a stack run when the system resides on 1311:

1. Place the system pack on the disk drive referred to in the SYSTEM ASGN control card and ready the drive. (This card immediately follows the 1402 or 1442 *card-boot* deck.)
2. Ready all the input/output devices to which the system logical files and/or the devices to which the Fortran numerical files are assigned. These are the assumed devices of the System Control Program and/or the devices defined by the ASGN cards. The assumed devices are: disk drive 0, the card reader, the card punch, and the printer.
3. Ready the console:
   a. Set the I/O check-stop switch off.
   b. Set the check-stop switch and disk-write switch on.
   c. Set the mode switch to RUN.
   d. Press CHECK RESET and START RESET.
4. Load the program.
   a. IBM 1402 Card Read-Punch: Press LOAD.
   b. IBM 1442 Card Reader: Press START on the reader, and PROGRAM LOAD on the console.

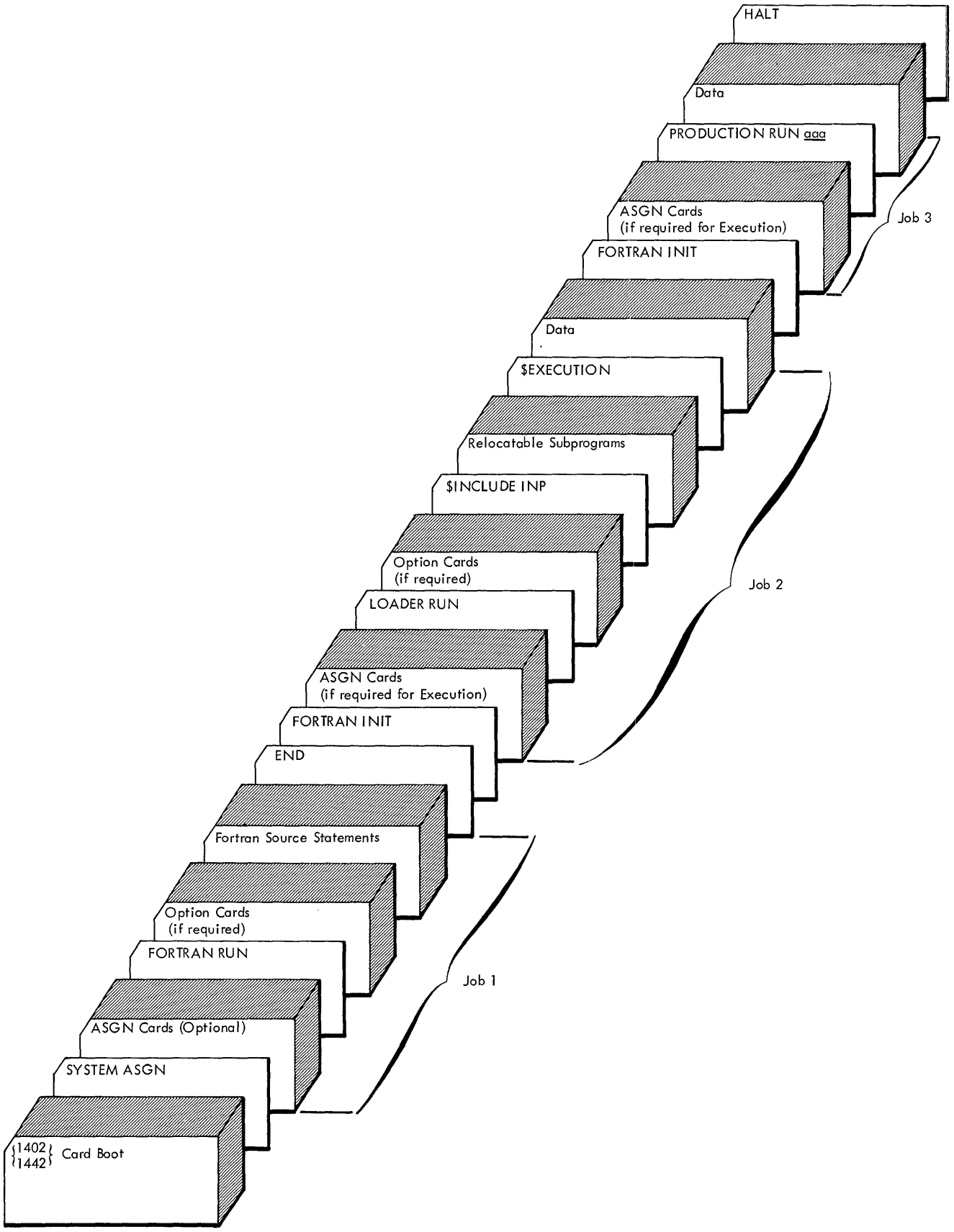


Figure 34. Stack with CONTROL and INPUT Files Assigned to the Same Device

5. When the system attempts to read the last card:
   a. IBM 1402 Card Read-Punch: Press START.
   b. IBM 1442 Card Reader: Press START on the card reader.

To perform a stack run when the system resides on 1301:

1. Ready all the input/output devices to which the system logical files and/or the devices to which the Fortran numerical files are assigned. These are the assumed devices of the System Control Program and/or the devices referred to in the ASGN cards. The assumed devices are: disk unit 0, the card reader, the card punch, and the printer.
2. Ready the console:
   a. Set the I/O check-stop switch off.
   b. Set the check-stop switch and disk-write switch on.
   c. Set the mode switch to RUN.
   d. Press CHECK RESET and START RESET.
3. Load the program:
   a. IBM 1402 Card Read-Punch: Press LOAD.
   b. IBM 1442 Card Reader: Press START on the reader, and PROGRAM LOAD on the console.
4. When the system attempts to read the last card:
   a. IBM 1402 Card Read-Punch: Press START.
   b. IBM 1442 Card Reader: Press START on the card reader.

To perform a stack run when the system resides on magnetic tape, and the *card-boot* deck and the SYSTEM ASGN card *are not* to be used.

1. Mount the system tape on unit 1.
2. Ready all the input/output devices to which the system logical files and/or the devices to which the Fortran numerical files are assigned. These are the assumed devices of the System Control Program and/or the devices defined by the ASGN cards. The assumed devices are: tape units 1, 3, 4, and 5 (FORTRAN RUN) or tape units 1 and 3 (LOADER RUN) or tape units 4, 5, and 6 (PRODUCTION RUN), and the card reader, the card punch, and the printer.
3. Ready the console:
   a. Set the I/O check-stop switch off.
   b. Set the check-stop switch on.
   c. Set the mode switch to RUN.
   d. Press CHECK RESET and START RESET.
4. Press TAPE LOAD.
5. When the system attempts to read the last card, press START.

To perform a stack run when the system resides on magnetic tape, and the *card boot* and the SYSTEM ASGN card *are* to be used:

1. Mount the system tape on the tape unit referred to in the SYSTEM ASGN control card, and ready the tape unit. (This card immediately follows the 1402 *card-boot* deck.)
2. Ready all the input/output devices to which the system logical files and/or the devices to which the Fortran numerical files are assigned. These are the assumed devices of the System Control Program and/or the devices defined by the ASGN cards. The assumed devices are: tape units 1, 3, 4, and 5 (FORTRAN RUN) or tape units 1 and 3 (LOADER RUN) or tape units 4, 5, and 6 (PRODUCTION RUN), and the card reader, the card punch, and the printer.
3. Ready the console:
   a. Set the I/O check-stop switch off.
   b. Set the check-stop switch on.
   c. Set the mode switch to RUN.
   d. Press CHECK RESET and START RESET.
4. Press LOAD.
5. When the system attempts to read the last card, press START.

Halts and Messages

The halts and messages shown in Figure 35 can appear during a stack run. To display the halt numbers, press the A-address register key. Messages are printed on the MESSAGE file.

Conditions may arise that the system recognizes as being instrumental in causing a failure. In these instances, the system automatically calls in a storage- and file-print program and continues by accepting a new job.

If the system is disk resident, the user can call in the storage- and file-print program by a manual branch to address 900. If the system is tape resident, the user can call in the storage- and file-print program by a manual branch to address 540. In both cases, a new stack can then be run.

WORK files are printed out successively beginning with WORK1. If an end-of-file indicator (a tape mark for tape files or a 1EOFb for disk files) is not present in a WORK file when the file-print program has control, the user can get a printout of the next WORK file by a manual branch to address 603 for disk-resident systems, or 472 for tape-resident systems.

Halt Number (A- Address Register	MESSAGE and/or Meaning	.Restart Procedure
001	WRONG SYSTEM. The message appears unconditionally on the printer.	1. Nonprocess run-out the cards in the reader. 2. Correct the SYSTEM ASGN card, or 3. Place the correct pack or tape on the unit indicated in the SYSTEM ASGN card. 4. Restart the stack.
002	TEN RD TRIES PRESS STRT FOR 10 MORE. The message appears unconditionally on the printer. It indicates any disk error while attempting to read the system file.	Press START for ten disk-read or tape-read retries.
003	SYSTEM ASGN NOT SENSED. The SYSTEM ASGN card did not immediately follow the card boot.	1. Nonprocess run-out the cards in the reader. 2. Place the SYSTEM ASGN card and the remainder of the stack in the read hopper. 3. If the reader is 1402, press START. 4. If the reader is 1442, press START on the reader and START on the console.
004	Parity check, wrong-length record, or no-address-compare error sensed 10 successive times during a disk or tape bootstrap operation.	Press START for 10 disk-read or tape-read retries.
005	End-of-file sensed in SYSTEM file during disk or tape bootstrap operation.	Nonprocess run-out the cards in the reader and restart the stack.
006	HALT card image. Indicates the end of the stack.	Hard halt.
007	Card-punch error.	1. 1402 card punch and 1444 card punch: nonprocess run-out the cards in the punch. Discard the last three cards (two nonprocessed cards and the card in error) in the stacker. Press START. 2. 1442 card punch: nonprocess run-out the cards in the punch. Discard the last two cards (the card in error and a blank card). Press START on the punch and START on the console.
008	Card-read error.	1. 1402 card reader: nonprocess run-out the cards in the reader. Place the last three cards (two nonprocessed cards and the card in error) in the hopper. Press START. 2. 1442 card reader: nonprocess run-out the cards in the reader. Place the two nonprocessed cards in the hopper. Press START on the reader and START on the console.
009	Printer error.	1. 1403 printer: press START. 2. 1443 printer: press START on the printer and START on the console.

● Figure 35. Halts and Messages (Part 1)

Halt Number (A-Address Register	MESSAGE and/or Meaning	Restart Procedure
010	Non-blank card at the punch station in the 1442 card read-punch.	Nonprocess run-out the cards in the 1442. Place blank cards before the nonprocessed cards. Press START on the 1442 and START on the console.
011	PAUSE card image.	Press START.
012	Console-printer error.	Press START for one retry of the read or write operation.
013	* * * ASGN card image. The halt indicates that the ASGN card is incorrectly punched.	1. 1402 card reader: the card in the stacker is the incorrect ASGN card. Correct the ASGN card. Nonprocess run-out the cards in the reader. Place the corrected ASGN card and the two nonprocessed cards in the hopper. Press START. 2. 1442 card reader: nonprocess run-out the cards in the reader. The first nonprocessed card is the incorrect ASGN card. Correct the ASGN card. Place the corrected ASGN card and the second nonprocessed card in the hopper. Press START on the reader and START on the console. 3. If the user wishes, he can ignore the two steps outlined above, and press START. The system will then use the effective device assignment for that particular file.
030	The object machine size has been assigned as 8K in the SYSTEM ASGN card.	Change the object machine size declaration in the SYSTEM ASGN card to 12K or 16K. Use Card Boot to restart.
031	In attempting to execute a COPY option during a library run (tape system only), the new or revised library tape was not found on WORK1 file.	Place library tape to be copied on tape unit assigned to WORK1 file. Press START.
032	In the tape system, when inserting a new subprogram in the library file, a card was found which could not be recognized. CARD NOT RECOGNIZED-BYPASS-CONTINUE INSERTION. REMOVE NECESSARY CARDS FROM READER. REPLACE REMAINING CHANGE CARDS AND END CARD. PRESS START.	Nonprocess run-out cards in the reader. Remove cards as required. Press START.
033	In the tape system, when performing an UPDAT job, the phase name specified in the UPDAT card was not found on the SYSTEM file. phase name NOT FOUND	Nonprocess run-out the cards in the reader. Remove cards as required. Pressing START will cause a return of control to the System Control Program.
034	In the tape system, when replacing a subprogram in the library file, a card was found which could not be recognized. CARD NOT RECOGNIZED-BYPASS-CONTINUE INSERTION.	Nonprocess run-out the cards in the reader. Correct card and begin library run over from the beginning by pressing START.

● Figure 35. Halts and Messages (Part 2)

Halt Number (A- Address Register)	MESSAGE and/or Meaning	Restart Procedure
035	All work files must be assigned to either a 1301 or 1311.	Change assignments and restart the stack with the Card Boot.
040	The logical file has been assigned to an area that overlaps a previously defined file label (1311 only).	Hard halt. Change the assignment and restart the stack with the Card Boot.
041	An end-of-file condition was encountered while reading from the INPUT file.	Restart the stack with the Card Boot.
042	An end-of-file condition was encountered while writing on the LIST file.	Restart the stack with the Card Boot.
043	An end-of-file condition was encountered while writing on the LOADER file.	Restart the stack with the Card Boot.
044	An end-of-file condition was encountered while writing on the OUTPUT file.	Restart the stack with the Card Boot.
045	An end-of-file condition was encountered while writing on a work file.	Restart the stack with the Card Boot.
046	An end-of-file condition was encountered while reading the library file (Disk system only).	Restart the stack with the Card Boot.
047	An end-of-file condition was encountered while writing on the file assigned for the name map.	Restart the stack with the Card Boot.
048	An end-of-file condition was encountered while writing on the file assigned for the Storage Print option.	Restart the stack with the Card Boot.
049	An end-of-file condition was encountered while writing on the file assigned for the Absolute Deck option.	Restart the stack with the Card Boot.
050	When producing an absolute deck, the arithmetic interpreter was not found in the library.	Rebuild the Fortran library before processing the remainder of the stacked input.
062	INVALID CONTROL CARD ASSUMED END OF LIBRARY RUN CORRECT AND RELOAD STACK PRESS START. Invalid control card appearing between LIBRARY RUN and END card. An END card condition will be simulated. CARD NOT RECOGNIZED ASSUMED END OF LIBRARY RUN CORRECT AND RELOAD STACK PRESS START. During an insertion procedure, an invalid card was encountered before reading of the "." card. A "." card will be generated, and an END card condition will be submitted.	Upon pressing START, control will return to SYSTEM CONTROL. The control file should be adjusted accordingly. Upon pressing START, control will return to SYSTEM CONTROL. The control file should be adjusted accordingly.

● Figure 35. Halts and Messages (Part 3)

Halt Number (A- Address Register)	MESSAGE and/or Meaning	Restart Procedure
066	NOT INSERTED- - TABLE AREA EXHAUSTED USE BUILD OPTION TO INCREASE NO. OF SECTORS This message will be printed when performing an INSER option and the library name table area has been exhausted. NOT INSERTED- - LIBRARY AREA EXHAUSTED This message will be printed when performing an INSER option and the library file has been exhausted.	Rebuild library before making any further library runs on this system. Rebuild library before making any further library runs on this system.
088	SYSTEM AREA MUST BE OPTIMIZED BEFORE PHASE name CAN BE INSERTED. PRESS START TO OPTIMIZE. (Disk users only) In attempting to add a phase to the system, the phase has not been inserted due to a lack of space in the system area.	Press START to cause the system area to be scanned for all unused sectors.
099	NO ROOM IN SYSTEM AREA FOR PHASE name-number MORE SECTORS REQUIRED. (Disk users only) After compression of the system area, there is still not enough space for a phase insertion.	Hard halt.
168	Phase not found in phase table while in supervisory call for phase (disk- resident systems only).	A part of the system must be rebuilt. Use the parts of the system deck labeled CARD BUILD, SYSTEM CONTROL, and FORTRAN COMPILER- LOADER- LIBRARIAN. Follow the procedures as described in Building a Fortran System.
371	Tape transmission error.	Press START for 10 tape- read or disk- write retries.
500	Disk not ready.	Ready the disk unit and press START.
629	Parity check, wrong- length record, or no- address- compare error sensed 10 successive times during a disk- read or write operation.	Press START for 10 disk- read or disk- write retries.
900	An END statement has been encountered during execution of user's program.	Press START for printout of storage. Otherwise use Card Boot to restart.
998	Tape subroutine for disk. Thirty attempts have been made to write a tape record.	Press START for another retry.
999	Tape subroutine for disk. Ten attempts have been made to read a tape record.	Press START for another retry.

● Figure 35. Halts and Messages (Part 4)

Halt Number (A- Address Register	MESSAGE and/or Meaning	Restart Procedure
	NOTE card image.	If a message is printed and no halt occurs, the next control card is processed.
	* * * card image. All cards not recognized by the System Control Program are flagged (***), written on the MESSAGE file, and bypassed by the system.	
	Card image INVALID UPDAT TYPE Update card with invalid update mode designated.	
	END CARD OMITTED ASSUMED END OF LIBRARY RUN The last card has been detected in the reader either during an insertion procedure or when attempting to read a control card.	If this message is received, the library should be carefully examined. If the error occurred during an insertion procedure, the library should be rebuilt.
	END OF LIBRARY RUN LIBRARY ASSIGNED TO REMAINING SECTORS TO Printed at the completion of a library job.	Control is returned to SYSTEM CONTROL.
	NOT FOUND IN LIBRARY Printed when performing a DELET or LIST option and the program was not found in the library.	The next library control card is read.
	EXISTING LIBRARY DELETED This message will be printed when rebuilding the library file.	A new library file will be defined and built.
	NO LIBRARY FOUND ON DISK This message will be printed when performing a LIST or DELET option and the library file has not been built.	The next library control card is read.
	PHASE XXX ALREADY ON SYSTEM. WILL DROP THIS SET OF CARDS	If a message is printed and no halt occurs, the next control card is processed.
	PHASE XXX NOT FOUND	
	HEADER CARD ERROR All header cards for disk must have 24232 in columns 1 through 5.	
	All header cards for tape must have 24235 in columns 1 through 5.	
	Card image PHASE AREA EXCEEDED	
	****PROCESSOR UNKNOWN****	
	{LST / OUT / INP} FILE {STARTS / ENDS} ON {1311 / 1301} UNIT n AT ADDRESS nnnnnn	

● Figure 35. Halts and Messages (Part 5)

Halt Number (A- Address Register)	MESSAGE and/or Meaning	Restart Procedure
	ROUTINE IN LIBRARY- DELETED AND INSERTED A program will be inserted which previously existed in the library file.	
	HEADER CARD MISSING- - BYPASS ROUTINE The first card read after the INSER card did not contain an "H" in column 72.	All cards on the input file will be printed until the next library control card is read.
	ASSUMED BUILD OPTION This message will be printed when performing an INSER option and a library file has not been previously defined.	
Halt Number (I- Address Register	MESSAGE and/or Meaning	Restart Procedure
1042	Tape system only. An end-of-file condition was encountered while writing on a work file.	Restart the stack with the Card Boot.
1124	Disk system only. An end-of-file condition was encountered while writing on a work file.	Restart the stack with the Card Boot.
2728	ERROR	See Figure 13.

● Figure 35. Halts and Messages (Part 6)

Building and Updating a Fortran System

Tape Residence System, Deck Description And Preparation

The tape supplied to the 1401 or 1460 user who wants a tape resident system contains a Fortran sample program, the card boot (used to start system operations), the Fortran system, a relocatable loader, and the Fortran Library. This tape is in card image form. These cards must first be punched from the tape, then used to create the tape resident system.

The card deck which can be used to start system operations consists of 13 cards, and is called the card boot. The first 6 cards are the 1402 load cards. They are numbered consecutively 1 through 6 in column 80, and identified by a 0-4-8 punch (% symbol) in column 79. The remainder of the cards are numbered consecutively 000 through 007 in columns 73-75, and identified by the code 51T02 punched in columns 76-80.

Building a Fortran Tape Resident System

The tape supplied to the 1401 or 1460 user is in card image form. Cards must first be punched from the tape, then used to build a tape resident system.

Punch the cards from the tape in the following manner:

1. Ready the card image tape on tape unit 1.
2. Ready the card punch.
3. Set the I/O Check Stop switch off.
4. Press CHECK RESET, START RESET, then TAPE LOAD.

A halt will occur and the following message will be printed on the printer.

A HALT WILL OCCUR AT EACH DECK SEGMENT.
MARK DECK AS PUNCHED, PRESS START TO CONTINUE.

The following B-register halts are applicable:

Halt Number (B-register)	*Meaning*
120	Printer error
121	Initial halt
122	Tape READ ERROR
123	Halt after each deck segment has been punched
999	End of Job halt

Four separate decks of cards will be punched. The first deck of cards is the Fortran sample program. The second deck of cards is the card boot. The third deck of cards contains the System Control program, the Fortran compiler and the Fortran relocatable loader.

The last deck of cards punched is the Fortran library of subroutines. The Fortran library is separated from the remainder of the Fortran system deck to enable the user to insert a Library ASGN card if he wishes the library to be placed on a tape separate from the system tape.

Build the Fortran tape resident system as follows:

1. Ready the tape that will become the System Tape on tape unit 4.
2. Ready a work tape on tape unit 1.
3. Place the deck marked System Control, Fortran compiler, Relocatable loader followed by the deck marked Fortran library in the card dreader.
4. Set the I/O check stop switch off.
5. Press START RESET and CHECK RESET.
6. Press LOAD on the Card Read Punch. A short program is written on tape unit 1, a message EOJ is printed and the system halts.
7. Press START RESET, TAPE LOAD, and START. The Fortran system will now be written on tape unit 4. A message PAUSE*****CHANGE TAPE UNITS*****, is printed.
8. Change tape unit 1 to 0; then change tape unit 4 to 1.
9. Press START RESET, TAPE LOAD, and START. The tape is read until the library portion is found, then the remaining cards are loaded on tape unit 1.
10. Press START to read last card. A message, PAUSE SYSTEM TAPE COMPLETED, is printed. File protect the tape on tape unit 1 and create a copy to be used for FORTRAN compilations. (Refer to *Duplicating the System Tape.)*
11. A halt with A-address register 519 during the building indicates a tape error. Press START for ten retries.

Disk Resident System, Deck Description And Preparation

The card deck supplied to the user who wants a disk resident system contains six sections as shown in Figure 36. One section, *marking program,* is used to separate the sections for ease in labeling the various components of the complete deck. Three sections, *write file-protected addresses, system control card build,* and *Fortran update* are used to build the system. A fifth section, the *card boot,* is used to operate the system. A sixth section, *sample program,* is used to test the system built by the user. The individual sections are separated by the *marking-program* control cards. In the instances where more than one set of cards comprises a section, a *marking-program* control card separates the sets.

To facilitate building and maintenance operations, mark the sections as indicated by the *marking-program* messages.

All cards in the system deck, except the four 1402 load-card sets and the four 1442 load-card sets, contain a sequence number in columns 73-75. The cards are numbered consecutively, beginning with 001. The first 1,000 cards have no zone punches above the sequence numbers. The second 1,000 have a 12-punch in column 75. The third 1,000 have a 12-punch in column 74. The fourth 1,000 have a 12-punch in column 73. The remaining cards have an 11-punch in column 75.

All load cards contain a sequence number in column 80. The 1402 load cards are numbered consecutively from 1 through 6 in column 80 and are identified by a 0-4-8 punch (% symbol) in column 79. The 1442 load cards are numbered consecutively from 1 through 7 in column 80 and are identified by a 3-8 punch (# symbol) in column 79.

If it should be necessary to resequence the system deck, the user should sort the cards in the following manner:

1. Sort on column 79 (0-4-8 punch) to select the 1402 load cards.
2. Sort the 1402 load cards on column 80 to sequence the cards.
3. Assemble the four sets of 1402 load cards.
4. Sort on column 79 (3-8 punch) to select the 1442 load cards.
5. Sort the 1442 load cards on column 80 to sequence the cards.
6. Assemble the four sets of 1442 load cards.
7. Sort the remaining cards into four groups corresponding to:
   a. No zone punch in any of the columns 73-75.
   b. 12-punch in column 75.
   c. 12-punch in column 74.
   d. 12-punch in column 73.
   e. 11-punch in column 75.

   Sort each of these groups on columns 75, 74, and 73.
8. An appropriate set of load cards makes up the first six cards (1402) or seven cards (1442) of the *marking* programs, *file-protect* programs, *card-build* programs, and *card boots.* Insert a 1402 or 1442 load card set preceding each of these programs according to the system (1401/1460 or 1440).

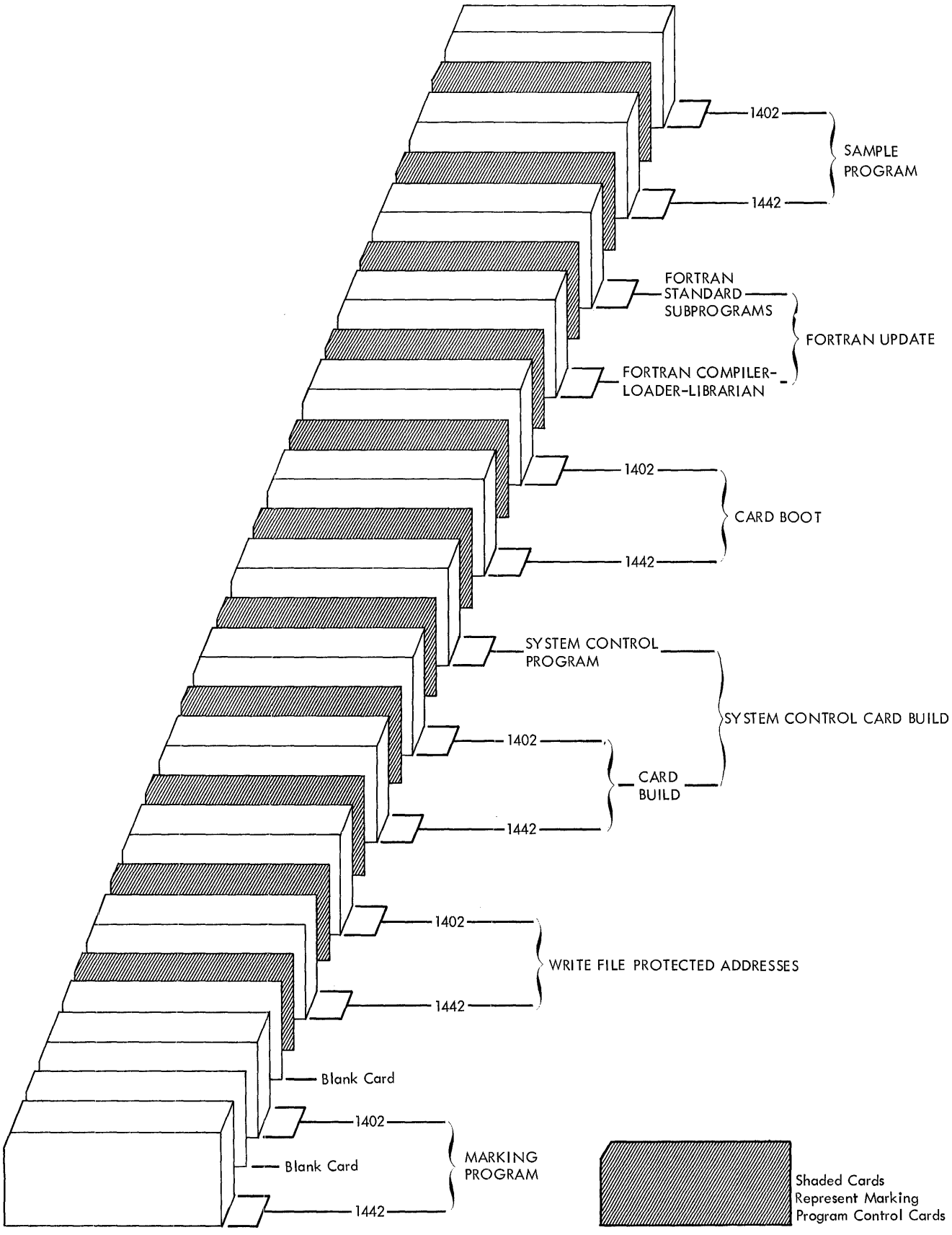


Figure 36. Fortran System Program Deck

Marking Program

The *marking-program* deck is made up of two sets. The set for the 1442 consists of 13 cards and, except for the load cards, has identification code 50ZY1 punched in columns 76-80. The set for the 1402 consists of 11 cards and, except for the load cards, has the identification code 50ZZ1 punched in columns 76-80. A blank card follows each set.

The *marking program* separates the various sections and sets that make up the system deck. When a control card is sensed, a halt occurs and a message is printed.

If the reader is 1442, the initial message is:

> HALT AT EACH DECK SEGMENT. DISCARD FIRST CARD, MARK DECK AS PRINTED, PRESS START TO CONTINUE.

If the reader is 1402, the initial message is:

> HALT AT EACH DECK SEGMENT. MARK DECK AS PRINTED, PRESS START TO CONTINUE.

Subsequent messages contain the name of the section to be marked.

To use the decks:

1. Set sense switch A on. Set all other sense switches off.
2. Set the I/O check stop switch off.
3. Press CHECK RESET and START RESET.
4. Select the *marking-program* deck that is appropriate for the system and remove the other deck.
5. Remove the blank card following the *marking-program* and place the program in the card reader, followed by the remainder of the Fortran system deck.
6. Load the program.
   a. IBM 1402 Card Read-Punch: Press LOAD.
   b. IBM 1442 Card Reader: Press START on the reader, and PROGRAM LOAD on the console.
7. Halt 003 procedure.
   a. IBM 1402 Card Read-Punch: Press START. The *marking program* is in the NR stacker.
   b. IBM 1442 Card Reader: Remove the *marking program* from stacker 1 and press START on the console.
8. Halt 001 procedure.
   a. IBM 1402 Card Read-Punch: Remove the cards from stacker 1 and press START. Mark the deck section as indicated in the message. The *marking-program* control card is in the NR stacker.
   b. IBM 1442 Card Reader: Remove the cards from stacker 1 and press START on the console. Discard the first card (*marking-program* control card) and mark the section as indicated in the message.

   NOTE: The *marking-program* control cards are identified by ##### in columns 1-5. These cards are only for the use of the *marking program* and should be discarded after the deck is marked.
9. When the system attempts to read the last card.
   a. IBM 1402 Card Read-Punch: Press START.
   b. IBM 1442 Card Reader: Press START on the reader. The last card is a *marking-program* control card and should be discarded.

The following halts can occur when using the *marking program*. To display the halt number, press the A-address register key.

Halt Number (A-Address Register)	*Meaning*
001	The deck section in stacker 1 should be marked.
002	End of job.
003	The initial message has been printed.
008	Card-read error. To retry the operation: *For the 1402:* Nonprocess run-out the cards. Remove the last three cards in the stacker and place them in the hopper. Press START. *For the 1442:* Nonprocess run-out the cards. Place the two nonprocessed cards in the read hopper. Press START on the reader and START on the console.
009	Printer error. To retry the operation, a. IBM 1403 Printer: Press START. b. IBM 1443 Printer: Press START on the printer and START on the console.

Write File-Protected Addresses

The *write file-protected addresses* section is punched in the Autocoder condensed-loader format. The deck consists of approximately 120 cards.

The set of cards for the 1442 has, except for the load cards, the identification code 50FS1 punched in columns 76-80. The set of cards for the 1402 has, except for the load cards, the identification code 50FP1 punched in columns 76-80.

This section writes disk addresses whose values are equal to the normal addresses plus 260,000. It is by use of these false addresses that the file-protected area is created.

System Control Card Build

This section contains control cards and cards punched in the Autocoder condensed-loader format. It includes both the 1402 and the 1442 *card-build* programs and

the System Control Program. All necessary control cards are incorporated within the section, which consists of approximately 1000 cards.

The *card-build* set for the 1442 has, except for the load cards, the identification code 50X41 punched in columns 76-80. The *card-build* set for the 1402 has, except for the load cards, the identification code 50X01 punched in columns 76-80.

The System Control Program section is identified by the code 50S*x*1 punched in columns 76-80, where *x* is alphameric. The section loads the System Control Program in disk storage.

Card Boot

The 1402 *card-boot* set, consisting of 17 cards, and the 1442 *card-boot* set, consisting of 19 cards, are punched in the Autocoder condensed-loader format. The 1442 *card-boot* set has, except for the load cards, the identification code 50SZ1 punched in columns 76-80. The 1402 *card-boot* set has, except for the load cards, the identification code 50PZ1 punched in columns 76-80.

Because the *card boot* is required for each stack of jobs to be performed by the system, the *card boot* must be removed and saved for future system operations.

Fortran Update

The *Fortran-update* section is made up of the Fortran compiler-loader-librarian set and the Fortran standard subprograms set.

The Fortran compiler-loader-librarian set is punched in the Autocoder condensed-loader format, Fortran relocatable format, and the UPDAT control card format. The set consists of approximately 3,000 cards and contains the phases of the Fortran compiler, the Fortran loader, and the library-build routine for the Fortran standard subprograms. The UPDAT cards are identified by the code UPFIV punched in columns 76-80. The Fortran compiler, loader, and librarian phases are identified by the code *nn*FIV punched in columns 76-80, where *n* is numeric.

The function of the deck is to load the Fortran processor phases on the disk unit, thus permitting a FORTRAN RUN or LOADER RUN, and to load librarian phases that control the building and maintaining of the Fortran library of standard and user-supplied subprograms.

The Fortran standard subprograms set is punched in the Fortran relocatable card format and the library control card format, and contains approximately 1,000 cards. The set has the identification code SSFIV punched in columns 76-80. This set places the Fortran standard subprograms on the LIBRARY file. Fortran requires that these subprograms be present during a LOADER RUN.

The last two cards of the set are a NOTE card and a HALT card which will cause the following comments to be printed on the MESSAGE file:

```
NOTE  SYSTEM BUILD COMPLETE
HALT  PREPARE SAMPLE PROGRAM
TO TEST SYSTEM
```

Fortran Sample Program

The Fortran Sample Program consists of approximately 150 cards. The sample program for the 1442 has the identification code S2FIV punched in columns 76-80. The sample program for the 1402 has the identification code S1FIV punched in columns 76-80. This source deck, written in the Fortran language, is used to test the effectiveness of the system built by the user.

Building A Fortran Disk Resident System

After all sets of cards have been labeled and those sets of cards not applicable to the user's system have been removed, the user is ready to use the prepared system deck to build the Fortran system.

Figure 37 is a block diagram showing the building of a disk-resident system.

The system unit must be prepared for writing the complete system from cards. The user must clear disk unit 0 in the move mode from 000000 to 000199, in the load mode from 000200 to 000259, in the move mode from 000260 to 000299, in the load mode from 000300 to 007199, and in the move mode from 007200 to 019979. The Clear Disk Storage Utility program applicable to the user's system can be used for this operation. As header labels are to be deleted, the write-address mode switch will initially be set off.

Figure 38 shows the disk storage allocation on the system unit.

The control cards for the utility program must be punched in the following manner.

For 1311,

Columns	*Contents*
1-15	M00000000019900
21-35	L00020000025900
41-55	M00026000029900

Columns	*Contents*
1-15	L00030000719900
21-35	M00720001997900

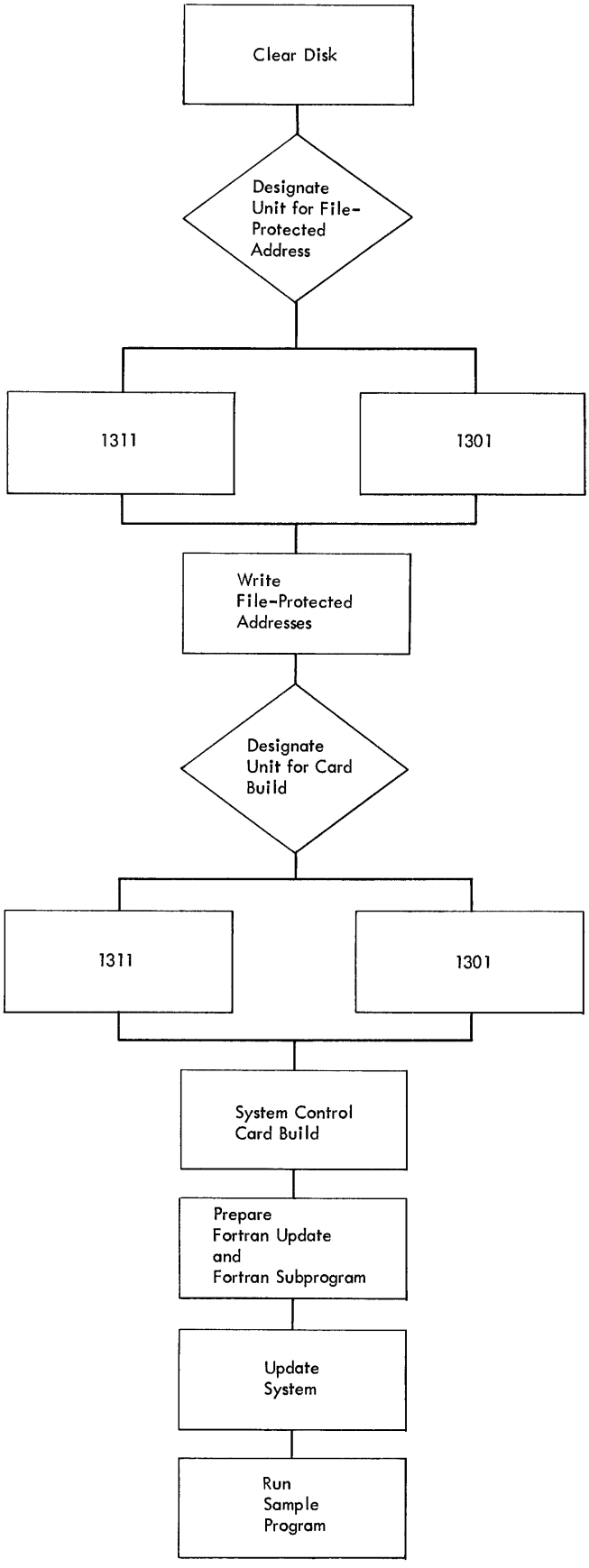


Figure 37. Building a Fortran System

For 1301,

Columns	*Contents*
1-15	M000000000199+++
21-35	L000200000259+++
41-55	M000260000299+++

Columns	*Contents*
1-15	L000300007199
21-35	M007200019979

The time required to clear the disk unit in the specified modes is approximately five minutes.

Write File-Protected Addresses

The last card in the section labeled WRITE FILE PROTECT is a control card that is partially prepunched. It is by the use of this control card that the limits of the file-protected area reserved for the SYSTEM file in the disk-storage unit are supplied. The user must indicate in the control card whether the system is to reside on a 1301 or 1311 disk unit. For both the 1301 and 1311, the system must be built on unit 0. In the case of the 1311, the system pack can be used on any drive once the system has been built. The control card is punched as follows:

Columns	*Contents*
1-15	FILE-PROTECT ON (prepunched)
17-20	1301 or 1311
22	0 (prepunched)
24-42	FROM NORMAL ADDRESS (prepunched)
44-49	002500 (prepunched)
51-52	TO (prepunched)
54-59	007200 (prepunched)

After columns 17-20 have been punched by the user, the card must be replaced as the last card of the section.

To use the section when the system is to reside on 1311:

1. Ready the pack on disk drive 0.
2. Set the write-address mode switch on.
3. Set the write-disk switch on.
4. Set the I/O check stop switch on.
5. Press CHECK RESET and START RESET.
6. Place the *write file-protected addresses* section in the card reader.
7. Load the program.
   a. IBM 1402 Card Read-Punch: Press LOAD.
   b. IBM 1442 Card Reader: Press START on the reader, and PROGRAM LOAD on the console.
8. When the system attempts to read the last card,
   a. IBM 1402 Card Read-Punch: Press START.
   b. IBM 1442 Card Reader: Press START on the reader.

File	Mode	File-Protected	Sector Range
SYSTEM File			
Not used	Move	No	000000-000199
	Load	No	000200-000259
	Move	No	000260-000299
	Load	No	000300-002499
System Control Program	Load	Yes	002500-002904
Fortran Processor Program	Load	Yes	002905-002979
Not used	Load	Yes	002980-002999
System Control Program	Load	Yes	003000-003175
Fortran Processor Program Area for User's Fortran Object Program Library	Load	Yes	003176-007199
WORK Files	Move	No	007200-010399
LOADER File	Move	No	010400-011999
LIBRARY File	Move	No	012000-013899

● Figure 38. Disk Storage Allocation

9. At the end of the job, set the write-address mode switch off.

To use the deck when the system is to reside on 1301:

1. Set the write-address mode switch on.
2. Set the write-disk switch on.
3. Set the I/O check stop switch on.
4. Press CHECK RESET and START RESET.
5. Place the *write file-protected addresses* section in the card reader.
6. Load the program.
   a. IBM 1402 Card Read-Punch: Press LOAD.
   b. IBM 1442 Card Reader: Press START on the reader, and PROGRAM LOAD on the console.
7. When the system attempts to read the last card,
   a. IBM 1402 Card Read-Punch: Press START.
   b. IBM 1442 Card Reader: Press START on the reader.
8. At the end of the job, set the write-address mode switch off.

The time required to perform this job is approximately 1 minute. The following halts can occur when writing file-protected addresses.

Halt Number (A-Address Register)	*Meaning*
020	Last card condition was sensed before the control card. The control card containing the initial and terminal addresses of the area to be file-protected must be the last card of the deck. When the system is restarted by pressing START, a read operation is performed.
021	An invalid disk type is specified in the control card. 1301 or 1311 are the only valid entries for columns 17-20 of the control card. When the system is restarted by pressing START, a read operation is performed.
022	An invalid disk unit is specified in the control card. The only valid entry for column 22 of the control card is 0. When the system is restarted by pressing START, a read operation is performed.
023	An invalid start address (columns 44-49) is specified in the control card. The start address must be 002500. When the system is restarted by pressing START, a read operation is performed.
024	An invalid end address (columns 54-59) is specified in the control card. The end address must be 007200. When the system is restarted by pressing START, a read operation is performed.
025	Disk unit 0 is not ready. When the system is restarted by pressing START, the disk I/O operation is retried.
026	The area specified in the control card is already file-protected (all or in part). If the system is restarted by pressing START, the entire specified area will be file-protected *and* cleared.
027	The area specified in the control card has neither the "normal" disk addresses (000000-?) nor file-protected addresses. This is a hard halt.

Halt Number (A-Address Register)	*Meaning*
028	Parity check or wrong-length record error occurred on the disk unit while writing addresses. When the system is restarted by pressing START, the disk I/O operation is retried.
029	Parity check or wrong-length record error occurred on the disk unit while determining the existing addressing scheme. This is a hard halt.
030	End of the job.

System Control Card Build

The last card in the section labeled CARD BUILD is a control card that is partially prepunched. It is by the use of this control card that disk residence is determined.

The user must indicate in the control card whether the system is to reside on a 1301 or 1311 disk unit. The assumed disk unit number is 0.

The control card is punched as follows:

Columns	*Contents*
6-11	SYSTEM (prepunched)
16-20	BUILD (prepunched)
21-24	1301 or 1311

After columns 21-24 have been punched by the user, the card must be replaced as the last card of the CARD BUILD deck.

The *system-control card build* consists of the card sections labeled CARD BUILD and SYSTEM CONTROL.

To use the *system-control card build* when the system is to reside on 1311:

1. Ready the pack on disk drive 0.
2. Set the write-address mode switch off.
3. Set the write-disk switch on.
4. Set the I/O check stop switch off.
5. Press CHECK RESET and START RESET.
6. Place the *system-control card build* section in the card reader.
7. Load the program.
   a. IBM 1402 Card Read-Punch: Press LOAD.
   b. IBM 1442 Card Reader: Press START on the reader, and PROGRAM LOAD on the console.
8. When the system attempts to read the last card,
   a. IBM 1402 Card Read-Punch: Press START.
   b. IBM 1442 Card Reader: Press START on the reader.

To use the *system-control card build* when the system is to reside on 1301:

1. Set the write-disk switch on.
2. Set write-address mode switch off.
3. Set the I/O check stop switch off.
4. Press CHECK RESET and START RESET.
5. Place the *system-control card build* section in the card reader.
6. Load the program.
   a. IBM 1402 Card Read-Punch: Press LOAD.
   b. IBM 1442 Card Reader: Press START on the reader, and PROGRAM LOAD on the console.
7. When the system attempts to read the last card,
   a. IBM 1402 Card Read-Punch: Press START.
   b. IBM 1442 Card Reader: Press START on the reader.

The time required to perform this job is approximately 5 minutes. The following halts can occur while using the *system-control card build* deck.

Halt Number (A-Address Register)	*Meaning*
008	Card-read error: To retry the operation: *For the 1402:* Nonprocess run-out the cards. Remove the last three cards in the stacker and place them in the hopper. Press START. For the *1442:* Nonprocess run-out the cards. Place the two nonprocessed cards in the read hopper. Press START on the reader and START on the console.
050	The SYSTEM BUILD control card is missing from the deck or the user entry is incorrectly punched.
051	End of job.
549	Disk unit 0 is not ready. When the system is restarted by pressing START, the disk I/O operation is retried.
554	A disk-write error occurred ten times. When the system is restarted by pressing START, the disk I/O operation is retried.

Fortran Update

To build the Fortran processor, the *Fortran update* section, made up of the sets of cards labeled FORTRAN COMPILER-LOADER-LIBRARIAN and FORTRAN STANDARD SUBPROGRAMS are used. Input for this building process is as follows:

1. IBM 1402 or 1442 *card boot,* followed by the

2. SYSTEM ASGN card, which must be punched by the user, followed by the
3. *Fortran update* section, followed by the
4. HALT card, which is the last card of the STANDARD SUBPROGRAMS.

To build the system when it is to reside on 1311:

1. Ready the pack on disk drive 0.
2. Set the I/O check-stop switch off.
3. Set the check-stop switch and disk-write switch on.
4. Set the mode switch to RUN.
5. Press CHECK RESET and START RESET.
6. Load the program.
   a. IBM 1402 Card Read-Punch: Press LOAD.
   b. IBM 1442 Card Reader: Press START on the reader, and PROGRAM LOAD on the console.
7. When the system attempts to read the last card,
   a. IBM 1402 Card Read-Punch: Press START.
   b. IBM 1442 Card Reader: Press START on the reader.

To build the system when it is to reside on 1301:

1. Set the I/O check-stop switch off.
2. Set the check-stop switch and disk-write switch on.
3. Set the mode switch to RUN.
4. Press CHECK RESET and START RESET.
5. Load the program.
   a. IBM 1402 Card Read-Punch: Press LOAD.
   b. IBM 1442 Card Reader: Press START on the reader, and PROGRAM LOAD on the console.
6. When the system attempts to read the last card,
   a. IBM 1402 Card Read-Punch: Press START.
   b. IBM 1442 Card Reader: Press START on the reader.

The time required to perform this job is approximately 20 to 30 minutes. The halts that can occur when using the *Fortran update* deck are shown in Figure 35.

Fortran Sample Program

The Fortran Sample Program is used to test the effectiveness of the system built by the user. A listing of the Sample Program is shown in *Appendix IV*. To prepare and run the Sample Program, see *Preparing a Stack* and *Running a Stack*. Figure 39 shows the input cards required to test the system by using the Sample Program.

The Sample Program consists of five separate jobs. The first job is a FORTRAN RUN. As a result of this job, a source program listing, a name dictionary, and a sequence number dictionary will be printed on the LIST file. Further, the object program in the relocatable format will be present on the LOADER file.

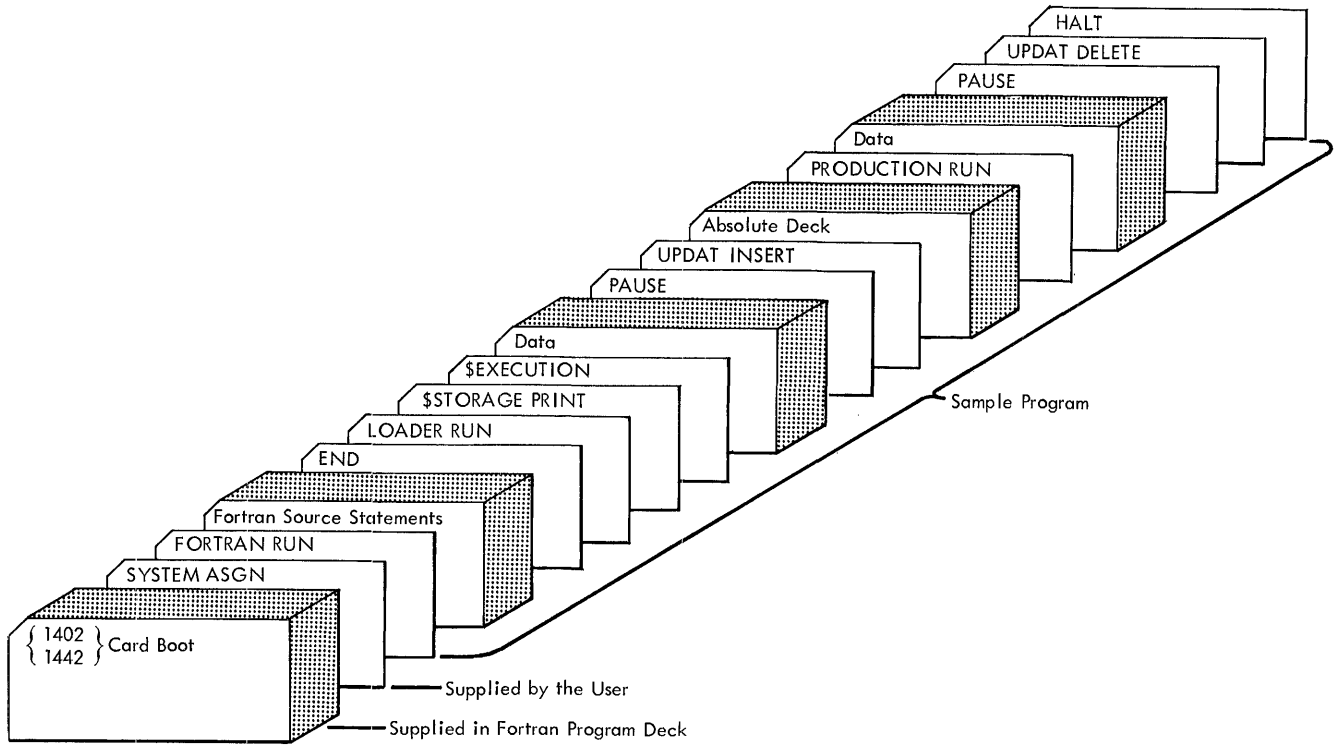


Figure 39. Fortran Sample Program

The second job is a LOADER RUN. As a result of this job, a storage print and a name map will be printed on the LIST file. Further, at the completion of processing, the object program will be executed. All necessary data cards are included.

Following the completion of execution, a temporary halt will occur, and the message

PAUSE PRESS START TO UPDATE SYSTEM WITH PHASE

will be printed on the MESSAGE file. When START is pressed, the third job will be performed.

The third job is a user-update job. The punched-card object program (phase) in the absolute format is placed on the SYSTEM file. This punched-card deck is identical to the punched-card deck that would have been produced if a loader-output-option control card ($ABSOLUTE DECK) had preceded the LOADER RUN that was previously performed.

The fourth job is a PRODUCTION RUN. As a result of this job, the phase stored on the SYSTEM file is executed. The data cards supplied are identical to the data cards used in the LOADER RUN job.

Following the completion of execution, a temporary halt will occur, and the message

PAUSE PRESS START TO DELETE SAMPLE PROGRAM FROM SYSTEM

will be printed on the MESSAGE file. When START is pressed, the fifth job will be performed.

The fifth job is a user-update. As a result of this job, the sample program (phase) is deleted from the SYSTEM file.

Updating a Fortran System

System updating is accomplished by the use of prepunched card decks supplied by IBM. All necessary control cards and data cards are included in the deck.

An update job is performed as described in *Preparing a Stack* and *Running a Stack.*

Duplicating the System Tape

To make a copy of the SYSTEM file when it resides on a tape unit, use the COPY option of the System Control Program. The content of the SYSTEM file, including the LIBRARY file if it resides on the same unit, is read into WORK1.

The required control cards for duplicating the system tape are:

1. The WORK1 file must be assigned if it differs from the assumed assignment (TAPE UNIT 4) of the System Control Program. Punch the WORK1 ASGN card in the following manner.

Columns	*Contents*
6-10	WORK1
16-19	ASGN
21-31	TAPE UNIT *n*

   The *n* represents the number of the tape unit, and can be 1, 2, 3, 4, 5, or 6.

2. The required COPY control card is punched in the following manner.

Columns	*Contents*
16-19	COPY

For a system-tape copy job, the ASGN card (if required) precedes the COPY card.

Appendix I

This section contains a summary of the processor jobs and a summary of the formats of all control cards that are recognized by the System Control Program. Also included are the compiler option control cards and output option control cards.

Each control card recognized by the System Control Program is punched in the Autocoder format, i.e., the label field is in columns 6-15, the operation field is in columns 16-20, and the operand field is in columns 21-72. The user is again reminded that blanks must appear in columns 21-72 where indicated in the individual formats. Further, entries in the label, operation, and operand fields must be left-justified in the respective fields.

Each entry in compiler option control cards and output option control cards begins in column 1. Blanks must appear where indicated in the individual formats.

Figure 40 shows a summary of a normal FORTRAN RUN job.

Figure 41 shows a summary of a normal LOADER RUN job.

Figure 42 shows a summary of a normal *user-update* job.

Figure 43 shows a summary of a normal PRODUCTION RUN job.

Figure 44, shows the formats of ASGN cards and the assumed assignments for the logical files. Figure 45 shows the valid device entries for the ASGN cards.

Figure 46 shows the formats of the remaining control cards recognized by the System Control Program.

Figure 47 shows the formats of the compiler option control cards, the compiler output option control cards, and the loader output option control cards.

NOTE: Update cards supplied by IBM are prepunched, and are included in card decks used for updating the user's system.

Input	Source program statements on the INPUT file
Output	1. Messages to the machine operator on the MESSAGE file. 2. Source - statement diagnostics on the LIST file. 3. Source program listing on the LIST file. 4. Name dictionary on the LIST file. 5. Sequence number dictionary on the LIST file. 6. Object program in the relocatable format on the LOADER file.
Optional Output	Object program in the relocatable format on the OUTPUT file. To get this option, use an OUTPUT ASGN card.
Required User Assignments	None
Required System Control Program Control Card	FORTRAN RUN
Compiler Option Control Cards	$INTEGER SIZE = nn $REAL SIZE = nn $OBJECT MACHINE SIZE = nnnnn $NO MULTIPLY DIVIDE $PHASE NAME = name
Compiler Output Option Control Cards	$NO LIST $NO NAME DICTIONARY $NO SEQUENCE NUMBER DICTIONARY $NO DICTIONARY

Figure 40. Summary of a Normal FORTRAN RUN Job

Input	Object program in the relocatable format on the LOADER file
Output	1. Messages to the machine operator on the MESSAGE file. 2. Loader diagnostic messages on the LIST file. 3. Name map on the LIST file.
Optional Output	1. Storage print on the LIST file. 2. Punched-card object program in the absolute format on the OUTPUT file.
Required User Assignments	None
Required System Control Program Control Card	LOADER RUN
Required Loader Control Card	$EXECUTION $NO EXECUTION
Loader Output Option Control Cards	$ABSOLUTE DECK [three-character file name] $STORAGE PRINT [three-character file name] $NO NAME MAP

Figure 41. Summary of a Normal LOADER RUN Job

Input	Punched-card object program in the absolute format on the CONTROL file, or object program in the absolute format on the SYSTEM file.
System Control Program Control Card	[user-comments] UPDAT three-character phase name, INSERT or [user comments] UPDAT three-character phase name, DELETE

Note: When the UPDAT INSERT card is used in a tape system, three-character phase name is the name of the phase after which the new phase is to be added.

Figure 42. Summary of a Normal User-Update Job

Input	Object program in the absolute format on the SYSTEM file
Required User Assignments	The unit(s) referenced by the object program
Required System Control Program Control Card	PRODUCTION RUN three-character phase name

Figure 43. Summary of a Normal PRODUCTION RUN Job

ASGN Card Formats			Assumed Assignment		Remarks
Label Field (Columns 6-15)	Operation Field (Columns 16-20)	Operand Field (Columns 21-72)	Compilation (FORTRAN RUN)	Execution (LOADER RUN or PRODUCTION RUN)	
SYSTEM	ASGN	{ 1311 UNIT n / 1301 UNIT 0 / TAPE UNIT n }	1311 unit: user-assigned 1301 unit: must be assigned to UNIT 0 Tape unit: user-assigned	1311 unit: user-assigned 1301 unit: must be assigned to UNIT 0 Tape unit: user-assigned	If the system residence is 1311 or 1301, the SYSTEM ASGN card is the only required ASGN card. It must follow the Card Boot in a stack of jobs. Any other SYSTEM ASGN cards in the stack are flagged and bypassed. If the user desires that the Fortran system use less than the number of core storage positions available in the processor machine, punch a comma in column 32 and 12K or 16K beginning in column 34. If the system residence is tape and the tape unit is 1, neither the Card Boot nor the SYSTEM ASGN card is required. (Pressing the TAPE LOAD key achieves the same purpose as the Card Boot and the SYSTEM ASGN card.) If the unit is 2, 3, 4, 5, or 6, both the Card Boot and the SYSTEM ASGN card are required to start systems operations.
CONTROL	ASGN	{ READER n / CONSOLE PRINTER }	READER 1	READER 1	If the CONTROL file and the INPUT file are assigned to the card reader, the assignment must be to the same card reader.
MESSAGE	ASGN	{ PRINTER n / CONSOLE PRINTER }	PRINTER 2	PRINTER 2	When the MESSAGE file is assigned to the CONSOLE PRINTER, carriage control characters used with the 1403 or 1443 printer may appear in the message. If the MESSAGE file and the LIST file are assigned to the printer, the assignment must be to the same printer. The MESSAGE file is equivalent to Fortran file 0.
LIST	ASGN	{ PRINTER n / 1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n / OMIT }	PRINTER 2	PRINTER 2	If the LIST file and the MESSAGE file are assigned to the printer, the assignment must be to the same printer. The LIST file is equivalent to Fortran file 3.
INPUT	ASGN	{ READER n / 1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n }	READER 1	READER 1	If the INPUT file and the CONTROL file are assigned to the reader, the assignment must be to the same reader. The INPUT file is equivalent to Fortran file 1.
OUTPUT	ASGN	{ PUNCH n / 1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n / OMIT }	OMIT	PUNCH 4 (1401 and 1460 systems) PUNCH 1 (1440 systems)	The OUTPUT file is equivalent to Fortran file 2.
LIBRARY	ASGN	{ 1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n }	1311 UNIT 0, START 012000, END 013900 1301 UNIT 0, START 012000, END 013900 TAPE UNIT 1	1311 UNIT 0, START 012000, END 013900 1301 UNIT 0, START 012000, END 013900 TAPE UNIT 1	1311 is assumed if the SYSTEM file is assigned to 1311. 1301 is assumed if the SYSTEM file is assigned to 1301. Tape is assumed if the SYSTEM file is assigned to tape. At execution time (LOADER RUN), the LOADER file can be assigned to READER n. If the MESSAGE, LIST, and WORK5 files are assigned to a printer, the assignment must be to the same printer. The WORK1 file is equivalent to Fortran file 4. The WORK2 file is equivalent to Fortran file 5. The WORK3 file is equivalent to Fortran file 6. The WORK4 file is equivalent to Fortran file 7. The WORK5 file is equivalent to Fortran file 8. The WORK6 file is equivalent to Fortran file 9.
LOADER	ASGN	{ 1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n / READER n / OMIT }	1311 UNIT 0, START 010400, END 012000 1301 UNIT 0, START 010400, END 012000 TAPE UNIT 3	1311 UNIT 0, START 010400, END 012000 1301 UNIT 0, START 010400, END 012000 TAPE UNIT 3	
WORK1	ASGN	{ 1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n }	1311 UNIT 0, START 007200, END 010400 1301 UNIT 0, START 007200, END 010400 TAPE UNIT 4	1311 UNIT 0, START 007200, END 007800 1301 UNIT 0, START 007200, END 007800 TAPE UNIT 4	
WORK2	ASGN	{ 1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n }	1311 UNIT 0, START 007200, END 010400 1301 UNIT 0, START 007200, END 010400 TAPE UNIT 5	1311 UNIT 0, START 007800, END 008400 1301 UNIT 0, START 007800, END 008400 TAPE UNIT 5	
WORK3	ASGN	{ 1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n }	1311 UNIT 0, START 007200, END 010400 1301 UNIT 0, START 007200, END 010400 TAPE UNIT 4	1311 UNIT 0, START 008400, END 008900 1301 UNIT 0, START 008400, END 008900 TAPE UNIT 6	
WORK4	ASGN	{ 1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n / OMIT }	OMIT	1311 UNIT 0, START 008900, END 009400 1301 UNIT 0, START 008900, END 009400 OMIT for tape systems	
WORK5	ASGN	{ 1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n / PRINTER n / OMIT }	OMIT	1311 UNIT 0, START 009400, END 009900 1301 UNIT 0, START 009400, END 009900 OMIT for tape systems	
WORK6	ASGN	{ 1311 UNIT n, START nnnnnn, END nnnnnn / 1301 UNIT n, START nnnnnn, END nnnnnn / TAPE UNIT n / OMIT }	OMIT	1311 UNIT 0, START 009900, END 010400 1301 UNIT 0, START 009900, END 010400 OMIT for tape systems	

● Figure 44. ASGN Card Formats and Assumed Assignments

Device Entry and Values of n and nnnnnn	Remarks
{1311 / 1301} UNIT n, START nnnnnn, END nnnnnn n is the number of the disk unit, and can be 0, 1, 2, 3, or 4. nnnnnn is a disk address.	The END address is the address of the next available sector. The values of nnnnnn must adhere to the following rules: 1. WORK1 and WORK2 files. If the disk unit is a 1311, the START address must be a multiple of 200. If the disk unit is a 1301, the START address must be a multiple of 800. The END address (1311 and 1301) must be a multiple of 40. 2. WORK3, WORK4, WORK5, and WORK6 files. The START address (1311 and 1301) must be a multiple of 100. The END address of each file must be a multiple of 10. In addition, WORK3 must be at least 300 sectors long. 3. LIBRARY File. For both 1311 and 1301, the START and END addresses must be multiples of 20. If these rules are violated, the system automatically narrows in the disk area to an area that does adhere to these rules.
TAPE UNIT n n is the number of the tape unit, and can be 1, 2, 3, 4, 5, or 6.	
READER n For 1402, n can be 0, 1, or 2. For 1442, n can be 1 or 2.	For 1402, n represents the pocket into which the cards are stacked. For 1442 or 1444, n represents the number of the unit.
PUNCH n For 1402, n can be 0, 4, or 8. For 1442, n can be 1 or 2. For 1444, n must be 3.	
PRINTER n n can be 1 or 2.	n represents the number of print positions available on the 1403 or 1443. For 1403, a 1 indicates 100 positions and a 2 indicates 132 positions. For 1443, a 1 indicates 120 positions and a 2 indicates 144 positions.
CONSOLE PRINTER	The console printer must be an IBM 1447 without a buffer feature.
OMIT	Select this option when the file is not to be used by the Fortran system. The LIST, OUTPUT, LOADER, WORK4, WORK5, and WORK6 files can be omitted.

● Figure 45. Valid Device Entries

Name of Card	Label Field (Columns 6-15)	Operation Field (Columns 16-20)	Operand Field (Columns 21-72)
Copy (Tape)		COPY	[Any message and/or identification]
Halt		HALT	[Any message and/or identification]
Initialize	FORTRAN	INIT	[Any message and/or comment]
Note		NOTE	Any message and/or instruction
Pause		PAUSE	Any message and/or instruction
Run	FORTRAN	RUN	
	LOADER	RUN	
	PRODUCTION	RUN	Three-character phase name
	LIBRARY	RUN	
Update *	[User comments]	UPDAT	Three-character phase name, INSERT
	[User comments]	UPDAT	Three-character phase name, DELETE
Cards associated	LIBRARY	RUN	
with **		BUILD	nnn, where nnn specifies a name-table length that differs from 030.
LIBRARY RUN			
cards		LIST	HEADERS
***	name	LIST	
		LIST	
***	name	INSER	
***	name	DELET	
****		COPY	
		END	

* When the UPDAT INSERT card is used in a tape system, three-character phase name is the name of the phase after which the new phase is to be added.

** This option applies only to libraries that reside on disk.

*** name is the six-character name of a subprogram.

**** This option applies only to libraries that reside on tape.

Figure 46. System Control Cards

Type of Control Card	Format - Columns 1 - ?	Remarks
Compiler option control cards	$INTEGER SIZE = nn	01 ≤ nn ≤ 20; assumed size is 05.
	$REAL SIZE = nn	02 ≤ nn ≤ 20; assumed size is 08.
	$OBJECT MACHINE SIZE = nnnnn	11999 ≤ nnnnn ≤ 15999; assumed size is 11999.
	$MULTIPLY DIVIDE	Multiply/divide feature assumed to be present on object machine.
	$NO MULTIPLY DIVIDE	
	$PHASE NAME = name	name is 3 alphameric characters; assumed name is ///.
Compiler output option control cards	$LIST	Source program listing is assumed.
	$NO LIST	
	$NAME DICTIONARY	Name dictionary is assumed.
	$NO NAME DICTIONARY	
	$SEQUENCE NUMBER DICTIONARY	Sequence number dictionary is assumed.
	$NO SEQUENCE NUMBER DICTIONARY	
	$DICTIONARY	Both the name and sequence number dictionaries are assumed.
	$NO DICTIONARY	
Loader output option control cards *	$ABSOLUTE DECK [three-character file name]	No absolute deck is assumed. If an absolute deck is specified, OUTPUT file is assumed.
	$NO ABSOLUTE DECK	
*	$STORAGE PRINT [three-character file name]	No storage print is assumed. If a storage print is specified, LIST file is assumed. The storage print uses a print line of 120 positions.
	$NO STORAGE PRINT	
*	$NAME MAP [three-character file name]	Name map is assumed output on the LIST file.
	$NO NAME MAP	
	$INCLUDE LDR [three-character main program name or six-character subprogram name]	1 ≤ i ≤ 6 The $INCLUDE card is punched as follows. Columns 1-8: $INCLUDE 16-18: LDR or INP or WKi 21-23: main program name or 21-26: subprogram name
	$INCLUDE INP [three-character main program name or six-character subprogram name]	
	$INCLUDE WKi [three-character main program name or six-character subprogram name]	
	$EXECUTION	One of these cards must be supplied by the user as the last card of a LOADER RUN.
	$NO EXECUTION	

* When used, [three-character file name] appears in columns 21-23 of the Loader output option control card.

Figure 47. Fortran Option Control Cards

The name, identification, and function of each phase in the Fortran system are given in the following sections.

System Control Program — Disk Resident

This section describes the phases that make up the System Control Program for disk-resident systems.

Phase Name	*ID*	*Function*
SYB	50S01	1. Determines machine size. 2. Initializes switches according to the type of reader, punch, and printer (serial or parallel). 3. Reads in the I/O package. 4. Calls the determiner.
FHW	50S11	Contains the assumed assignments for the logical files.
IOP	50S21	1. Reads or writes disk in the move or load mode. The mode depends on the processor operation. 2. Determines whether the user has exceeded specified file limits. 3. Branches to the processor phase, or branches to the end-of-file routine if the end-of-file has been sensed.
SU0	50S31	Reads in the specified phase from disk storage and branches to the specified phase.
SU1	50S41	
SU2	50S51	
SU3	50S61	
SU4	50S71	
SU5	50S81	
SU6	50S91	
OP1	50SA1	Initializes the specified area with a twenty-character control word. This control word is obtained from the temporary file-hardware table.
OP2	50SB1	
DET	50SC1	Reads the CONTROL file until a control card (HALT, PAUSE, NOTE, INIT, UPDAT, RUN, or ASGN) is sensed. When a control card is sensed, the determiner causes a halt or pauses, prints out a note, calls the update determiner, calls the selector, or calls the configurator, depending upon the type of card.
PIT	50SD1	Contains the locations of the phases in the system.
CFG	50SE1	Updates the temporary file-hardware table as specified by the ASGN card(s).
SEL	50SF1	Initializes the files used by the processor being called, and calls the first phase of that processor.
UPD	50SG1	Determines the type of update operation being performed, and calls in that particular updater.
UIN	50SH1	Places a new phase on the SYSTEM file in any available location.
UHD	50SI1	Updates the header of a phase that is in the SYSTEM file, as specified by a header card.
UDL	50SJ1	Deletes a phase from the SYSTEM file.
UPT	50SK1	Patches a part of a phase on the SYSTEM file.
DMP	50SL1	Prints storage on the LIST file.
DM2	50SM1	
F/P	50SN1	Prints all WORK files on the LIST file.
F/2	50SO1	
F/3	50SP1	
UPK	50SQ1	Compresses phases within the system area so that all unused sectors are made available.
MNE	AUMNE	These phases are used by the Autocoder Assembler Program.
2XB	EX2XB	
4XB	EX4XB	
6XB	EX6XB	
8XB	EX8XB	

System Control Program — Tape Resident

This section describes the phases that make up the System Control Program for tape-resident systems.

Phase Name	*ID*	*Function*
SYB	50S01	1. Determines machine size. 2. Initializes switches according to the type of reader, punch, and printer (serial or parallel). 3. Reads in the I/O package. 4. Calls the determiner.
IOP	51T05	1. Contains the assumed assignments for the logical files. 2. Reads or writes tape in the move or load mode. The mode depends on the processor operation.

Phase Name	*ID*	*Function*
		3. Branches to the processor phase, or branches to the end-of-file routine if the end-of-file has been sensed. 4. Reads in the specified phase from the system tape and branches to the specified phase. 5. Initializes the specified area with a twenty-character control word. This control word is obtained from assumed assignments for logical files.
DET	51T06	1. Reads the CONTROL file until a control card (HALT, PAUSE, COPY, NOTE, INIT, UPDAT, RUN, or ASGN) is sensed. When a control card is sensed, the determiner causes a halt, or pauses, or prints out a note. 2. Updates assumed file assignments as specified by the ASGN card(s). 3. Initializes the files used by the processor being called, and calls the first phase of that processor. 4. Determines the type of update operation being performed, and calls the updater.
UPD	50SH1	1. Places a new phase on the SYSTEM file in any available location. 2. Updates the header of a phase that is in the SYSTEM file, as specified by a header card. 3. Deletes a phase from the SYSTEM file. 4. Patches a part of a phase on the SYSTEM file.
DMP DM2	50SL1 50SM1	Prints storage on the LIST file.
F/P	51T10	Prints all WORK files on the LIST file.

Fortran Processor Program

This section describes the phases that make up the Fortran Processor Program.

Phase Name	*ID*	*Function*
00F	00FIV	1. Reads compiler option control cards, if any, from the INPUT file and outputs them on the LIST file. 2. Initializes deblocking routines (GETEX and PUTEX).
		3. Sets up an area in upper core storage for GETEX and PUTEX buffers. 4. Passes the first source-program statement to phase 10F.
10F	10FIV	1. Reads source statements from INPUT file until the END card is sensed. 2. Assigns sequence numbers to all source statements except comments statements. 3. Outputs the source program listing on the LIST file, if the option is exercised. 4. Replaces key words (COMMON, DIMENSION, GO TO, etc.) with internal three-character symbols. Replaces remaining portion of each statement with internal symbols. 5. Replaces unrecognizable statements with diagnostic codes. 6. Outputs nonexecutable and I/O name lists on WORK2 and executable statements on WORK1. 7. Calls phase 20F, 25F, or 30F, depending on type of source statements.
20F	20FIV	1. Extracts DIMENSION, COMMON, EQUIVALENCE, FUNCTION, SUBROUTINE, EXTERNAL, and type statements from WORK2. 2. Builds a name-attribute table in upper core storage.
21F	21FIV	1. Uses name-attribute table built by phase 20F and allocates object-time storage for COMMON variables and applicable EQUIVALENCE definitions. 2. Allocates storage for normal variables with EQUIVALENCE or DIMENSION definitions, and adds this information to the name-attribute table. No storage is allocated unless the complete set of variable attributes has been determined. 3. Compresses the name-attribute table, deleting information no longer required. 4. Outputs a macro reflecting ordering or various variable types within COMMON on the PUTEX file. 5. Calls phase 25F, if required. Otherwise, phase 30F is called.

Phase Name	*ID*	*Function*
25F	25FIV	1. Extracts FORMAT, DATA, and DEFINE FILE statements and I/O name lists from WORK2. 2. Replaces names in DATA or I/O name lists and source or generated constants from I/O name lists with object-time addresses. 3. Generates sequence of macros with associated parameters on the PUTEX file for each statement. An increased object-time character count is included with each sequence number or label assignment macro reflecting the total number of object-time characters since the last sequence number or label assignment macro. 4. Begins label table for FORMAT statement numbers. 5. Continues building name-attribute table and continues allocating storage as required.
30F	30FIV	1. Extracts executable statements from the GETEX file, which can contain macros from phase 25F. 2. Continues building name-attribute table and label table, with allocation accomplished when required. An address is substituted for a name and an internal representation is substituted for a label. 3. Generates macros with appropriate parameters for all executable statements, except for expressions that are passed to phase 40F. Subscripted variables are processed as in phase 25F, generating macros even when they are part of an expression. The object-time incremental character count is included with sequence number or label assignment macros. 4. If source label and name table overflows, subsequent source labels and/or names are passed in expanded form for replacement by phase 34F.
33F	33FIV	Processes subscripted variables.
34F	34FIV	Phase 34F is an optional phase. When required, it completes the storage allocation for normal variables. It replaces names with addresses, and source labels with internal label notation.
35F	35FIV	1. Outputs a name dictionary on the LIST file, if requested. 2. Outputs constants as macros in the name table on the PUTEX file. 3. Allocates storage in upper core storage for a source label table and a generated label table which will eventually contain actual addresses. 4. Calls phase 36F if compiling a subprogram.
36F	36FIV	When a subprogram is being compiled, macros are generated on the PUTEX file to represent the necessary processing when the subprogram is called. A "prologue" is generated; also, an "epilogue" is generated which represents any necessary resetting of values before returning control to the calling program. If variable dimensions have been used, a pass on the GETEX file is required when building the prologue name to indicate additional object-time calculations.
40F	40FIV	1. Extracts expressions from the GETEX file that were partially processed by phase 30F. The expressions are reordered according to implied operator precedence and parentheses, and macros are generated on the PUTEX file. 2. Extracts label assignments and sequence number macros from the GETEX file. For a sequence number, the incremental character count is replaced with an actual accumulative character count, and the macro is passed. Label assignment macros are not passed further. Actual addresses for source labels and generated labels are entered into the label table and generated label table. 3. Passes control to phase 45F if a DATA statement is sensed.
45F	45FIV	1. Extracts DATA name list macro sequences from the GETEX file. The macros are expanded into object code in a phase work area. The code references some included subroutines. DATA literal list macros are matched with object-time addresses and passed as regular literal macros. 2. Continues label assignments and sequence number processing as in phase 40F.
53F	53FIV	1. This phase is called only if diagnostic codes were output by a previous phase. The

Phase Name	*ID*	*Function*
		codes are extracted and translated into diagnostic messages that are output on the LIST file. These errors can be warnings, or severe errors that would prevent a successful compilation. 2. If the messages were merely warnings, phase 70F is called. 3. If the messages indicated errors that would prevent a successful compilation, the system halts, then control reverts to the DET (determiner) phase of the System Control Program, which will read a control card from the CONTROL file.
70F	70FIV	Generates object code in the relocatable format from internal macros and associated parameters. Addresses are substituted for label references. A header card image is always generated first, and a trailer card image is always generated as the last card. Between these two card images will be relocatable and external name cards to indicate object characters to be loaded and interprogram references. Card images appear on the LOADER and/or OUTPUT files, depending on actual device assignments. Control is returned to the DET (determiner) phase of the System Control Program.
78F	78FIV	Builds, updates, or lists the Fortran relocatable subprogram library.
79F	79FIV	1. Reads loader control cards from the CONTROL file. 2. Relocates and loads a main program and required subprograms, possibly from the subprogram library. 3. Establishes interprogram communication by replacing external references by actual addresses.
80F	80FIV	Produces an external name map on the LIST file, if requested.
81F	81FIV	Produces storage print of the loaded program on the LIST file, if requested. The storage print does not include the standard overlay package.
82F	82FIV	Produces an absolute deck on the OUTPUT file, if requested.
90F	90FIV	Standard overlay package for 1401 or 1460, including arithmetic interpreter and standard I/O routines.
91F	91FIV	Standard overlay package for 1440, including arithmetic interpreter and standard I/O routines.

Building a System that Contains Fortran and Autocoder

In this section, the Autocoder system refers to *1401/1440/1460 Autocoder (on Disk)*, program number 1401-AU-008. The specifications and operating procedures for this program are contained in the Systems Reference Library publications, *Autocoder (on Disk) Language Specifications for IBM 1401, 1440, and 1460*, Form C24-3258 and *Autocoder (on Disk) Program Specifications and Operating Procedures for IBM 1401, 1440, and 1460*, Form C24-3259.

File Considerations

Because the System Control Program is the controlling element of the Autocoder system as well as the Fortran system, it is possible to build a SYSTEM file that contains both the Fortran Processor Program and the Autocoder Assembler Program.

Figure 48 shows the disk-storage allocation on the system unit when both Fortran and Autocoder are present.

The user should consult the referenced Autocoder publications for a description of the Autocoder system.

Two differences exist when the Autocoder system resides alone on a disk unit as opposed to when it resides on a disk unit that also contains the Fortran system. These differences are:

1. The assumed assignment of the Autocoder LIBRARY file.
2. The assumed assignments of the Autocoder WORK files.

Figure 49 gives the assumed assignments of the Autocoder LIBRARY file and the Autocoder WORK files when Autocoder and Fortran reside on the same SYSTEM file. Do *not* consider the assumed assignments for these files as given in the Autocoder operating procedures publication.

Note also that Autocoder uses a maximum of three WORK files, whereas six WORK files are defined for Fortran. Further, the user should note that Autocoder defines a CORELOAD file (not applicable to Fortran), and Fortran defines a LOADER file (not applicable to Autocoder). The assumed assignments for the remaining logical files (CONTROL, MESSAGE, LIST, INPUT, and OUTPUT) are the same for both the Autocoder and the Fortran systems.

File	Mode	File-Protected	Sector Range
SYSTEM File			
Autocoder Preprocessor Work Area	Move	No	000000-000089
Autocoder Preprocessor	Move	No	000090-000199
Autocoder Preprocessor	Load	No	000200-000259
Autocoder Preprocessor	Move	No	000260-000299
Autocoder Preprocessor	Load	No	000300-000899
Not Used	Loadq	No	000900-002499
System Control Program	Load	Yes	002500-002904
Fortran Processor Program	Load	Yes	002905-002979
Not Used	Load	Yes	002980-002999
System Control Program	Load	Yes	003000-003175
Fortran Processor Program Autocoder Assembler Program Area for User's Fortran Object Program Library	Load	Yes	003176-007199
WORK Files	Move	No	007200-010399
LOADER File (Fortran only. This area is used by Autocoder as a continuation of the WORK files.)	Move	No	010400-011999
LIBRARY File (Fortran)	Move	No	012000-013899
LIBRARY File (Autocoder)	Move	No	013900-019979

● Figure 48. Disk Storage Allocation

File	Assumed Assignment
LIBRARY	1311 UNIT 0, START 013900, END 019980 1301 UNIT 0, START 013900, END 019980
WORK1	1311 UNIT 0, START 007200, END 010400 1301 UNIT 0, START 007206, END 010400
WORK2	1311 UNIT 0, START 010400, END 011600 1301 UNIT 0, START 010400, END 011600
WORK3	1311 UNIT 0, START 011600, END 012000 1301 UNIT 0, START 011600, END 012000

● Figure 49. Assumed Assignments of Autocoder LIBRARY and WORK Files

Building a Combined System

To build a system comprising Fortran and Autocoder, first build a Fortran system as described in this publication.

After the Fortran system has been built, use the Autocoder marking program to separate the various components of the Autocoder program deck as described in *Autocoder (on Disk) Program Specifications and Operating Procedures for IBM 1401, 1440, and 1460*, Form C24-3259. Use only the *Autocoder-update* section of the Autocoder program deck to build the Autocoder system. This section is made up of the sets of cards labeled AUTOCODER PROCESSOR, AUTOCODER PREPROCESSOR, and AUTOCODER MACROS. After the system has been built, use the appropriate SAMPLE PROGRAM to test the system.

Perform a user-update job to add the Autocoder Assembler Program to the SYSTEM file. Input for this job is as follows.

1. The 1402 or 1442 *card boot*, followed by
2. The SYSTEM ASGN card, which must be punched by the user, followed by
3. The *Autocoder update* section, made up of the sets of cards labeled AUTOCODER PROCESSOR, AUTOCODER PREPROCESSOR, and AUTOCODER MACROS, followed by
4. The HALT card, which must be punched by the user.

To build the system when it is to reside on 1311:

1. Ready the Fortran system pack on disk drive 0.
2. Set the I/O check-stop switch off.
3. Set the check-stop switch and the disk-write switch on.
4. Set the mode switch to RUN.
5. Press CHECK RESET and START RESET.
6. Load the program.
   a. IBM 1402 Card Read-Punch: Press LOAD.
   b. IBM 1442 Card Reader: Press START on the reader, and PROGRAM LOAD on the console.
7. When the system attempts to read the last card,
   a. IBM 1402 Card Read-Punch: Press START.
   b. IBM 1442 Card Reader: Press START on the reader.

To build the system when it is to reside on 1301:

1. Set the I/O check-stop switch off.
2. Set the check-stop switch and disk-write switch on.
3. Set the mode switch to RUN.
4. Press CHECK RESET and START RESET.
5. Load the program.
   a. IBM 1402 Card Read-Punch: Press LOAD.
   b. IBM 1442 Card Reader: Press START on the reader, and PROGRAM LOAD on the console.
6. When the system attempts to read the last card,
   a. IBM 1402 Card Read-Punch: Press START.
   b. IBM 1442 Card Reader: Press START on the reader.

The halts that can occur when using the *Autocoder-update* section are shown in Figure 35.

Testing the Autocoder System

The appropriate SAMPLE PROGRAM deck, which is used to test the effectiveness of the system built by the user, calculates and lists a table of salaries. A listing of the sample program is shown in an appendix of the Autocoder operating procedures publication.

The first card in the sample program is a partially prepunched control card used for assigning the CORELOAD file. This partially prepunched control card cannot be used when the Autocoder and Fortran systems reside on the same unit because the assignment of the CORELOAD file designated in the control card is within the area defined for the Fortran LIBRARY file. Discard this control card and replace it with a new control card punched in the following format.

Columns	*Contents*
6-13	CORELOAD
16-19	ASGN
21-24	1311 *or* 1301
26-57	UNIT 0, START 000040, END 000090

To prepare and run the sample program, see *Preparing a Stack* and *Running a Stack.*

```
                              FORTRAN   RUN                                                    77KTST00
$NO MULTIPLY DIVIDE                                                                  77L
      C      SAMPLE PROGRAM TO TEST SYSTEM                                              77M SAMP
      C                                                                                 77N SAMP
      C      PROGRAM FOR FINDING THE LARGEST VALUE                                      77O SAMP
      C        ATTAINED BY A SET OF NUMBERS                                             77P SAMP
001           DIMENSION A%12¤                                                          77Q
002           READ %1,1¤N,%A%I¤,I#1,N¤                                                 77R SAMP
003         1 FORMAT %I3/%12F6.2¤¤                                                     78I SAMP
004           BIGA#A%1¤                                                                78J SAMP
005           DO 20 I#2,N                                                              78K SAMP
006           IF %BIGA.LT.A%I¤¤ BIGA # A%I¤                                            78L SAMP
007        20 CONTINUE                                                                 78M SAMP
008           WRITE %3,2¤N,BIGA                                                        78N SAMP
009         2 FORMAT %21H1THE LARGEST OF THESE,I3,11H NUMBERS IS,F9.2¤                 78O SAMP
010           STOP                                                                     78P SAMP
011           END                                                                      78Q SAMP
```

```
NAME DICTIONARY
  00126 A        00131 N        00136 I        00149 BIGA

 SEQUENCE NUMBER DICTIONARY

   002-00151   004-00170   005-00182   006-00205   007-00249   000-00249   008-00253   010-00272

   011-00282   001-00292   002-00292   003-00355   008-00408   009-00423   011-00506
```

```
                              LOADER    RUN                                                  78R SAMP
$STORAGE PRINT                                                                       79I
```

```
                                        *** NAME MAP ***

05851  ///      06515  ¤C      06978  ¤D      06228  ¤A      06415  ¤B      07155  ,9
08004  *1       06982  ¤F      07050  ¤G      08025  *3      06206  LINK    07102  ¤H
07405  G.       07676  P.      07906  Q.      07484  H.      07523  M.      06511  ¤E
07155  ,0       06441  Z¤1     06320  Z¤      06482  Y¤      06375  U¤      06416  V¤
07207  %0       07219  %1      07271  %2      07291  %3      07251  %4      07231  %5
07377  G.3      07404  G.1     07341  G.2

                                    *** END OF NAME MAP ***
```

◐ Figure 50. Sample Program (Part 1 of 3)

```
                                          *** STORAGE PRINT ***

05700 X0‡  01??0?
           1111
05800 Y0‡                                         1?A            2BL2900M1M X0‡0!5VZ9S B‡45Y2W>Y4 Z‡BR8SB?5‡ Y3WY5‡Y3/X
                                                  1 1            11   1  11 1  11  1   1   1  1    11   1    1  1  1  1
05900 Z0‡ 0SZ5TB‡45Y 4Z-$Y3WY3Y $Y3W5Z4Z‡B ‡45$Y3WY3Y $Y3W>Y4Z‡B Y8WBL2902N 2MX0‡0J2TJ 0YBK0WZ8/D ETBK0WZ9/D ETBR75YC/%
            1   1  1    111 1   11  1   11    11  1   11  1   11    1   1   111 11   1   1   1  1   1   1  1   1   1  1
06000 !0‡ B!2UBR75$Y 3WY3Y$YTW% BA0SBR8SB? 5‡Y3WY3ZY3 /X0S!5/B!0 UB?93BD0V0 01BF7W0010 03IBI0W/BD 0V001BF7W0 12006F002B
          1   1   11   1  11  1 1   1   1    1  1  1   1  1  1    1   1   1   1   1  1   1 1   11   1  1   1   1  11  1
06100 J0‡ D8UBI0W BR 75YC/YUZ%B ?93BD0V001 BE2T0211TH E LARGEST  OF THESEBF 7W001003IB E2T011 NUM BERS ISBF7 W001009F00
             1   11    1  1  11    1   1   1   1  1                      1    1  1  11    1  1           1    1  1  11
06200 K0‡ 2BI0W H089 M0‡2K2‡MK2 T835B746SM 8SM0!6K5WM 0!3K4Z?000 M8ZL000M9U BN1/9601@M 8ZN0‡AN0‡M 8SCM7ZM8UB M5ZTBL1T0!
           1   11    1      1       1   1    1      1       1       1       1       1       1       1      1     1
06300 L0‡ 7$H0940!6B K3SHL7VL7W SN0VM8SBL3 SZAN0XM7ZB L3SZYM7YM8 S?M8‡090YL XYM8‡MM8S0 89B000#0J0 089Y0!9088 H0940!7BM1
            1      1    1       1      1     1      1             1              1       1   1       1       1      1
06400 M0‡ W0!4,H0940 !4B000M0!7 M2ZM089000 H/020!4BX4 7M0!7M5U#0 89000BM3‡B Y76SSEN!02 .M5Z2SKB      15
               1       1   1       1       1      1    1      1       1   1    1  1    1   1   1     1 1     1
06500 N0‡  160009F    HR7YHR4/0 ?0HN8/HR4Y 0!0HR5V0‡0 MR7Y099,0? 10?4M0?3N8 YM0?6N9V¤0 ?10?4H0990 00H00S000H 00Z000H089
           1    1     1   1       1   1       1       1      1       1       1      1       1       1      1       1
06600 00‡ 000H094000 MR7Y099HR5 Z0?7?0‡0R8 ‡YR7Z0‡0BP 6W0?0%C0‡0 0!0H099,0? 0Q099S0!0¤ 0?1HP6/0!0 ?0?0R8/YR7 Z0!0YR7Z0?
             1       1      1       1       1      1        1       1   1    1   1   1    1       1      1       1
06700 P0‡ 0BP2X0?00A 0‡00!0SR7Z 0?0B09UVP5 V0?01D0!0Q 094D0?0Q09 9B09UH0940 00BR0/D0!0 Q099C0?00‡ 0Q094,0!1V Q1V0?12?0?
           1       1       1       1   1        1   1    1   1    1   1       1   1    1   1       1   1   1        1
06800 Q0‡ 1R8/YR7Z0? 1,Q5ZS0‡00 ?1VQ4/0?1K AR7Z0!0BQ1 VA0‡00?1YR 7Z0?1BQ8WN H0990?1H09 40!1¤0!0,0 !1BP8Z¤Q5Z YR8/0?1¤0!
              1       1   1       1        1      1    1     1        1     1       1      1   1    1   1    1      1
06900 R0‡ 1VR6‡R8‡KV R6YR8/K?0! 0YR5WR1XYR 8‡0‡0H0990 00H094000H 089000B000 VR1XR8/K!R 1XBR1X    1   H094H094 0!4H?4/0?0
           1       1       1     1      1       1       1      1       1    1       1    1   1  1  111  1       1
07000 ?0‡ M956099M0! 5?1X?000AC UM0!2?3ULA CU000H0990 00,A0/B?5Y H094¤A0/M0 !2A0YM0J1M 0!8A1SM0!2 A1ZM0J4A2W HA3U0J5B0J
          1      1       1       1      1       1       1   1    1   1   1       1   1       1       1       1      1
07100 A0‡ 5NA000000? 000A5US000 A5UV000A5U KB000                          BB0X0 !40BB1Z0!4 1BB7/0!42B B9/0!43BB5 /0!44BB3/0
           11      1       1       1        1   1                     1        1        1       1        1        1
07200 B0‡ !45.900BC1 /222 BC2SB C2S222 BC1 /BC1/222 V C1/2G9BBC2 SBC1/222 V C1/2G9KBC2 SBC2S222 V C1/2G9BBC2 SBC2S222 V
             1   1        1   1        1    1       1        1    1       1        1    1       1        1    1       1
07300 C0‡ C1/2G9KBC2 SM0!7094B0 !0B0!90!8‡ H/020!48U6 3HD8S000HD 7Z000MD8SD 7WMMMMHK90 000B059MD5 YK90MD5VK9 6PD5ZD5TBB
                 1    1      1    1        1      1    1      1       1       11111       1   1       1       1       1
07400 D0‡ 18   MD7WD 8SMMMMMK90 D5YMK96M0! 6K90H0940! 7HK96HD0UB B18                                       ‡H094#K 20K90CK03K
            1  1       11111       1   1       1       1   1   1    1  1  1      1      1     11   1       1
07500 E0‡ 90BC7XSBB1 8TMK96094B B18M0!6K81 ,F3SM089K8 4#K20K81AK 81K84CL02K 84BF4‡UH09 40!7AK8108 9#K81094BF 0XL161M0!0
            1    1     1      1    1       1   1       1      1       1       1    1       1       1      1        1
07600 F0‡ 1T2BF1UM1T 20!0H0940! 1H0890‡1BB 18K24*B000 ML02K84S08 9K84MK84K8 1BQ18E06¤F 3SBE6XB000 K99#M0J0L1 2,042045M0
             1   1       1       1      1        1    1      1       1       1   1  1    1   1        1       1       1
07700 G0‡ J3047M0!90 44HK740?0H K710!0VG4V L281M0!6K6 2,L28M0440 94A089094# K20094CL02 094BH9ZUHK 680!1HK650 !0DK17001Y
               1       1      1       1        1       1   1       1      1       1       1    1       1       1      1
07800 H0‡ K98001YK35 H4THH9YC41 BH5/L12LBH 6ZL12IBH8‡ 001 BQ06E0 3YK34H4THH 9Y003BH3WY K36H4TBH3W YK17K98B00 3L12ABH9YB
                1       1       1       1        1        1  1    1      1       1   1       1    1      1        1   1
07900 I0‡ Q06E04BI4Z 0!4/BI3XK9 9#VI4VK751 BQ18E05,K7 5B000¤K75H 003I6UV003 L17SBI9‡0! 4/MD0UL24# 00LL24BC4/ HL240!5B05
             1  1        1        1        1   1  1    1   1   1       1        1        1      1       1    1      1
08000 00! 9I9C1INP                   3LST
           1   11                    11                     1

                                          *** END OF STORAGE PRINT ***
```

```
$EXECUTION
```

```
THE LARGEST OF THESE 12 NUMBERS IS  9876.54
```

● Figure 50. Sample Program (Part 2 of 3)

```
                    PAUSE  PRESS START TO UPDATE SYSTEM WITH PHASE
```

```
                    UPDAT///,INSER                                  001  ///
```

```
                PRODUCTIONRUN  ///
```

```
THE LARGEST OF THESE 12 NUMBERS IS  9876.54
                    PAUSE  PRESS START TO DELETE SAMPLE PROGRAM FROM SYSTEM
```

```
                FORTRAN   UPDAT///,DELETE
                          HALT  SAMPLE PROGRAM COMPLETE, ALL SYSTEMS ARE GO.
```

Figure 50. Sample Program (Part 3 of 3)

Index

IBM **Technical Newsletter**

File Number GENL-25

Re:Form No. C24-3322-2

This Newsletter No. N21-5051

Date December 15, 1966

Previous Newsletter Nos. None

This Technical Newsletter supplies replacement pages for FORTRAN IV Language Specifications, Program Specifications, and Operating Procedures; IBM 1401, 1440, and 1460.

Each page on which changes have been made is coded in the upper outside corner. Text changes are indicated by a vertical line at the left of the change, figure changes by a dot at the left of the figure title.

REPLACE these pages:

3-4
13-14
15-16
51-52
53-54
55-56
57-58

ADD these pages:

97.1-97.2

Keep this page as a record of changes.

International Business Machines Corp., Programming Publications Dept. Rochester, Minn. 55901

Printed in U.S.A.

N21-5051 (C24-3322-2)

Contents

Form C24-3322-2
Page Revised 12/15/66
By TNL N21-5051

In the example, if K is equal to 3 at the time of execution, the program will transfer control to the third statement in the list, statement 50.

The Logical IF Statement

General Form. IF *(t) s*

t is a logical expression.

s is any executable statement except DO, an arithmetic IF, or another logical IF.

If the logical expression *t* is true, statement *s* is executed. Control then transfers to the next sequential executable statement, unless *s* is a GO TO statement, in which case control is transferred as indicated.

If *t* is false, control transfers to the next sequential statement.

If *t* is true and *s* is a CALL statement, control transfers to the next sequential executable statement on return from the subprogram called.

Examples.

```
IF (A .AND. B) F = SIN (R)
IF (16 .GT. L) GO TO 24
IF (D .OR. X .LE. Y) GO TO (18, 20), I
IF (Q) CALL SUB (Q)
```

The Arithmetic IF Statement

General Form. IF *(a)* n_1, n_2, n_3

a is an arithmetic expression, type integer or type real.

n_1, n_2, n_3 are statement numbers of executable statements.

The arithmetic IF statement is used to transfer control to one of three specified statements depending on the value of an arithmetic expression. The arithmetic statement *a* is tested. If *a* is less than zero, control transfers to statement n_1. If *a* is equal to zero, (plus or minus), control transfers to statement n_2. If *a* is greater than zero, control transfers to n_3.

Examples.

```
IF (A(J, K) − B) 10, 4, 30
IF (D*E + BRN) 9, 9, 15
```

The DO Statement

General Form. DO *n* $i = m_1,m_2,m_3$

n is a statement number of an executable statement.

i is a nonsubscripted integer variable.

m_1,m_2,m_3 are either unsigned integer constants greater than zero or unsigned nonsubscripted integer variables whose value is greater than zero. The number of digits in the integer size (*k*) must be at least one greater than the greatest number of digits used for m_1,m_2, or m_3. In the example below, *k* must be greater than or equal to 3.

m_3 is optional. If m_3 is not stated, its value is assumed to be 1. If it is omitted, the preceding comma must also be omitted.

Examples.

```
DO 30 I = 1, M, 2
DO 24 I = 1, 10
```

The DO statement is an instruction to execute repeatedly the statements that follow, up to and including the statement numbered *n*. The first time the statements are executed, *i* has the value m_1 and each succeeding time *i* is increased by the value of m_3. After the statements have been executed with *i* equal to the highest value that does not exceed m_2, control passes to the executable statement following statement number *n*. This is called the normal exit from the DO statement.

The Range of the DO Statement. The range of the DO statement is that set of statements that will be executed repeatedly. That is, it is the sequence of statements immediately following the DO statement, up to and including the statement numbered *n*. After the last execution in the range, the DO is said to be satisfied.

The Index of the DO Statement. The index of the DO statement is the integer variable *i*. Throughout the range of the DO, the index is available for computation, either as an ordinary integer or as the variable of a subscript. After a normal exit from a DO, the index *i* must be redefined before it is used in computation. After exiting from a DO by transferring out of the range of the DO, the index *i* is available for computation and is equal to the last value it attained.

DO's Within DO's. A DO statement can be contained within another DO statement. This is called a nest of DO's. If the range of a DO contains another DO, then all statements in the range of the enclosed DO must be within the range of the enclosing DO. The maximum depth of nesting, not including implied DO's in I/O lists, is twelve. That is, a DO can contain a second DO, the second can contain a third, the third can contain a fourth, and so on up to twelve statements.

Transfer of Control. Control cannot be transferred into the range of a DO from outside its range. However, control can be transferred out of a DO range. In this case, the value of the index remains available for use. If the exit is transferred out of the range of a set of nested DO's, then the index of each DO is available.

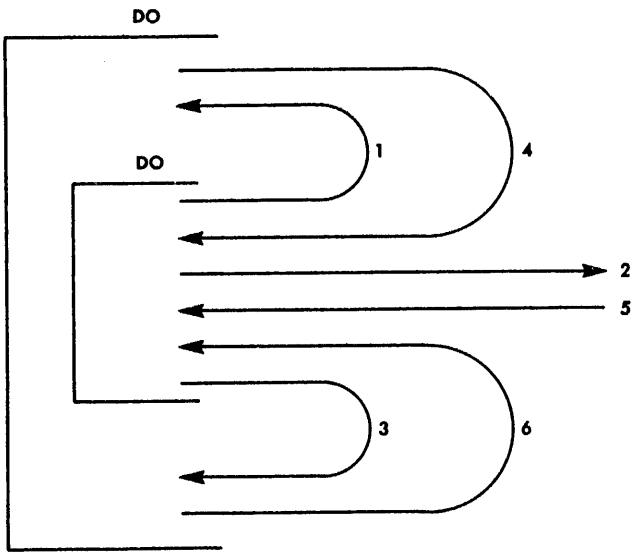


Figure 5. Nest of DO's

Figure 5 illustrates the possible transfers in an out of the range of a DO. In the figure, 1, 2, and 3 are permitted, but 4, 5, and 6 are not permitted.

Restrictions on Statements in the Range of a DO. Any statement that redefines the index or any of the indexing parameters (*m*'s) is not permitted in the range of a DO.

The range of a DO cannot end with a GO TO type statement or another DO. The range of a DO can end with a logical IF, in which case control is handled in the following manner. If the logical expression *t* is false, the DO is repeated. If the logical expression *t* is true, statement *s* is executed and then the DO is repeated. If *t* is true and *s* is a transfer type statement, control is transferred as indicated by *s*.

When a reference to a subprogram is made in the range of a DO, care must be taken that the called subprogram does not alter the index or any of the indexing parameters.

The CONTINUE Statement

General Form. CONTINUE

CONTINUE is a dummy statement that does not produce any executable instructions. It is most frequently used as the last statement in the range of a DO to produce a branch address for GO TO statements that are intended to begin another repetition of the DO range.

Example.

```
      .
      .
      .
      DO 20 I = 2, N
      IF (BIGA .LT. A (I) )BIGA = A (I)
   20 CONTINUE
      .
      .
      .
```

The PAUSE Statement

General Form. PAUSE or PAUSE *n*

n is an unsigned integer constant of one to three digits.

The statement causes the machine to halt. The integer constant *n* is in the B-address register. If *n* is not specified, it is assumed to be zero. When the machine is restarted by pressing the start key, the next Fortran statement is executed.

The STOP Statement

General Form. STOP

The STOP statement terminates execution of the program. When the STOP statement is executed, control returns to the System Control Program. The program can have any number of STOP statements.

The END Statement

General Form. END

The END statement defines the end of a program or a subprogram. Physically, it must be the last statement of each program or subprogram. As the END statement is not executable, it must not be encountered in the flow of the program.

The Specification Statements

The specifications statements provide information about storage allocation and the variables and constants used in the program.

The DIMENSION Statement

General Form. DIMENSION v_1 (i_1), v_2 (i_2), . . . , v_n (i_n)

$v_1, v_2, \ldots, v_n$ are the names of arrays.

$i_1, i_2, \ldots, i_n$ are each composed of 1, 2, or 3 unsigned integer constants separated by commas. Each integer specifies the maximum value of the subscript. i can be an integer variable only when the DIMENSION statement appears in the subprogram.

Examples.

```
DIMENSION A(10), B(5, 15)
DIMENSION S(10), K(5,5,5), G(100)
```

The DIMENSION statement provides the information necessary to allocate storage for arrays in the object program. It defines the dimensionality and the maximum size of each array listed.

Each variable that appears in subscripted form in the source program must appear in a DIMENSION statement contained in the source program. However, if the dimension information for a variable is included in a COMMON statement in the source program, it must not be included in a DIMENSION statement.

A single DIMENSION statement can specify the dimensions of any number of arrays. The DIMENSION statement that specifies the array size must precede the first appearance of each subscripted variable in an executable or DATA statement. (See Figure 6).

Dimensions specified in a COMMON statement are subject to all the rules for the DIMENSION statement.

```
DIMENSION A(5, 10)
      .
      .
      .
CALL MAYMY (...,A,...)
      .
      .
      .
SUBROUTINE MAYMY (...,R,...)
      .
      .
      .
DIMENSION...,R(5, 10),...
      .
      .
      .
```

Figure 6. Passing Array Names

The COMMON Statement

General Form. COMMON *a, b, c, . . .*

a, b, c, . . . are variable or array names that can be dimensioned

Example.

```
COMMON A, B, C (5, 10)
```

The COMMON statement refers to a common area in core storage. Variables or arrays that appear in main programs and subprograms can be made to share the same storage locations by using the COMMON statement. For example, if one program has the statement COMMON A and a second program has the statement COMMON B, the variables (or arrays) of A and B will occupy the same storage locations in the COMMON area. These variables (or arrays) appearing in COMMON statements are assigned locations relative to the beginning of the COMMON area.

Within a specific program or subprogram, variables and arrays are assigned core storage locations from the high core-storage addresses to lower core-storage addresses in the sequence in which their names appear in the COMMON statement. Subsequent sequential storage assignments within the same program or subprogram are made with additional COMMON statements.

For example, if the main program contains the statement

```
COMMON A, B, C
```

and a subprogram contains the statement

```
COMMON L, M, N
```

then A, B, and C are assigned sequential locations, as are L, M, and N. Further, A and L will occupy the same location, B and M will occupy the same location, and C and N will occupy the same location.

Variables declared in COMMON must agree, respectively, in type. In the preceding example, A and L are type real, as are B and M, and C and N. (L, M, and N must be declared as real in the subprogram.)

A dummy variable can be used in a COMMON statement to establish shared locations of variables that would otherwise occupy different locations. For example, the variable X can be assigned to the same location as the variable C of the previous example by using the following statement.

```
COMMON R, S, X
```

where R and S are dummy names that are not used elsewhere in the program.

Redundant entries are not permitted in a COMMON statement. For example, the following statement is invalid.

```
COMMON F, G, H, F
```

Variables brought into COMMON through EQUIVALENCE statements may increase the size of COMMON (see *The EQUIVALENCE Statement*).

Two variables in COMMON cannot be made equivalent to each other, either directly or indirectly.

The EQUIVALENCE Statement

General Form. EQUIVALENCE (*a, b, c,...*), (*d, e, f,...*),...

a, b, c, d, e, f,..., are variables which may be subscripted. Subscripted variables can have single or multiple subscripts. These subscripts must be integer constants.

Example.

```
EQUIVALENCE (A, B(1), C(5)), (D(17), E(3))
```

The EQUIVALENCE statement controls the allocation of core storage by causing two or more variables to share the same core storage location.

An EQUIVALENCE statement can be placed anywhere in the source program. Each pair of parentheses in the statement list encloses the names of two or more variables that are to be stored in the same location during execution of the object program. Any number of equivalences (sets of parentheses) can be given.

In an EQUIVALENCE statement, *C(p)* is defined as the location of the *pth* element in the array *C*. Thus, in the preceding example, the EQUIVALENCE statement indicates that A, and the B and C arrays are to be assigned storage locations so that the elements A, B(1), and C(5) are to occupy the same location. In addition, it specifies that D(17) and E(3) are to occupy the same location. This implies that D(15) and E(1) occupy the same location.

All variables that are to occupy the same location as a result of an EQUIVALENCE statement must be of the same type and must not be inconsistent in relative core-storage locations. For example, the statement

```
EQUIVALENCE (A(4), C(2), D(1)), (A(2), D(2))
```

is invalid. The equivalencing of A(4), C(2), and D(1) sets up an equivalence among elements of each row below.

```
A(1)
A(2)
A(3)    C(1)
A(4)    C(2)    D(1)
A(5)    C(3)    D(2)
 .       .       .
 .       .       .
 .       .       .
```

Thus, D(2) must not be equivalenced to A(2). EQUIVALENCE (A(3), A(4)) is also invalid.

Variables or arrays not mentioned in an EQUIVALENCE statement will be assigned unique locations.

The sharing of storage locations requires a knowledge of which Fortran statements cause a new value to be stored in a location. There are four such statements.

1. Execution of an arithmetic statement stores a new value in the variable to the left of the equal sign.
2. Execution of a DO statement, the terminal statement of a DO, or an implied DO in an I/O list stores a new indexing value.
3. Execution of a READ statement stores new values at the locations specified by variable names in the input list.
4. Execution of a subroutine or function may store a new value in any of its actual arguments or any variables in COMMON.

Variables brought into COMMON through EQUIVALENCE statements can increase the size indicated by the COMMON statements, as in the following example.

```
COMMON A, B, C
DIMENSION D(3)
EQUIVALENCE (B, D(1))
```

The layout of core storage indicated by this example (extending from the lowest location of COMMON to the highest location of COMMON) is:

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

A variable cannot be made equivalent to an element of an array in such a way as to cause the array to extend beyond the beginning of COMMON. For example, the following coding is invalid.

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

This would force D(1) to precede A, as follows.

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

FORTRAN RUN

This is the type of run that translates a source program written in the Fortran language into an object program in the relocatable format. The output for this run is then ready for processing by the Fortran loader.

Assumed Input Device. INPUT file on READER 1.

Input. Source program.

Assumed Output Devices. MESSAGE file on PRINTER 2; LIST file on PRINTER 2; LOADER file on 1311 or 1301 UNIT 0, START 010400, END 012000 or TAPE UNIT 3.

Output.

1. Source-statement diagnostics on the LIST file, if errors are sensed.
2. Source program listing on the LIST file, unless an output option control card specifies that the listing be omitted.
3. Name dictionary on the LIST file, unless an output option control card specifies that the name dictionary be omitted.
4. Sequence number dictionary on the LIST file, unless an output option control card specifies that the sequence number dictionary be omitted.
5. Object program in the relocatable format on the LOADER file, unless the LOADER file has been omitted.

Output Options Available. Object program in the relocatable format on the OUTPUT file.

Required User Assignments. If the object program is to be written on the LOADER file, no file assignments are required. If the user wants the object program on the OUTPUT file, an OUTPUT ASGN card is required.

Control Cards.

1. The RUN card is the only required control card. Punch the RUN card in the following manner.

Columns	*Contents*
6-12	FORTRAN
16-18	RUN

2. If the object program is to be on the OUTPUT file, an OUTPUT ASGN card is required. This card must precede the RUN card. Generally, the only time that the user would want to use the OUTPUT file is when a punched card deck in the relocatable format is desired. If this is the case, punch the OUTPUT ASGN card in the following manner:

Columns	*Contents*
6-11	OUTPUT
16-19	ASGN
21-27	PUNCH *n*

If the OUTPUT file is assigned to PUNCH *n*, *n* can be 0, 4, or 8 for 1402, 1 or 2 for 1442, 3 for 1444.

If the OUTPUT file is assigned, the user may wish to omit the LOADER file. If this be the case, a LOADER ASGN card is required and must precede the RUN card. Punch the LOADER ASGN card in the following manner.

Columns	*Contents*
6-11	LOADER
16-19	ASGN
21-24	OMIT

3. The following cards are output option control cards and compiler option control cards. They are used when any of the options are desired. The option cards immediately precede the Fortran source program in the INPUT file and can be in any order. Option control cards are output on the LIST file.

a. If the integer size is to differ from the assumed value of 5, punch the following card. *Note:* $01 \leq nn \leq 20$.

Columns	*Contents*
1-8	$INTEGER
10-13	SIZE
15	=
17-18	*nn*

b. If the real size is to differ from the assumed value of 8, punch the following card. *Note:* $02 \leq nn \leq 20$.

Columns	*Contents*
1-5	$REAL
7-10	SIZE
12	=
14-15	*nn*

c. If the object machine differs from the assumed value of 11999, punch the following card.

Columns	*Contents*
1-7	$OBJECT
9-15	MACHINE
17-20	SIZE
22	=
24-28	15999

d. If the multiply/divide feature is not present in the object machine, punch the following card. (The multiply/divide feature is assumed by the Fortran compiler to be available.)

Columns	*Contents*
1-3	$NO
5-12	MULTIPLY
14-19	DIVIDE

e. If the main program (phase)name differs from ///, punch the following card. Note: *name* is three alphameric characters in length. At least one character of the three alphameric character name must be alphabetic. If the program is to be stored as a phase on the SYSTEM file, the arrangement of the three characters must be unique, i.e., the arrangement must differ from:

1. Any phase name of the System Control Program
2. Any phase name of the Fortran Processor Program
3. Any phase name that the user previously may have added to the SYSTEM file.

Columns	*Contents*
1-6	$PHASE
8-11	NAME
13	=
15-17	*name*

f. If no source program listing is desired, punch the following card.

Columns	*Contents*
1-3	$NO
5-8	LIST

g. If no name dictionary listing is desired, punch the following card.

Columns	*Contents*
1-3	$NO
5-8	NAME
10-19	DICTIONARY

h. If no sequence number dictionary is desired, punch the following card.

Columns	*Contents*
1-3	$NO
5-12	SEQUENCE
14-19	NUMBER
21-30	DICTIONARY

i. If neither the name dictionary nor the sequence number dictionary is desired, punch the following card.

Columns	*Contents*
1-3	$NO
5-14	DICTIONARY

Arrangement. The arrangement of input cards for a FORTRAN RUN is shown in Figures 14 and 15. The output option cards must be in the INPUT file and can be in any order.

LOADER RUN

This is the type of run that changes object programs in the relocatable format into object programs in the absolute format, establishing interprogram communication and including required subprograms from the subprogram library. These object programs can be executed at the completion of the LOADER RUN, or at a later time, depending on the wishes of the user.

Assumed Input Device. LOADER file on 1311 or 1301 UNIT 0, START 010400, END 012000 or TAPE UNIT 3.

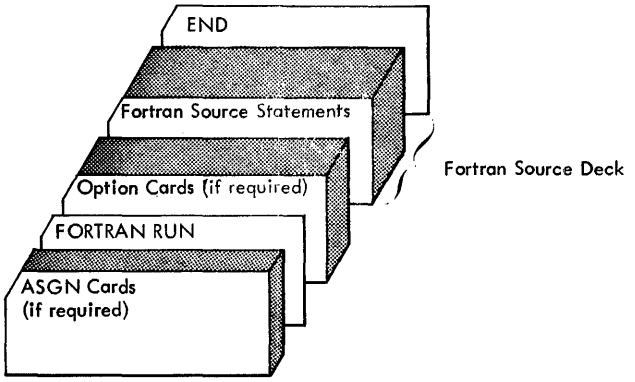


Figure 14. FORTRAN RUN with CONTROL and INPUT Files Assigned to the Same Device

Input. Object program(s) in the relocatable format.

Assumed Output Devices. LIST file on PRINTER 2, OUTPUT file on PUNCH 4 (1401 or 1460 systems) or PUNCH 1 (1440 systems).

Output.

1. Loader diagnostic messages on the LIST file, if errors are sensed.
2. Name map on the LIST file, unless an output option control card specifies that the name map be omitted.

Output Options Available.

1. Storage print on the LIST file, if an output option card is included that specifies a storage print.

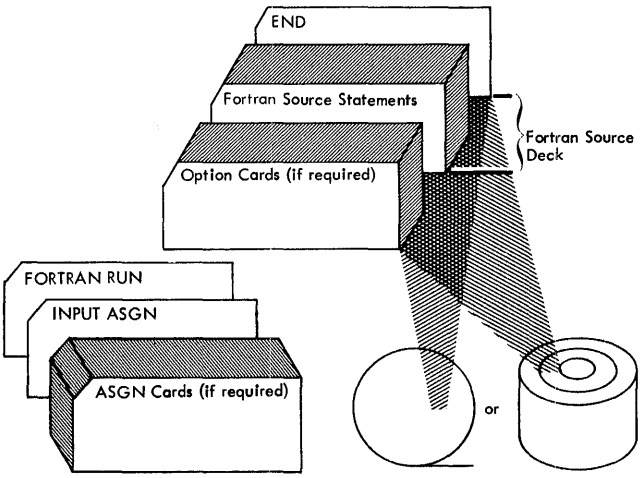


Figure 15. FORTRAN RUN with CONTROL and INPUT Files Assigned to Different Devices

2. Object program card deck in the absolute format on the OUTPUT file, if an output option card is included that specifies an absolute deck.

Additional Results. At the completion of a LOADER RUN, the program is ready for execution.

Required User Assignments. If the object program in relocatable format is on the LOADER file, no user-assignment is required. If the object program in relocatable format is on any other file (INPUT, WORK1-WORK6), an ASGN card is required designating this file. In this case, an associated output option card is required.

Control Cards.

The first card of a LOADER RUN job must be the LOADER RUN control card which is punched in the following manner:

Columns	*Contents*
6-11	LOADER
16-18	RUN

The last card of a LOADER RUN job must either be a $EXECUTION control card or a $NO EXECUTION control card.

a. If execution is desired, punch the following card:

Columns	*Contents*
1-10	$EXECUTION

b. If execution is not desired, punch the following card:

Columns	*Contents*
1-3	$NO
5-13	EXECUTION

Depending on the user's requirements, any or all of the following Loader control cards may be present between the LOADER RUN card and the $EXECUTION or $NO EXECUTION card. They may appear in any order.

a. The $INCLUDE card is punched in the following manner:

Columns	*Contents*
1-8	$INCLUDE
16-18	*three-character file name*
21-23	*three-character main program name*
or	or
21-26	*six-character subprogram name*

File Name	Three-Character File Name
LOADER	LDR
MESSAGE	MSG
INPUT	INP
OUTPUT	OUT
LIST	LST
WORK1	WK1
WORK2	WK2
WORK3	WK3
WORK4	WK4
WORK5	WK5
WORK6	WK6

Figure 16. Equivalence Between Logical File Names and Three-Character File Names

This card causes the Fortran Loader to search the file specified in columns 16-18 for the relocatable main program specified in columns 21-23, or the relocatable subprogram specified in columns 21-26. When the relocatable program is found, it is loaded into core storage and the next control card is read. When columns 21-26 are blank, the entire file of relocatable programs is loaded. The valid file names which may appear in columns 16-18 are; LDR, INP, WK1, WK2, WK3, WK4, WK5, and WK6. The correspondence between the three-character file names and the actual file names is shown in Figure 16.

b. If an absolute deck is desired, punch the following card. (Unless otherwise specified in columns 21-23, the deck will be on the OUTPUT file.)

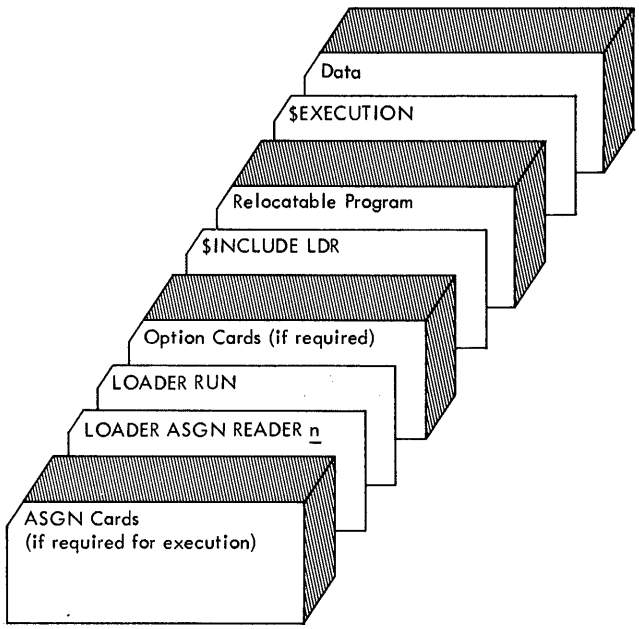
Columns	*Contents*
1-9	$ABSOLUTE
11-14	DECK
21-23	*three-character file name*

c. If a storage print is desired, punch the following card. (Unless otherwise specified in columns 21-23, the storage print will be on the LIST file.)

Columns	*Contents*
1-8	$STORAGE
10-14	PRINT
21-23	*three-character file name*

d. If no name map is desired, punch the following card:

Columns	*Contents*
1-3	$NO
5-8	NAME
10-12	MAP



● Figure 17. LOADER RUN with CONTROL and LOADER Files Assigned to the Same Device

Arrangement. The arrangement of input cards for a LOADER RUN is shown in Figures 17 and 18. The output option cards must be in the CONTROL file and can be in any order.

NOTE: If execution is to follow immediately after the LOADER RUN, indicated by a $EXECUTION card, the user must make sure that any Fortran numerical files referenced in the program have been assigned to the correct input/output devices. If ASGN cards are required to change file assignments, the cards precede the LOADER RUN card. Further, the user must make sure that the files referenced during a LOADER RUN do not conflict with files referenced in the object program.

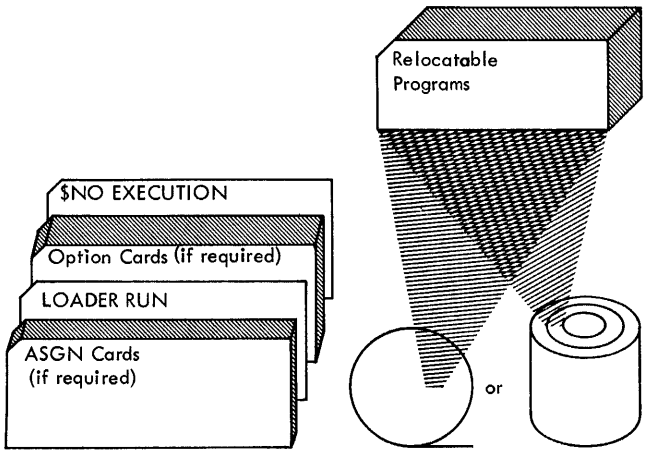


Figure 18. LOADER RUN with CONTROL and LOADER Files Assigned to Different Devices

Fortran Loader Operation

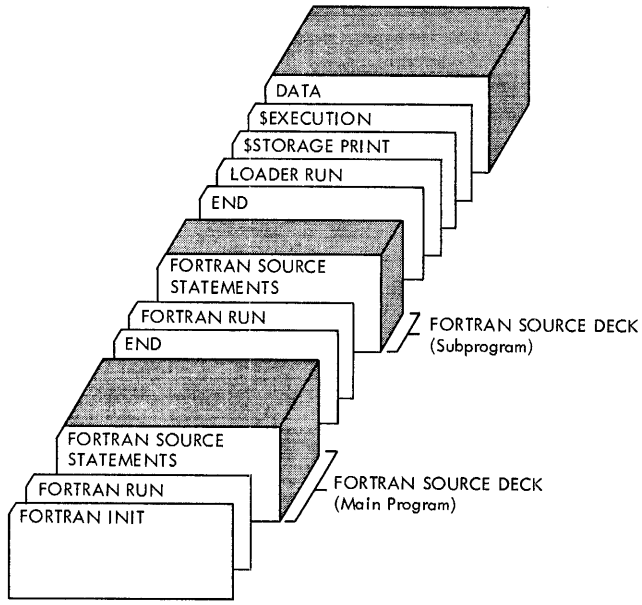
The Fortran LOADER is called by the System Control Program when the LOADER RUN card is read from the CONTROL file. All cards after the LOADER RUN card (up to and including the $EXECUTION or $NO EXECUTION card) on the CONTROL file are read by the Fortran LOADER. Relocatable programs may be loaded through the use of $INCLUDE cards as previously described. In addition, the entire LOADER file, which is developed by one or more Fortran runs, is loaded when the $EXECUTION or $NO EXECUTION card is read. The user should be aware that the LOADER file is not referenced if it has been referred to in a $INCLUDE card in the same LOADER run, or if it has been omitted with an ASGN card.

The LIBRARY file is always referenced and is the last file referenced in a LOADER run. Any subprograms, either supplied by IBM or entered on the LIBRARY file by the user, which are needed by the main program and/or subprograms already loaded, are extracted from the LIBRARY file.

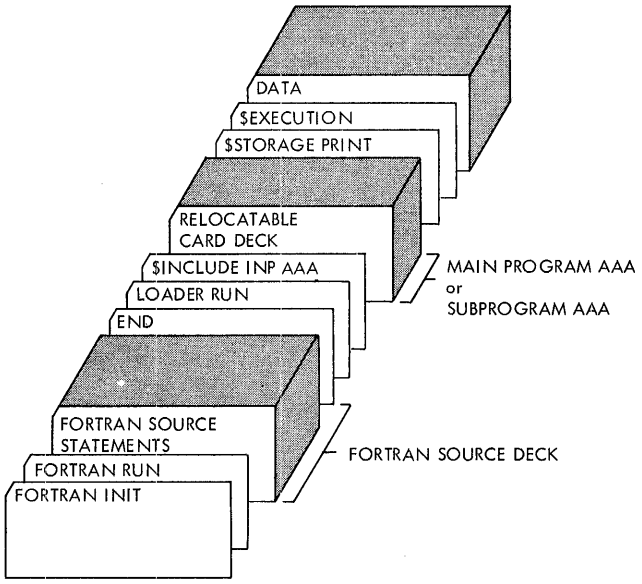
Following are three possible executions of the LOADER RUN job:

1. If the user has written a main program and an associated subprogram and wishes to compile and execute them, the card deck sequence of Figure 19 could be used.

   The two Fortran run jobs will place their respective relocatable decks on the LOADER file. Since there is no $INCLUDE card, at LOADER run time the entire LOADER file will be loaded when the $EXECUTION card is sensed.



● Figure 19. Fortran runs followed by *LOADER RUN*



● Figure 20. Main Program with Subprogram in Relocatable Card Form

2. If the user wishes to compile a Fortran program and then execute this program along with another program in relocatable card form, he may use the card deck sequence shown in Figure 20. One of the programs must be a main program and the other a subprogram. A program in the relocatable card form may be obtained from a Fortran run by assigning the OUTPUT file to the card punch.

   The Fortran run will place a relocatable deck on the LOADER file. When the $INCLUDE INP AAA card is read at LOADER RUN time, the Fortran loader loads program AAA from the INPUT file. Since this example uses the assumed logical file assignments, the INPUT file is on the same card reader as the CONTROL file. Therefore, the relocatable program AAA must immediately follow the $INCLUDE card.

   When the $EXECUTION card is read, the LOADER file, containing the program just compiled, will be loaded.

3. If the user wishes to load and execute one or more programs all of which are in the relocatable card form, he may use the card deck sequence shown in Figure 21.

   One of the three relocatable programs must be a main program, and the other two, subprograms.

   In this example, the CONTROL file, the INPUT file, and the LOADER file are all assigned to the same card reader. When each $INCLUDE card is read, the Fortran loader loads the specified relocatable deck from the LOADER file. When the $EXECUTION card is sensed, the Fortran loader will not reference the LOADER file again since it was referenced previously in a $INCLUDE card.

PRODUCTION RUN

This is the type of run that executes an object program in the absolute format. In order to perform a PRODUCTION RUN, the object program must be in the SYSTEM file, stored as a phase of the Fortran system. The method for placing the object program in the SYSTEM file is described in the following section, *Preparing User-Update Jobs.*

Assumed Input Device. For 1301, the SYSTEM file is on UNIT 0. For 1311, and for tape. the unit is user-assigned.

Input. Object program in the absolute format, stored as a phase of the Fortran system.

Assumed Output Devices. Not applicable.

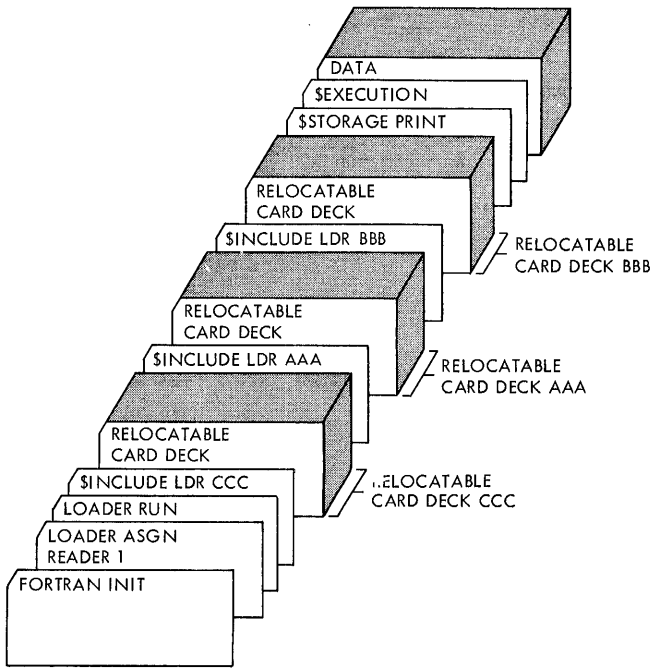
Output. Not applicable.

Required User Assignments. The unit(s) referenced by the source program.

Control Cards. The required RUN card is punched in the following manner.

Columns	*Contents*
6-15	PRODUCTION
16-18	RUN
21-23	*three-character phase name*

The *three-character phase name* is the three-character name assigned to the program before it was compiled. It is the same three-character name that appeared in columns 21-23 of the first card of the



● Figure 21. Main Program with Two Subprograms in Relocatable Card Form

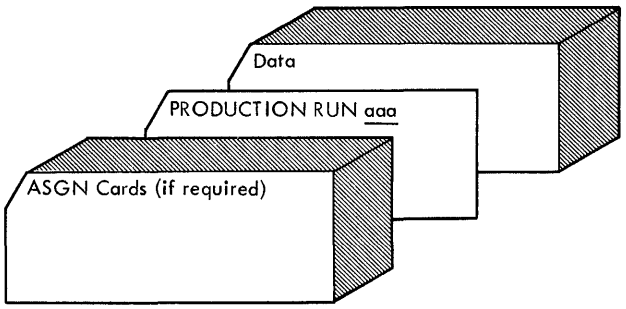


Figure 22. PRODUCTION RUN with CONTROL and INPUT (Equivalent to Fortran File 1) Files Assigned to the Same Device

absolute deck that was used to insert the program as a phase on the SYSTEM file.

Arrangement. The arrangement of input cards for a PRODUCTION RUN is shown in Figures 22 and 23.

Preparing User-Update Jobs

In order to perform a PRODUCTION RUN job, it is necessary that the object program in the absolute format be present in the SYSTEM file. This object program deck is output by a LOADER RUN, when specified by the user. Object programs can be inserted on or deleted from the SYSTEM file according to the needs of the user. If the system resides on a disk unit, these user-inserted phases (object programs in the absolute format) reside within the file-protected limits of the SYSTEM file.

On a tape-oriented system, the WORK1 file is used in conjunction with the SYSTEM file when a user-update job is performed. After a new phase is inserted (or deleted), the new SYSTEM file is present on WORK1. If the LIBRARY file followed the old SYSTEM file, it is copied following the new SYSTEM file on WORK1. According to the needs of the user, the WORK1 file can be transferred back to the master tape by performing a system-tape copy job. See *Duplicating the System Tape.* Since a system-tape copy job copies *from* the SYSTEM file *to* the WORK1 file, the user must make sure that the SYSTEM and WORK1 file assignments used for the user-update job are interchanged for the system-tape copy job. (Inserting a PAUSE card immediately after the user-update job and immediately preceding the system-tape copy job provides a temporary halt in the system that allows the user to interchange the two file assignments.)

The tape user is advised that when performing an UPDAT INSERT job, the phase after which the new phase is to be inserted must be present on the original SYSTEM file. The order of insertions (or deletions) must be the same as on the SYSTEM file.

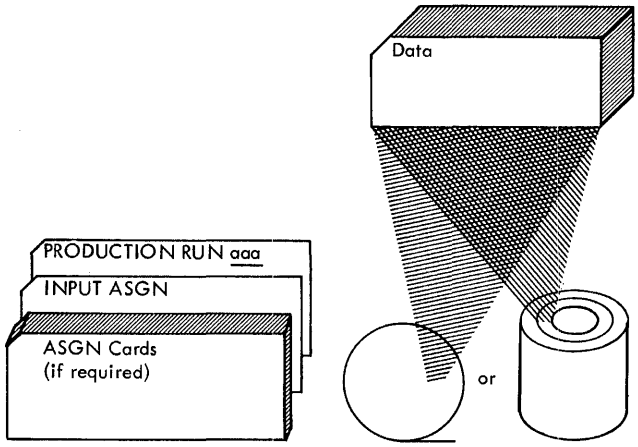


Figure 23. PRODUCTION RUN with CONTROL and INPUT (Equivalent to Fortran File 1) Files Assigned to Different Devices

Assumed Input Device. CONTROL file on READER 1.

Input. Object deck in the absolute format. This deck is obtained by selecting the absolute deck option in a LOADER RUN.

Assumed Output Devices. Not applicable.

Output. Not applicable.

Required User Assignments. None.

Control Cards.

1. If a phase is to be inserted on the SYSTEM file, the required UPDAT control card is generated by the Fortran loader. It is the first card of the absolute deck when the user specifies that an absolute deck be punched as a result of a LOADER RUN.

NOTE. If the SYSTEM file resides on tape, the user must specify that the phase be inserted after a particular phase on the SYSTEM file. In order that this be accomplished, punch the three-character phase name after which the phase is to be inserted in columns 21-23 of the UPDAT control card, the first card of the absolute deck. (Columns 21-23 of the UPDAT card punched by the LOADER contains the name of the phase that is to be inserted and must be changed.) 79F is the name of the last phase of the Fortran system. When deleting a phase and inserting another phase in its place, it is recommended that these two user-update jobs be performed in two separate stacks.

2. If a phase is to be deleted from the SYSTEM file, punch the following card.

Columns	*Contents*
6-15	[*any user comments*]
16-20	UPDAT
21-23	*three-character phase name*
24	comma
25-30	DELETE

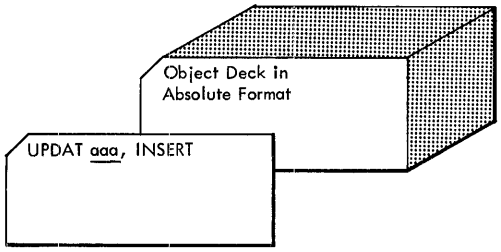


Figure 24. User-Update Job

In the control card, *three-character phase name* refers to the three-character main program name.

Arrangement. The arrangement of input cards for a user-update job is shown in Figure 24.

NOTE TO DISK USERS: Attempts to add a phase to the system may result in a halt (A-Address Reg.—088) indicating that the phase has not been inserted due to a lack of space in the system area. The following message will be printed before the halt occurs: SYSTEM AREA MUST BE OPTIMIZED BEFORE PHASE *name* CAN BE INSERTED. PRESS START TO OPTIMIZE. Pressing START will cause the system area to be scanned for all unused sectors. If unused sectors are found (resulting from prior deletions), the system area will be rearranged so that all unused sectors become available for phase insertion. Upon completion of this compression, another attempt will be made by the system to insert the phase. If the phase will not fit after compression, a hard halt will occur (A-Address Reg.—099). The following message will be printed before the halt occurs: NO ROOM IN SYSTEM AREA FOR PHASE *name-number* MORE SECTORS REQUIRED. Phases on the system that are no longer used may be deleted in order to make room for the phase to be inserted. A table containing the name and disk address of every phase in the system area is located in the file-protected area. A list of this table may be obtained by printing sectors 262583 through 262615 using a seek address of 002583 through 002615 (Load mode).

Preparing Library Jobs

Library jobs are associated with the maintenance of the Fortran library. The Fortran library is a mass-storage file that supports the Fortran loader. The file contains a library table (disk-resident systems only) and subprograms, such as standard Fortran functions and subroutines.

The three standard library jobs are:

1. Library build that enables the user to *define* a LIBRARY file. A library-build job, performed when a disk-resident Fortran system is built, defines a LIBRARY file on the same disk unit as the SYSTEM file. The limits of this LIBRARY file are 012000 *through* 013899. Thus, the assumed assignment is 1311 or 1301 UNIT 0, START 012000, END 013900.

   After the library-build job has been performed, the LIBRARY file contains the library table. The library table is thirty sectors in length. The user can enlarge the name table according to his needs.

   If the system is tape resident, the library can be built on the same tape unit with the SYSTEM file, or on another tape unit, if specified by a LIBRARY ASGN card. As a result, the tape user need only be concerned with library changes and library listings.
2. Library listing that enables the user to get a list of the library subprograms and/or the names of the subprograms that are in the LIBRARY file.
3. Library change that enables the user to modify the content of a LIBRARY file. A library-change job, first performed when the disk-resident system is built, transfers the subprograms to the LIBRARY file after the file has been defined.

A library job begins with a LIBRARY RUN card and terminates upon encountering the END card. Only four types of control cards can appear between the LIBRARY RUN and the END card. They are BUILD, LIST, INSER, and DELET.

At the completion of a library job (LIBRARY RUN) on a disk-oriented system, three messages are printed on the LIST file. The messages are:

END OF LIBRARY RUN
LIBRARY ASSIGNED *nnnnnn* TO *nnnnnn*
REMAINING SECTORS *nnnnnn* TO *nnnnnn*

In the message, *nnnnnn* is a disk address. From these messages, the user is able to determine the size of the present library, and the number of sectors available for any additional subprograms that may be added.

Any subprogram is stored in disk storage one card per sector in the move mode. As the input for a LIBRARY RUN must be in card form, the user can determine whether a subprogram will fit in the library by merely counting the cards in the deck output from a FORTRAN RUN.

If the LIBRARY file is full and the user wants to add a new subprogram, one of two steps can be followed.

1. The user can define and build a new LIBRARY.
2. The user can delete an existing subprogram from the LIBRARY file and insert the new subprogram. This can be done if the new subprogram will occupy the same number or fewer sectors than the old subprogram.

For tape-oriented systems, at the completion of a LIBRARY RUN, the message,

END OF LIBRARY RUN

is printed on the LIST file.

Library Build

Library-build jobs apply only to disk-resident systems. Each library-build job defines a LIBRARY file that contains a name table 30 sectors in length. If a table of more (or less) than 30 sectors is required, specify the sector number desired in the control card.

The control cards required for a library-build job are:

1. A LIBRARY ASGN card is required if the assignment of the LIBRARY file differs from that assumed by the System Control Program. Punch the ASGN card in the following manner:

Columns	*Contents*
6-12	LIBRARY
16-19	ASGN
21-57	1311 UNIT *n*, START *nnnnnn*, END *nnnnnn* *or* 1301 UNIT *n*, START *nnnnnn*, END *nnnnnn*

   For disk, the value *n* indicates the number of the disk unit, and can be 0, 1, 2, 3, or 4; *nnnnnn* represents a disk address. The limits of the library must be supplied.

2. Punch the required RUN card in the following manner:

Columns	*Contents*
6-12	LIBRARY
16-18	RUN

3. Punch the library-build card in the following manner:

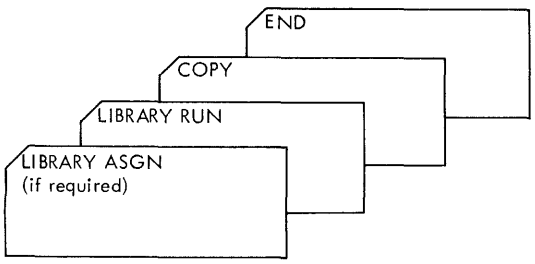
Columns	*Contents*
16-20	BUILD
21-23	[*nnn*]

   The value *nnn* is used only when the name table is to differ from 030 sectors.

4. The END card must be the last card of a library-build job. Punch the END card in the following manner:

Columns	*Contents*
16-18	END

The arrangement of control cards for a library build job is shown in Figure 25. The cards are read from the CONTROL file.



● Figure 25. Library Build

Library Listing

The user can request three types of library listings.

1. A listing of the names or headers of all the subprograms in the Fortran library.
2. A listing of the entries in a specific subprogram.
3. A listing of the entries in every subprogram.

The control cards required for a library-listing job are:

1. A LIBRARY ASGN card is required if the assignment of the library file differs from that assumed by the System Control Program. See *Library Build* for the format of the ASGN card.

2. The required RUN card is punched in the following manner.

Columns	*Contents*
6-12	LIBRARY
16-18	RUN

3. One of the following three cards are required for the library-listing job. The one that is selected depends upon the type of listing that is required.

   a. If a listing of the headers of all the tape subprograms is required, punch the following card. If a listing of the names and disk addresses of all the disk subprograms is required, punch the following card. The listing is output on the LIST file.

Columns	*Contents*
16-19	LIST
21-27	HEADERS

   b. If a listing of the entries in a specific subprogram is required, punch the following card. The listing is output on the LIST file.

Columns	*Contents*
6-11	*name*
16-19	LIST

   name is the six-character name of the specific subprogram entries that are required.

   c. If a listing of the entries in every subprogram is required, punch the following card. The listing is output on the LIST file.

Columns	*Contents*
16-19	LIST

4. The END card must be the last card of a library-listing job. Punch the END card in the following manner.

Columns	*Contents*
16-18	END

Appendix V — Increasing the Area of the Object Program Library

The disk area presently available for the FORTRAN IV user's object program library can be increased by either of the following methods:

1. Create a separate pack containing the System Control Program and the user's library.
2. Extend the file-protected area on the FORTRAN system pack.

Create a Separate Pack

This method of increasing the user's object program library disk area involves clearing a pack, file-protecting it, and then building the System Control portion of the pack. This can result in a library area of more than 16,800 sectors; approximately 1700 sectors were available previously.

To use this method, clear the pack from 002500 to 019980 in the load mode, then run the program to write file-protected addresses. Change the upper limit of the file-protected area (Col. 54-59 of the last card of this program) to 019980. After the pack has been file-protected, use only the portion of the FORTRAN IV (disk resident) system deck labelled Card Build and System Control. The use of these decks will place the System Control Program on the pack.

Now make a patch to IOP (the Input/Output Phase of the System Control Program). This patch consists of the following three cards:

	Columns	*Contents*
Card 1	16-29	UPDATIOP, PATCH
Card 2	1-11	84306279980
Card 3	16-20	PAUSE

Perform a normal update run with these cards placed behind the Card Boot.

After these steps have been completed, the user may use this separate pack to store and execute his object programs, just as he would if they were stored on the system pack.

Extend the FORTRAN System Pack

The second method of increasing the user's object program library disk area is to increase the size of the file-protected area already available on the FORTRAN system pack. The user should consider the following points before deciding to use this method:

1. To use this method, the user must rebuild the system pack.
2. The number of sectors to be gained is limited, because space must be reserved for the Loader File, the Library File, and (unless a second pack is available on-line) the Work Files.
3. Because the file-protected area must be continuous, the assumed assignments of all files assigned to disk (Work1—Work6, Library, and Loader) must be changed through the use of ASGN cards, and these ASGN cards must be used each time the system pack is used.
4. Use of a FORTRAN INIT card is prohibited.

To accomplish this method, the user must allocate the disk area to the various files to determine the upper limit of the file-protected area. Then he should clear and file-protect the pack, using the new upper limit in the clear storage cards and in the file-protect control card (Col. 54-59).

After file-protection is complete, the pack may be built as described in the section labeled *Building a Fortran System*, except that the new ASGN cards must be used each time that the Card Boot is used.

A patch to IOP (the Input/Output Phase of the System Control Program) is also needed. The upper limit of the file-protected area in its file-protected form (upper limit + 260,000) must be patched. This patch consists of the following three cards:

	Columns	*Contents*
Card 1	16-29	UPDATIOP, PATCH
Card 2	1-11	84306nnnnn0
Card 3	16-20	PAUSE

Place these cards behind the Card Boot and perform a normal update run. The pack then will contain the entire FORTRAN IV processor and an increased area for the user's object program library.

READER'S COMMENT FORM

Fortran IV Language Specifications, Programming Specifications and Operating Procedures 1401/1440/1460 C24-3322-2

- Your comments, accompanied by answers to the following questions, help us produce better publications for your use. If your answer to a question is "No" or requires qualification, please explain in the space provided below. All comments will be handled on a non-confidential basis. Copies of this and other IBM publications can be obtained through IBM Branch Offices.

	Yes	No
• Does this publication meet your needs?	☐	☐
• Did you find the material:		
Easy to read and understand?	☐	☐
Organized for convenient use?	☐	☐
Complete?	☐	☐
Well illustrated?	☐	☐
Written for your technical level?	☐	☐

- What is your occupation? _____
- How do you use this publication?

As an introduction to the subject?	☐	As an instructor in a class?	☐
For advanced knowledge of the subject?	☐	As a student in a class?	☐
For information about operating procedures?	☐	As a reference manual?	☐

Other _____

- Please give specific page and line references with your comments when appropriate. If you wish a reply, be sure to include your name and address.

COMMENTS:

- Thank you for your cooperation. No postage necessary if mailed in the U. S. A.

Staple

Fold

Fold

FIRST CLASS
PERMIT NO. 387
ROCHESTER, MINN.

BUSINESS REPLY MAIL
NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY . . .

IBM Corporation
Systems Development Division
Development Laboratory
Rochester, Minn. 55901

Attention: Programming Publications, Dept. 286

Fold

Fold

Cut Along Line

IBM 1401, 1440, and 1460 Printed in U.S.A. C24-3322-2

IBM

International Business Machines Corporation

Data Processing Division

112 East Post Road, White Plains, N. Y. 10601

Additional Comments:

IBM 1401, 1440, and 1460 Printed in U. S. A. C24-3322-2

IBM

International Business Machines Corporation

Data Processing Division

112 East Post Road, White Plains, N. Y. 10601