AA61516-03C

GENERAL SPECIFICATION
HARRIS FORTRAN COMPILER

Original
October, 1974

Revision A
February, 1975

Revision B
January, 1976

Revision C
June, 1976

HARRIS COMMUNICATIONS AND INFORMATION HANDLING

HARRIS CORPORATION Computer Systems Division

1200 Gateway Drive, Fort Lauderdale, Florida 33309 305/974-1700

LIST OF EFFECTIVE PAGES

TOTAL NUMBER OF PAGES IN THIS PUBLICATION IS: 179
CONSISTING OF THE FOLLOWING:

Page No.	Change	Page No.	Change	Page No.	Change
Title	C	3-1	A		
A	C	3-2 thru 3-4	Original		
i	Original	3-5	A		
ii	B	3-6	B		
iii	A	3-7 thru 3-18	C		
iv thru viii	C	4-1	Original		
1-1	A	4-2	B		
1-2, 1-3	C	4-3 thru 4-4	Original		
2-1 thru 2-3	A	4-5	B		
2-4 thru 2-6	Original	5-1 thru 5-5	A		
2-7	A	6-1	C		
2-8 thru 2-14	Original	6-2 thru 6-13	Original		
2-15	A	6-14 thru 6-16A	B		
2-16	Original	6-17 thru 6-20	Original		
2-17	A	7-1 thru 7-19	C		
2-18 thru 2-22	Original				
2-22A	A				
2-23 thru 2-31	Original				
2-32 thru 2-34	A				
2-35 thru 2-36	Original				
2-37	A				
2-38 thru 2-42A	B				
2-43	Original				
2-44	A				
2-44A thru 2-45	B				
2-46 thru 2-56	Original				
2-57	A				
2-58 thru 2-60	Original				
2-61	B				
2-62	Original				
2-63	B				
2-64 thru 2-66	Original				
2-67	B				
2-68 thru 2-79	Original				
2-80 thru 2-80A	B				
2-81	A				
2-82 thru 2-85	Original				
2-86	B				
2-87 thru 2-94	Original				

Insert Latest Revision Pages. Destroy Superseded Pages.

CONTENTS

Section		Page
I	INTRODUCTION	
	1-1 SCOPE	1-1
	1-2 GENERAL DESCRIPTION	1-1
	1-3 COMPILER ASSEMBLY OPTIONS	1-1
	1-4 EXTENSIONS	1-2
II	DC6024 FORTRAN LANGUAGE	
	2-1 GENERAL DESCRIPTION	2-1
	2-2 FORTRAN PROGRAMS	2-1
	2-3 DATA	2-3
	Integer Data	2-3
	Real Data	2-3
	Double-Precision Data	2-3
	Complex Data	2-4
	Logical Data	2-4
	Literal Data	2-4
	Constants	2-4
	Integer Constants	2-4
	Octal Constants	2-4
	Real Constants	2-5
	Double-Precision Constants	2-5
	Complex Constants	2-6
	Logical Constants	2-6
	Literal Constants	2-6
	Identifiers	2-7
	Variables	2-8
	Scalars	2-8
	Arrays and Subscripts	2-8
	Arrays in Storage	2-9
	2-4 EXPRESSIONS	2-9
	Arithmetic expressions	2-9
	Arithmetic Operaters	2-9
	Evaluation Hierarchy	2-10
	Mixed Expressions	2-11
	Relational Expressions	2-12
	Logical Expressions	2-13
	Logical Operators	2-14
	Evaluation Hierarchy	2-14
	Logical-Integer Operators	2-15

CONTENTS (CONT'D.)

Section		Page
II	DC 6024 FORTRAN LANGUAGE (CONT'D.)	
	The .SHIFT. Operator	2-15
	The .ROTAT. Operator	2-16
	Evaluation Hierarchy	2-17
2-5	Assignment Statement	2-17
2-6	Control Statements	2-19
	Labels	2-19
	GO TO Statements	2-19
	Unconditional GO TO Statements	2-20
	Assigned GO TO Statements	2-20
	Computed GO TO Statement	2-21
	ASSIGN Statement	2-21
	Arithmetic IF Statements	2-22
	Logical IF Statement	2-23
	CALL Statement	2-23
	RETURN Statement	2-24
	DO Statement	2-25
	CONTINUE Statement	2-29
	PAUSE Statement	2-29
	STOP Statement	2-30
	END Statement	2-30
2-7	Input/Output	2-30
	I/O List	2-31
	Simple List	2-31
	DO-Implied Lists	2-32
	Free Format READ/WRITE	2-33
	READ Statement	2-35
	WRITE Statement	2-36
	FORMAT Statement	2-36
	F Format	2-38
	E Format	2-39
	D Format	2-40
	G Format	2-41
	I Format	2-41
	O Format	2-42
	L Format	2-42
	A Format	2-42A
	R Format	2-44
	H Format	2-44
	X Specification	2-45
	T Specification	2-46
	P Specification	2-46
	/ Specification	2-48
	Parenthesized Format Specifications	2-48
	Numeric Input Strings	2-49
	FORMAT and List Interfacing	2-50
	FORMAT - Stored in Arrays	2-51

CONTENTS (CONT'D.)

Section		Page
II	DC 6024 FORTRAN LANGUAGE (CONT'D.)	
	Auxiliary Input/Output Statements	2-52
	REWIND Statement	2-52
	BACKSPACE Statement	2-52
	END FILE Statement	2-53
	OPEN Statement	2-53
	CLOSE Statement	2-53
	CALL EOFTST	2-53
	CALL SSWTCH	2-54
	CALL BTIME	2-54
	CALL ETIME	2-54
	List Output Carriage Control	2-54
	Input/Output Logical File Assignments	2-55
	Random Access I/O	2-55
	DEFINE FILE Statement	2-56
	Disc I/O Statements	2-56
	Random Access I/O - Disc I/O Statements	2-57
	ENCODE/DECODE Statements	2-58
	Asynchronous Input/Output	2-61
2-8	Declaration Statements	2-63
	Classification of Identifiers	2-63
	Implicit Declarations	2-64
	Explicit Declarations	2-64
	Conflicting and Redundant Declarations	2-65
	Array Declarations	2-65
	Array Storage	2-66
	Reference to Array Elements	2-66
	DIMENSION Statement	2-67
	IMPLICIT Statement	2-67
	Storage Allocation Statements	2-68
	Allocation of Variable Types	2-68
	COMMON Statement	2-68
	Labeled COMMON	2-69
	Blank COMMON	2-70
	Arrangement of COMMON	2-70
	MONITOR COMMON Statement	2-72
	SPECIAL COMMON Statement	2-73
	EQUIVALENCE Statement	2-73
	COMMON and EQUIVALENCE Interactions	2-75
	EXTERNAL Statement	2-78
	DATA Statement	2-78
	DATA Variable List	2-79
	DATA Constant List	2-80

CONTENTS (CONT'D.)

Section		Page
II	DC 6024 FORTRAN LANGUAGE (CONT'D.)	
	2-9 Programs and Subprograms	2-81
	Main Program	2-81
	Subprograms	2-81
	Functions	2-81
	Subroutines	2-81
	Statement Functions	2-82
	FUNCTION Subprograms	2-83
	Library Functions	2-84
	SUBROUTINE Subprograms	2-86
	BLOCK DATA Subprograms	2-88
	ENTRY Statement	2-88
	Arguments and Dummies	2-89
	Dummy Scalars	2-91
	Dummy Arrays	2-91
	Adjustable Dimensions	2-93
	Dummy Subprograms	2-93
	RECUR Statement	2-94
	NAME Statement	2-94
III	FORTRAN DIAGNOSTIC MESSAGES	
	3-1 General Description	3-1
	3-2 Compile-Time Diagnostics	3-1
	3-3 Run-Time Diagnostics	3-5
	3-4 In-Line Control Statements	3-9
	3-4.1 Conditional-compile Control Statements	3-9
	3-4.1.1 Skip-Statements	3-9
	3-4.1.2 ESKP-Statement	3-10
	3-4.1.3 Use of Conditional-Compilation	3-11
	3-4.2 Option Control Statements	3-12
	3-4.3 Initial Values of Compile-Time options	3-13
	3-5 Compile-Time Options	3-13
	3-6 Debug Statements	3-17
IV	INTERFACING FORTRAN AND ASSEMBLER GENERATED MODULES	
	4-1 Calling FORTRAN Generated Subprograms	4-1
	4-2 Calling of FORTRAN Functions	4-1
	4-3 FORTRAN-Callable Assembler Language Programs	4-2
V	OPTIMIZATION	
	5-1 General	5-1
	5-2 Compiler Optimization	5-1
	5-2.1 Sub-Program Entries	5-1
	5-2.2 IF Statements	5-1

CONTENTS (CONT'D.)

Section Page

V OPTIMIZATION (CONT'D.)

5-2.3 Immediate Instructions 5-1
5-2.4 Subscripts 5-1
5-2.5 Sub-Expression Optimization 5-2
5-2.6 Logical IF Optimization 5-2
5-2.7 Intrinsic Functions 5-2
5-2.8 Simple Integer Expressions 5-2
5-2.9 Index Registers 5-3
5-2.10 Inter Statement Optimization 5-3

VI IN-LINE CODE

6-1 Introduction 6-1
6-1.1 Scope of Extension 6-1
6-1.2 General Description 6-1
6-2 Source Language Format 6-1
6-2.1 Scope 6-1
6-2.2 Statement Number or Label Field 6-1
6-2.3 Operation Field 6-2
6-2.4 Operand and Comments Field 6-2
6-2.5 Sequence Field 6-2
6-3 Operand Formats 6-2
6-3.1 Scope 6-2
6-3.2 Current Location (*) 6-3
6-3.3 Symbolic Labels 6-3
6-3.3.1 Common 6-3
6-3.3.2 Array 6-4
6-3.3.3 Data Reference 6-5
6-3.3.4 Statement Number 6-5
6-3.3.5 Dummy 6-6
6-3.3.6 Variable 6-7
6-3.3.7 Function Results 6-7
6-3.3.8 External Requests 6-7
6-3.4 Absolute Constants 6-8
6-3.5 Address Arithmetic 6-8
6-3.6 Indexed Address Reference 6-9
6-3.7 Text 6-9
6-3.8 Literal 6-9
6-3.9 FORTRAN External 6-10
6-3.10 Memory Referencing Boolean Instructions (Bit-Processor) 6-10
6-4 Pseudo-Operations 6-11
6-4.1 Scope 6-11
6-4.2 RORG (Relative Origin) 6-11
6-4.3 BLOK (Reserve Memory) 6-11
6-4.4 DATA 6-12

CONTENTS (CONT'D.)

Section | Page

VI IN-LINE CODE (CONT'D.)

6-4.4.1 Single Integer 6-12
6-4.4.2 Single-Precision Real 6-12
6-4.4.3 Double-Precision Real 6-12
6-4.4.4 Octal Constants 6-13
6-4.4.5 Logical Constants 6-13
6-4.4.6 Complex 6-13
6-4.4.7 Hollerith 6-14
6-4.4.8 Truncated Text 6-14
6-4.4.9 Binary 6-14
6-4.5 DAC 6-14
6-4.6 BAC 6-15
6-4.7 *** 6-14
6-4.8 ZZZ 6-14
6-4.9 Octal Operation Code 6-15
6-4.10 REEN 6-15
6-4.11 PORG 6-16
6-4.12 PDATA 6-16
6-4.13 PBLOK 6-16
6-4.14 RDAT 6-16
6-4.15 PRDAT 6-16
6-5 Compile-Time and Run-Time Options 6-16A
6-5.1 Scope 6-16A
6-5.2 List Control 6-16A
6-5.3 Compile-Time Options 6-16A
6-5.4 Run-Time Options 6-16A
6-5.5 Error Codes and Messages 6-16A
6-5.6 Run-Time Diagnostics 6-18
6-6 Miscellaneous 6-18
6-6.1 Scope 6-18
6-6.2 DO Termination 6-18
6-6.3 Automatic Symbol Assignment 6-19
6-6.4 Second Operand Field 6-19
6-6.5 Forward Reference Symbols 6-19
6-6.6 Path Diagnostic 6-20
6-6.7 Statement Function Dummies 6-20

VII HARRIS STRUCTURED - FORTRAN LANGUAGE

7-1 General 7-1
7-2 Structured Programs 7-1
7-2.1 Scope 7-1
7-2.2 Structured - FORTRAN Compiler 7-1
7-2.3 Block-Statements, Boundary-Statements and Exit-Statements 7-1
7-2.4 Blocks of a Program Unit 7-2
7-2.5 Range(s) of a S-block 7-2
7-2.6 Initial-Statements 7-2

CONTENTS (CONT'D.)

Section Page

VII HARRIS STRUCTURED - FORTRAN LANGUAGE (CONT'D.)

7-2.7 Terminal-Statements 7-2
7-2.8 Alternative-Initial-Statements 7-2
7-2.9 Exit-Statements 7-3
7-2.10 S-Blocks and Associated Block Statements 7-3
7-2.11 Block-Level 7-5
7-3 BOUNDARY-STATEMENTS 7-5
7-3.1 Scope 7-5
7-3.2 IF-Block 7-5
7-3.2.1 Block-IF Statement 7-6
7-3.2.2 OR IF Statement 7-6
7-3.2.3 ELSE Statement 7-7
7-3.2.4 Termination of Execution of a Range of an IF-block . . . 7-7
7-3.2.5 END IF Statement 7-7
7-3.2.6 Order of OR IF and ELSE Statements 7-7
7-3.3 WHILE-BLOCK 7-8
7-3.3.1 WHILE Statement 7-8
7-3.3.2 END WHILE Statement 7-8
7-3.4 DO-UNTIL-BLOCK 7-9
7-3.4.1 Block-DO Statement 7-9
7-3.4.2 UNTIL Statement 7-10
7-3.5 FOR-Block 7-11
7-3.5.1 FOR Statement 7-11
7-3.5.2 LOOP-Control Processing of a FOR-block 7-11
7-3.5.3 END FOR Statement 7-12
7-3.5.4 Transfer Into the Range of a FOR-block 7-12
7-3.6 LOOP-BLOCK 7-13
7-3.6.1 LOOP Statement 7-13
7-3.6.2 Loop-Control Processing of a Finite-LOOP-block 7-14
7-3.6.3 END LOOP Statement 7-14
7-3.6.4 Iteration Count Specification of a Finite-LOOP-block . 7-14
7-3.6.5 Transfer into the Range of a LOOP-block 7-14
7-4 Exit Statements 7-15
7-4.1 Scope 7-15
7-4.2 Unconditional-exit-Statements 7-15
7-4.3 Conditional-exit-Statement 7-15
7-5 Indented Listing 7-17
7-6 Compile-Time Diagnostics 7-18

ILLUSTRATIONS

Figure		Page
2-1	FORTRAN Coding Sheet	2-2
2-2	An Example of Array Storage	2-66
7-1	Indented Listing	7-19

TABLES

Table		Page
2-1	Allowable Mixed Mode Exponentiation	2-11
2-2	Evaluation of Logical Operators	2-14
2-3	Truth Table for Logical Operators	2-14
2-4	Expression Type for Mixed Variable Assignments	2-18
2-5	List Output Device Carriage Control Characters	2-54
2-6	Logical File Assignments	2-55
2-7	Storage Allocation Requirements	2-68
2-8	Library Functions	2-85
2-9	Permissible Correspondence Between Arguments and Dummies	2-89
3-1	Compile Time Diagnostic Message	3-2
3-2	Run Time Diagnostics	3-9
3-3	Compile-Time Options	3-14
4-1	Register Location of Function Values	4-2
7-1	S-Blocks and Associated Block-Statements	7-3
7-2	A Structured Program	7-4

SECTION I
INTRODUCTION

1-1 SCOPE OF SPECIFICATION

This specification contains the information required to write and execute Harris FORTRAN programs. Also included are the requirements for interfacing FORTRAN and Harris assembly language programs.

The material in this document is presented on a level that assumes the reader has previous experience with the FORTRAN language.

1-2 GENERAL DESCRIPTION

The Harris FORTRAN system consists of a compiler and a FORTRAN library. The computer translates FORTRAN programs and subprograms into relocatable modules. Compiler-generated modules are accepted by the Harris Link Loader and can be externally linked with FORTRAN or assembly language programs and subsequently executed. The FORTRAN library is a set of subroutines, coded in assembly language, that provides: (a) all functions whose calls are generated by the compiler; (b) run-time diagnostic messages; and (c) a set of mathematical routines that includes USA standard FORTRAN functions as a subset. The entire FORTRAN system operates within all the Harris Operating Systems.

1-3 COMPILER ASSEMBLY OPTIONS

The Compiler Assembly options consist of the following:

1. Basic Compiler (Assembled with FLAG 03 on) - Uses approximately 4000_8 words of core less than the extended compiler.
2. Extended Compiler (FLAG 03 off). Has the following additional features:
    a. Random Access I/O.
    b. The extended listing prints generated Mnemonics as well as octal code.
    c. $OPTIONS 4 treats Real numbers as double precision.
    d. Compile time errors print a message as well as a number.
    e. NAME statement available.
    f. RECUR statement available.
    g. RETURN n.
    h. Buffer In/Buffer Out.
    i. IMPLICIT statement.
    j. In-Line Assembly Code.

k. FORTRAN II I/O

l. Multiple Entry Points

m. Output TRIAD Option

3. DO option (FLAG 0 set) - the range of the DO loop will always be executed at lease once, even if conditions for termination are met initially.

4. SAU (FLAG 5 set) - for use on all machines with floating-point hardware.

5. VULCAN Compiler (FLAG 04 off). Must be used with VULCAN operating systems

6. DMS/DOS/TOS/ROS Compiler (FLAG 04 on). Must be used on DMS, DOS, TOS and ROS operating systems.

7. Structured-FORTRAN Compiler (FLAG 06 on) - Processes structured programming statements in addition to normal FORTRAN statements. This extension is availabl in the Basic Compiler as well as the Extended Compiler.

1-4 EXTENSIONS

The Harris Compiler meets the requirements of the ANSI X3.9 specification and has several extensions.

Random Access I/O*
Buffer In/Buffer Out*
Encode/Decode
Free Format I/O
FORTRAN II I/O
IMPLICIT Statement*
ENTRY Statement*
Recursion Statement*
Multiple RETURN definition*
NAME Statement*
Octal Constants
Mixed Mode Arithmetic
SHIFT Operator
ROTATE Operator
Exclusive OR Operator
Tab Specification
In-Line Assembly Code*

*Extended Compiler Only.

In-Line Control Statements*

Conditional-Compilation of blocks of Statements*

Conditional-Compilation of Debug Statements*

Structured programming Statements**

Automatic Identation of Structured programs**

*Extended Compiler Only.

**Structured FORTRAN Compiler only.

SECTION II
HARRIS FORTRAN LANGUAGE

2-1 GENERAL DESCRIPTION

The Harris FORTRAN language is a super set of USA Standard FORTRAN X3.9 - 1966.

2-2 FORTRAN PROGRAMS

FORTRAN programs are comprised of an ordered set of statements that describe the procedure to be followed during execution of the program and the data to be processed by the program. Some data values to be processed may be external to the program and read into the computer during program execution. Similarly, data values generated by the program can be written out while processing continues. Statements are of one of two general classes:

1. Executable statements, which perform computation, input/output operations, and program flow control.

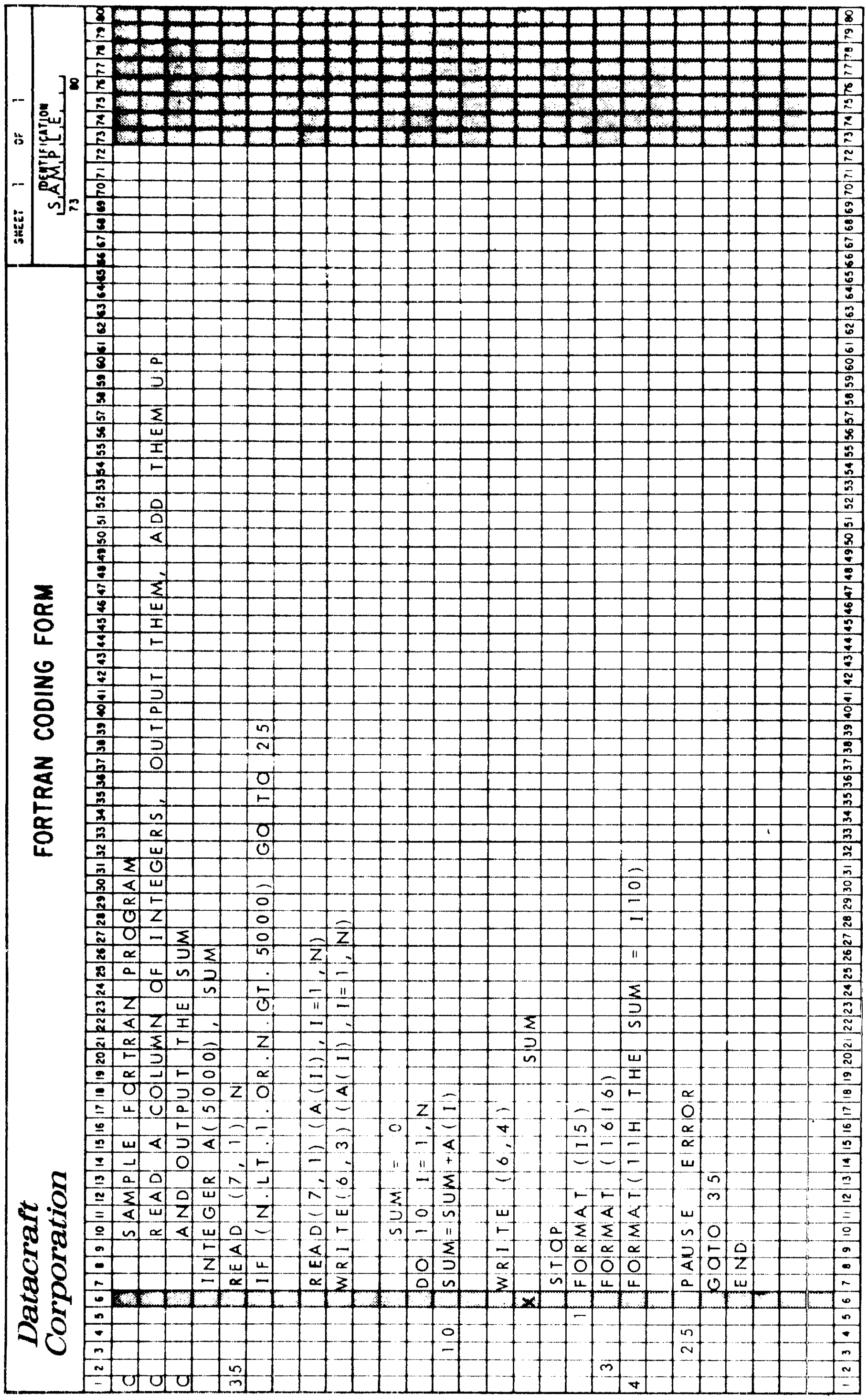
2. Nonexecutable statements, which provide information to the computer about storage assignments, data types, and program form, as well as providing information to the program during execution about input/output formats and data initialization.

Statements defining a FORTRAN program follow a prescribed format. Figure 2-1 is a sample FORTRAN Coding Form. Each line on the form consist of 80 spaces or columns; however, the last eight columns are used only for identification or sequence numbers and have no effect on the program. Columns 1 through 72 are used for the statements.

The first field, columns 1 through 5, is used for statement labels. Statement labels allow a statement to be referenced by other portions of the program. Labels are written as decimal integers, with all blanks (leading, embedded, or trailing) ignored. A more extensive discussion of statement labels is covered in Section 2-6.

The body of each statement is written in columns 7 through 72, but if additional space is required, a statement may be continued on as many lines as necessary. Each continuation line must contain a character other than blank or zero in column 6. The initial line of each statement may contain only the characters blank or zero in column 6. If a statement is labeled, the label must appear on the initial line of the statement, labels appearing on continuation lines will generate a non-fatal error.

Column 1 may contain the character C to indicate that the line is to be treated as a comment only, with no effect on the program. Comment lines may appear anywhere in the program.



Datacraft Corporation

FORTRAN CODING FORM

SHEET 1 OF 1

IDENTIFICATION SAMPLE

```
C     SAMPLE FORTRAN PROGRAM
C     READ A COLUMN OF INTEGERS, OUTPUT THEM, ADD THEM UP
C     AND OUTPUT THE SUM
      INTEGER A(5000), SUM
35    READ (7,1) N
      IF (N.LT.1.OR.N.GT.5000) GO TO 25
      READ(7,1)(A(I),I=1,N)
      WRITE(6,3)(A(I),I=1,N)
         SUM = 0
      DO 10 I=1,N
  10  SUM=SUM+A(I)
      WRITE (6,4)
     X            SUM
      STOP
    1 FORMAT (I5)
 3    FORMAT (16I6)
4     FORMAT(11H THE SUM = I10)
  25  PAUSE ERROR
      GOTO 35
      END
```

Figure 2-1. FORTRAN Coding Sheet

2-2 FORTRAN PROGRAMS (CONT'D.)

Statements may have blanks inserted as desired to improve readability, except within literal fields (e.g., in Hollerith constants and FORMAT statements).

The set of characters acceptable to Harris FORTRAN is:

Letters:	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Digits:	0 1 2 3 4 5 6 7 8 9
Special characters:	+ - * / = ( ) . , $ ' blank "

2-3 DATA

Numerical quantities (constants and variables) as distinguished in FORTRAN are a means of identifying the nature of the numerical values encountered in a program. A constant is a quantity whose value is explicitly stated. For example, the integer 5 is represented as "5", the number pi , to three decimal places, as "3.142". A variable is a numerical quantity that is referenced by name rather than by its explicit appearances in a program statement. During the execution of the program, a variable may take on many values rather than being restricted to one. A variable is identified and referenced by an identifier.

All data processed by the Harris FORTRAN program can be classed as one of six types; (1) integer, (2) real, (3) double-precision, (4) complex, (5) logical, and (6) literal.

Limits on Values of Quantities

Integer Data

Integer data are precise representation of the range of integers from -8,388,608 to +8,388,607; that is, -2^{23} to $+2^{23}-1$. Integer data may only be assigned integral values within this range.

Real Data

Real data (also called floating-point data) can be assigned approximations of real numbers with magnitudes within the range of 2.94×10^{-39} to 1.7×10^{38}. A real datum may acquire positive or negative values within the this range or the value zero. Real data have an associated precision of 6+ significant digits. That is, the sixth most significant digit will be accurate, while the seventh will sometimes be accurate, depending on the value assigned to the datum.

Double-Precision Data

Double-precision data may approximate the identical set of values as real data. However, double-precision data have an associated precision of 11+ significant digits.

Limits on Values of Quantities (Cont'd.)

Complex Data

Complex data are approximations of complex numbers. These approximations take the form of an ordered pair of real data. The first of the two real data approximates the real part, and the second real datum approximates the imaginary part of the complex number. The values that may be assigned to each part are identical to the set of values for real data.

Logical Data

Logical data can acquire only the values "true" or "false".

Literal Data

Literal data are character strings. Like logical data, literal data do not have numeric values. Any of the characters discussed under "Literal Constants" may appear in literal data.

Constants

Constants are data that do not vary in value and are referenced by naming their values. There are constants for each type of data. Numeric constants may be preceded by a "+" or "-". If not preceded by either of these, the constant is considered to be positive.

Integer Constants

Integer constants are represented by strings of digits. The magnitude of an integer constant must not exceed 8,388,607.

Examples:

382	+997263	100000	000546	-8
+13	1961	3344224	372436	0

Octal Constants

Octal constants are represented by a string of digits from the subset of digits 0, 1, 2, 3, 4, 5, 6, 7 and preceded by an apostrophe ('). There must be no more than eight significant digits in a string. If less than eight digits appear, the octal constant specified is right justified in the 24-bit word with leading zeros. An octal constant may be used interchangeably with an integer constant.

Constants (Cont'd.)

Examples:

'77777777	'0234	'40005
'10020040	'10000	'12345770

Real Constants

Real constants are represented by strings of digits with a decimal point and/or an exponent. The exponent follows the numeric value and consists of the letter E followed by a signed or unsigned integer that represents the power of ten by which the numeric value is to be multiplied. Thus the following forms are permissible:

```
n.m        n.       .m
n.mE±e     n.E±e    .mE±e       nE±e
```

where: n, m, and e are strings of digits, and the plus or minus sign preceding e is optional.

For example, .567E5 has the meaning $.567 \times 10^5$ and can also be represented by any of the following equivalent forms:

0.567E+05	5.67E4	56700.
567000E-1	567E02	56700.000E-00

The value of a real constant may not exceed the limits for real data. As many digits as desired may be written in a real constant, but only 6+ most significant digits are retained.

Since any real constant may be written in a variety of ways, the user can choose the form he prefers.

Examples:

5.0	7.6E+5	3.1415926535897
0.01	6.62E-27	5878550402984.0

Double Precision Constants

Double-precision constants are formed exactly like real constants, except that the letter D is used preceding the exponent instead of E. To denote a constant specifically as double-precision, the exponent must be present. Thus, a double-precision constant may be written in any of the following four forms:

```
n.mD±e     n.D±e    .mD±e       nD±e
```

where: n, m, and e are strings of digits, the plus or minus sign preceding e is optional, and D signifies a double-precision constant.

Constants (Cont'd.)

Double Precision Constants (Cont'd.)

The value of a double-precision constant may not exceed the limits for double-precision data. As many digits as desired may be written in a double-precision constant but only the 11+ most significant digits are retained.

Complex Constants

Complex constants are expressed as an ordered pair of constants in the form (c_1,c_2), whe c_1 and c_2 are signed or unsigned real constants. The complex constant (c_1,c_2) is interpreted as meaning $c_1 + c_2i$, where $i = \sqrt{-1}$. Thus, the following complex constants have values as indicate

(1.34,52.01)	=	1.34	+	52.01i
(98344.,0.34452E+02)	=	98344.0	+	34.452i
(-1.,-1000.)	=	-1.0	+	-1000.0i

Neither part of a complex constant may exceed the value limits established for real data.

Logical Constants

Logical constants may assume either of two forms:

.TRUE.   .FALSE.

These forms have the logical values "true" and "false", respectively.

Literal Constants

A literal constant takes one of the following forms:

nHs   's'   "s"

where:

s is a Hollerith string. Note that blanks are significant in Hollerith strings. If it is desired to use quotes (' or ") within either of the last two forms, the following technique should used. If the quote that is used to delimit the string is to be included in the string, then the quot should be entered as two consecutive quotes:

' " ' is equivalent to " ' " and """" is equivalent to '"'.

2-3 DATA (CONT'D.)

Constants (Cont'd.)

Literal Constants (Cont'd.)

n is an unsigned integer specifying the number of characters in the string. Note that if this form is used, then quotes may be freely included within the string without additional care:

1H' is equivalent to '''' or "'" and

1H" is equivalent to ' " ' or """"

Literal constants may be used freely in place of integer or real constants. The mode of literal constants as determined by the first non-literal data which is associated with it. For example:

I = 1HA	(literal is integer)
X = J + 1H$.	(literal is integer)
I = 1H3 + '8' + 1.5	(literal is real)

NOTE

Except as arguments to subprograms and in DATA statements, Literal constants have a length limitation. The total effective length of a literal constant with a mode of integer is three. The total effective length for a "real" literal constant is six.

Identifiers

Identifiers are strings of letters and decimal digits used to name variables, subprograms and COMMON blocks. An identifier in Harris FORTRAN consists of one to six alphanumeric characters. The first of which must be a letter.

Examples:

X A345Q STRESS J3 BOYER

Blank characters embedded in identifiers are ignored; therefore, BIG CAT and BIGCAT are identical. There are no restricted identifiers in Harris FORTRAN; however, for clarity, it is advisable not to use identifiers that correspond to Harris FORTRAN statement types (see 2-6 CONTROL STATEMENTS).

Constants (Cont'd.)

Identifiers (Cont'd.)

Identifiers having more than six characters are accepted but only the first six characters are used. In this case, a warning message is output so that the user may determine the uniqueness of all identifiers greater than six characters. Subsequent execution is not suppressed.

Variables

Variables are data whose values may vary during program execution and which are referenced with an identifier. Variables may be any of the data types. There is no such entity a a literal variable; any type of variable (except logical) may contain a literal string.

If a variable has not been assigned to a particular data type (refer to "2-8 DECLARATIC STATEMENTS - Classification of Identifiers"), the following implicit typing conventions are assumed:

1. Variables whose identifiers begin with letters I, J, K, L, M, or N are integer.
2. Variables whose identifiers being with any other letter are real.

Consequently, double-precision, complex, and logical variables must be explicitly declared as such (refer to Section 2-8, DECLARATION STATEMENTS - Explicit Type Statements). The values assigned to variables may not exceed the limits established for the applicable data types.

NOTE

This implicit typing may be modified through the use of the IMPLICIT statement (See Section 2-8).

Scalars

A scalar variable is a single datum entity and is accessed via an identifier of the appropriate type.

Examples:

```
J1      NAME      SCALAR      EQUATE      E      NEW      DHO      XXX8
```

Arrays and Subscripts

A variable may be made to represent a one, two, or three dimensional array by adding subscripts to the variable name. In FORTRAN, the variable name is followed by parentheses enclosing the subscripts which are separated by commas. The subscripts determine the element of the array to which reference is made.

Every subscripted variable must have the array size declared in a DIMENSION, COMMON or Type statement.

Examples of array element references are:

```
A (1)
VECTOR (M)
B (2*I+5, 3*J-2)
C (4*INT)
YELLOW (I, J, K)
```

Arrays and Subscripts (Cont'd.)

A subscript may be any arbitrary expression whose final mode is real or integer. If the mode is real, the value is converted to integer (the fractional part is not rounded) before the subscripting operation is performed. The following array element references are valid:

```
A(A(I))
BETA (SIN(X/Y)+0.5)
ABC(4,FUNC(3.4),TANH(W)/2)
```

Note that if a subscript expression contains another array element reference, then this nesting of subscripts may be continued to a maximum depth of 10.

Arrays in Storage

Arrays are stored in column sequence, with the first subscript varying most rapidly and the last varying least rapidly.

For example, the two dimensional array A_{mn} is stored as follows; from lowest numbered memory location to highest:

$$A_{11} \quad A_{21} \quad A_{31} \cdots \quad A_{m1} \quad A_{12} \quad A_{22} \quad A_{32} \cdots A_{1n} \cdots A_{mn}$$

Note that the first element of any array has a subscript of 1, not 0. An array name must normally always have a subscript. In certain cases the subscript may be omitted. An array name used without a subscript refers to the entire array.

2-4 EXPRESSIONS

Expressions are strings of operands separated by operators. Operands may be constants variables, or functions references. Operators may be unary or binary; i.e., they may operate on a single operand or on pairs of operands.

An expression may be classed as arithmetic, logical, or relational. It may contain subexpressions. A subexpression is an expression enclosed by parentheses. An expression is single value, that is, its evaluation has a unique result.

Arithmetic Expressions

An arithmetic expression is a sequence of operands (integer, real, double-precision, complex, constant, variable, or function references) connected by arithmetic operators.

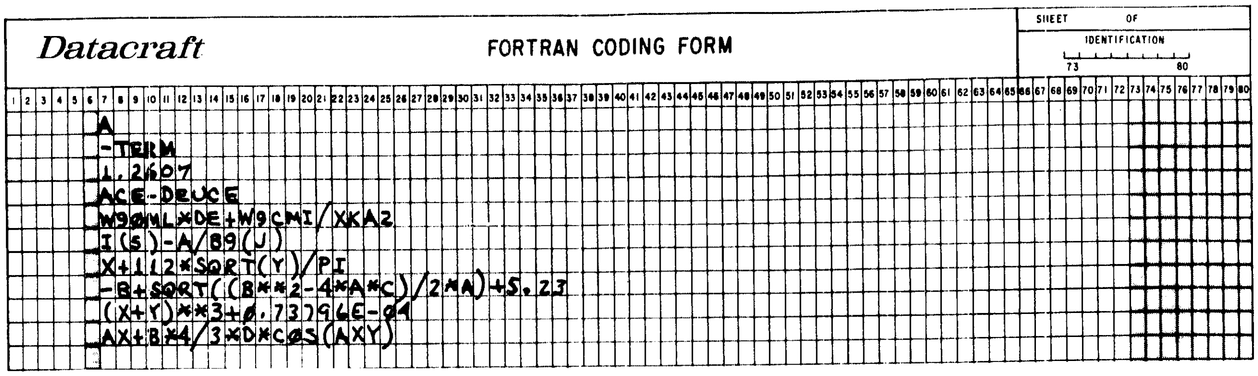
Arithmetic Operators

The arithmetic operators are:

Operator	Operation
+	Addition (binary) or Positive (unary)
-	Subtraction (binary) or Negative (Unary)
*	Multiplication
/	Division
**	Exponentiation

Arithmetic Operators (Cont'd.)

Some examples of arithmetic expressions are:



Datacraft FORTRAN CODING FORM

```
      A
      -TERM
      1.2607
      ACE-DEUCE
      W9ØML*DE+W9CMI/XKAZ
      I(S)-A/B9(J)
      X+112*SQRT(Y)/PI
      -B+SQRT((B**2-4*A*C)/2*A)+5.23
      (X+Y)**3+Ø.73796E-Ø4
      AX+B*4/3*D*CØS(AXY)
```

Evaluation Hierarchy

The evaluation hierarchy of arithmetic operators is as follows:

1. The innermost subexpression, followed by the next innermost subexpression, until all subexpressions have been evaluated.

2. The arithmetic operations, in the following order of precedence:

Operation	Operator	Order
Exponentiation	**	1 (highest)
Multiplication and Division	* /	2
Addition and Subtraction	+ -	3

Some additional conventions are necessary.

1. At any one level of evaluation, operations of the same order of precedence are evaluated from left to right. Consequently, I/J/K/L is equivalent to ((I/J)/K)/L

2. The sequence "operator operator" is not permissible. Therefore, A*-B must be expressed as A*(-B).

3. As in algebraic notation, parentheses are used to define evaluation sequences explicitly. Thus $\frac{A + B}{C}$ is written as (A+B)/C.

Evaluation Hierarchy (Cont'd.)

Example:

The expression A*(B+C*(D-E/(F+G)-H)+P(3)) is evaluated in the following sequence:

r_1 = F+G

r_2 = E/r_1

r_3 = D-r_2-H

r_4 = C * r_3

r_5 = B + r_4 + P(3)

r_6 = A * r_5

where the r_i indicates the levels of evaluation.

Mixed Expressions

Where an arithmetic expression contains elements of more than one type, it is known as a mixed expression. Integer, real, double precision, and complex elements may be mi› ·d in an arithmetic expression using any of the arithmetic operators EXCEPT exponentiation.

Allowable mixing using exponentiation (**) is shown in Table 2-1. The entries in the table give the variable type of the result, if allowed.

Table 2-1. Allowable Mixed-Mode Exponentiation

BASE ** EXPONENT = RESULT

Base	Exponent	Result	Comment
Real	Real	Real	
	Integer	Real	
	Double Precision	Double Precision	
	Complex	None	Fatal Compile Time Error
Double Precision	Real	Double Precision	
	Integer	Double Precision	
	Double Precision	Double Precision	
	Complex	None	Fatal Compile Time Error
Integer	Real	None	Fatal Compile Time Error
	Integer	Integer	
	Double Precision	None	Fatal Compile Time Error
	Complex	None	Fatal Compile Time Error

Mixed Expressions (Cont'd.)

Table 2-1. Allowable Mixed-Mode Exponentiation (Cont'd.)

Base	Exponent	Result	Comment
Complex	Real	Complex	
	Integer	Complex	
	Double Precision	Complex	Real Precision Result
	Complex	None	Fatal Compile-Time Error

Within a mixed expression, elements of lower precedence type are converted to the higher type before being combined with other elements. Thus, for example, (3/4) is an integer expression and has the value zero, while 3./4 is a real expression and has the value 0.75.

The following rules also apply to mixed expressions:

1. Expressions appearing as subscripts are independent of the expression in which the array appears. The subscript expressions are evaluated in their own mode and neither affect the mode of the outer expression nor are affected by it.

2. With the exception of certain basic functions, the same rule applies to expressions appearing as arguments as to those appearing as subscripts. They are always evaluated in their own mode; they may, however, affect the mode of the function result and thus of the expression in which the result is used. See Table 2-8, Library Functions.

3. Integer expressions that appear as exponents (i.e., to the right of an ** operator) are evaluated in their own mode; that is, integer.

4. Integer, real, and double-precision values that appear in complex expressions are assumed to have imaginary parts of zero.

5. Values of expressions, subexpressions, and elements may not exceed the limits associated with the mode of the expression.

Relational Expressions

The form of a relational expression is

$e_1 \ r \ e_2$

where e_1 and e_2 are arithmetic expressions and r is a relational operator.

Evaluations of relational expressions result in either of the two values "true" or "false", i.e., a logical value.

Relational Expressions (Cont'd.)

Relational operators cause comparisons between expressions.

Operator	Meaning
.LT.	$<$ Less than
.LE.	$\leq$ Less than or equal to
.EQ.	$=$ Equal to
.NE.	$\neq$ Not equal to
.GE.	$\geq$ Greater than or equal to
.GT.	$>$ Greater than

When two arithmetic expressions are compared, using a relational operator, the two expressions are first evaluated, each in its own mode, then the comparison is made in the mode of higher precedence.

If the mode of either or both of the expressions is complex, then the relational operator must be .EQ. or .NE. If it is not, then an error will result. Complex values are considered equal only if both the real and imaginary portions are equal.

Logical Expressions

Logical expressions are expressions of the form:

$e_1 \; c_1 \; e_2 \; c_2 \; e_3 \; c_3 . \; . \; . \; e_n$

where the e_i are logical elements and the c_i are the binary logical operators.

Evaluations of logical expressions result in either of the two values "true" or "false".

Logical elements are defined as one of the following entities.

1. A logical variable or function reference
2. A logical constant
3. A relational expression
4. Any of the above enclosed in parentheses
5. A logical expression enclosed in parentheses
6. Any of the above, preceded by the unary logical operator .NOT.

Logical Operators

Logical operators are listed and evaluated in Table 2-2. Table 2-3 is a truth table for the logical operators.

Table 2-2. Evaluation of Logical Operators

LOGICAL OPERATOR	OPERATOR TYPE	EXPRESSION	EXPRESSION EVALUATION
.NOT.	unary	.NOT. e	true only when e is false.
.AND.	binary	e_1 .AND. e_2	true only when both e_1 and e_2 are true.
.OR.	binary	e_1 .OR. e_2	true when either or both e_1 and e_2 are true.
.XOR.	binary	e_1 .XOR. e_2	true when either but not both e_1 and e_2 are true.

Table 2-3. Truth Table for Logical Operators

EXPRESSION CONDITION		EXPRESSION EVALUATION			
e_1	e_2	.NOT.e_1	e_1.AND.e_2	e_1.OR.e_2	e_1.XOR.e_2
True	True	False	True	True	False
True	False	False	False	True	True
False	True	True	False	True	True
False	False	True	False	False	False

Evaluation Hierarchy

The evaluation hierarchy for logical expressions are:

1. arithmetic expressions
2. relational expressions (The relational operators are all of equal precedence.)
3. The innermost logical subexpressions, followed by the next innermost logical sub-expression, etc.

2-4 EXPRESSIONS (CONT'D.)

Evaluation Hierarchy (Cont'd.)

4. the logical operations, in the following order of precedence:

Operator	Order
.NOT.	1 (highest)
.AND. .XOR.	2
.OR.	3

For example, the expression

L.OR. .NOT.M.AND. X.GE.Y

is interpreted as

L.OR. ((.NOT.M) .AND. (X.GE.Y))

Logical-Integer Operators

Harris FORTRAN allows integer typed variables and constants to be combined using the standard FORTRAN logical operators plus two additional operators which are defined for integer typed data only:

.SHIFT.

.ROTAT.

The .SHIFT. Operator

e.SHIFT.i

e is an integer arithmetic expression, and
i is an integer constant such that $-24 < i < 24$. The value of i specifies both the magnitude and direction of the shift. If i is positive, the direction of the shift is left, and if i is negative, the direction of the shift is right. This convention is adopted so that using a shift operator on a word is equivalent to multiplying the 24 bit unsigned word by 2^i.

The .SHIFT. Operator (Cont'd.)

Examples:

EXPRESSION	VALUE OF K	VALUE OF EXPRESSION
K.SHIFT.5	'77777777	'77777740
K.SHIFT.5	'03040506	'42024300
K.SHIFT.-5	'77777777	'01777777
K.SHIFT.-5	'03040506	'00061012

The .ROTAT. Operator

e.ROTAT.i

where:

i specifies both the magnitude and direction of rotation.

The value of i specifies both the magnitude and direction of rotation.

If i is positive, the direction of the rotation is left, and if i is negative, the direction of the rotation is right, as in the shift operator.

Examples:

EXPRESSION	VALUE OF K	VALUE OF EXPRESSION
K.ROTAT.6	'12345670	'34567012
K.ROTAT.6	'40506070	'50607040
K.ROTAT.-6	'12345670	'70123456
K.ROTAT.-6	'40506070	'70405060

When the logical operators, .AND., .OR., and .XOR. are used on integer data, these data are considered to be 24-bit machine words, and the effect of the operator is as follows:

I.AND.J	is the bit by bit logical intersection (and) of the words I and J.
I.OR.J	is the bit by bit logical union (or) of the words I and J.
I.XOR.J	is the bit by bit logical exclusive or of the words I and J.

2-4 EXPRESSIONS (CONT'D.)

Evaluation Hierarchy

The evaluation hierarchy for logical-integer operators is:

Operator	Order
.SHIFT. .ROTAT.	1 (highest)
.AND. .XOR.	2
.OR.	3

Note that the operator .NOT. is not included with the logical-integer operators. A logical complement of an integer value can be obtained by an exclusive or with a constant consisting of all ones; i.e., the logical (ones) complement of the integer I, can be obtained by the expression:

```
I.XOR. '77777777 or I.XOR.-1
```

Examples of the use of logical-integer operators for bit and byte manipulation and testing are:

1. Move the left byte from word J into the middle byte of word K without disturbing the left and right bytes of word K:

   ```
   K = (K.AND. '77600377) .OR. (J.SHIFT. -8.AND. '177400)
   ```

   The parentheses in example 1 are used for clarity, and are not necessary since the hierarchical order of the operators guarantee performance of the operations in the proper order.

2. Jump to statement number 750 if bit 20 of word J is on (i.e., contains a 1) otherwise go to statement number 500:

   ```
   If (J .AND. '4000000) 500, 500, 750
   ```

3. If bit 3 of word J is off, turn on bit 5 of word K:

   ```
   If ( (J .AND. '4) .EQ. 0) K = K .OR. '20
   ```

2-5 ASSIGNMENT STATEMENT

An assignment statement has the form:

v = e

where

v is a variable (a scalar or an array element of any type), and e is an expression of appropriate type (see Table 2-4).

The statement means, "assign to v the value of the expression e". The expression need not be the same type as the variable, though in practice it usually is. When it is not, the expression is evaluated in its own mode, independent of the type of the variable. Then, if permissible, it is converted to the type of the variable according to Table 2-4 and assigned to the variable.

Table 2-4. Expression Type for Mixed Variable Assignments

VARIABLE = EXPRESSION

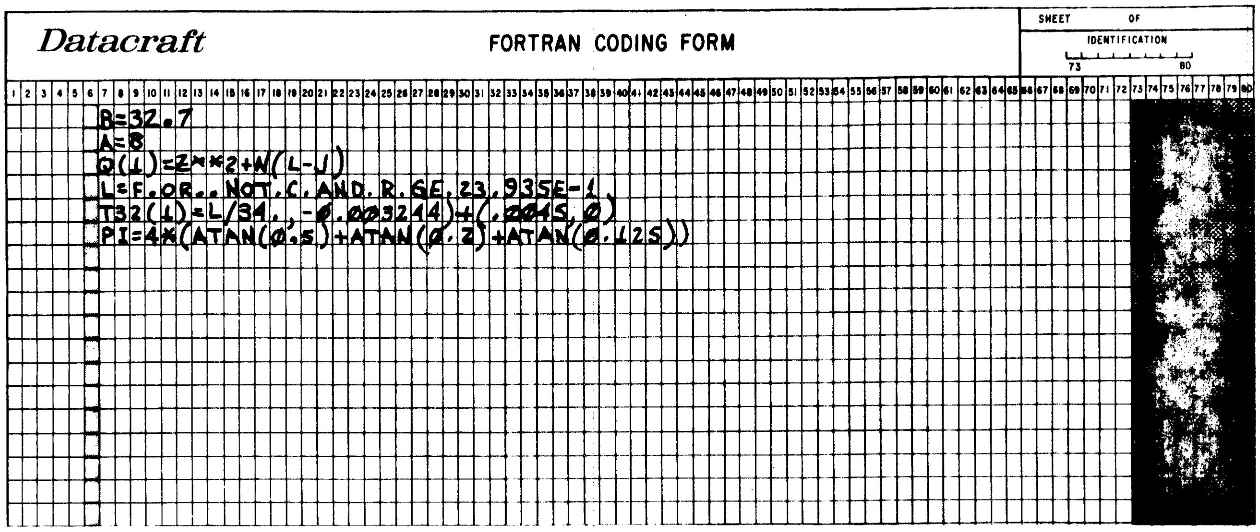
Variable	Expression Type	Result	Comment
Integer	Integer	Integer	
	Real	Integer (1)	
	Double Precision	Integer (1)	
	Complex	None	Fatal Compile Time Error
	Logical	None	Fatal Compile Time Error
Real	Integer	Real (2)	
	Real	Real	
	Double Precision	Real (3)	
	Complex	None	Fatal Compile Time Error
	Logical	None	Fatal Compile Time Error
Double Precision	Integer	Double Precision (2)	
	Real	Double Precision (3)	
	Double Precision	Double Precision	
	Complex	None	Fatal Compile Time Error
	Logical	None	Fatal Compile Time Error
Complex	Integer	Complex (2)	Imaginary Part Set to O. O
	Real	Complex	Imaginary Part Set to O. O
	Double Precision	Complex (3)	Imaginary Part Set to O. O
	Complex	Complex	
	Logical	None	Fatal Compile Time Error
Logical	Logical	Logical	
	All other expression cause fatal compile time errors.		

(1) Converted to integer and truncated such that $-8388608 \leq N \leq 8388607$

(2) Converted to appropriate type.

(3) Precision adjusted to fit type.

ASSIGNMENT STATEMENT EXAMPLES



Datacraft FORTRAN CODING FORM

SHEET OF

IDENTIFICATION 73 80

```
      B=32.7
      A=B
      Q(I)=Z**2+W(L-J)
      L=F.OR..NOT.C.AND.R.GE.23.935E-1
      T32(I)=L/34.,-Ø.ØØ3244)+(.ØØ45,Ø)
      PI=4*(ATAN(Ø.5)+ATAN(Ø.2)+ATAN(Ø.125))
```

2-6 CONTROL STATEMENTS

Each executable statement in a FORTRAN program is executed in the order of its appearance in the source program, unless this sequence is interrupted or modified by a control statement.

Labels

If program control is to be transferred to a particular statement, that statement must be identified. Statements are identified by labels, which are also called statement numbers. Statement numbers consist of up to five decimal digits; they must be greater than zero; embedded blanks and leading zeros are not significant.

Examples:

400 99999 756 1 00500

GO TO Statements

GO TO statements transfer control unconditionally from one point in a program to another. FORTRAN includes three forms of GO TO statements: unconditional, assigned, and computed.

Unconditional GO TO Statements

This statement has the form

GO TO k

where k is a statement number. The result of the execution of this statement is that the next statement executed is the statement whose label is k.

Example:

```
GO TO   502
98  X = Y
    .
    .
    .
502 A = B
    .
    .
    .
```

statement 502 will be executed immediately after the GO TO statement.

Assigned GO TO Statement

The format of the assigned GO TO statement is

GO TO v

or optionally,

GO TO v, $(k_1, k_2, k_3, \ldots, k_n)$

where:

v is an integer variable that has been assigned the location of a statement via an ASSIGN statement.

k_i is a statement number.

This statement transfers control to the statement whose location has been assigned to the variable v.

If the optional form is used (i.e., the list of k_i), each label appearing in the list must be defined in the program in which the GO TO statement appears (i.e., must be the label of a program statement). This form is provided for compatibility with other systems.

Assigned GO TO Statements (CONT'D.)

Examples:

```
ASSIGN 5371 to LOC
GO TO LOC
GO TO LOC, (117, 56, 101, 5371)
```

The two GO TO statements transfer control to the statement labeled 5371.

Computed GO TO Statement

The computed GO TO statement is expressed as

GO TO (k_1, k_2, k_3, . . . , k_n), i

where:

k_i is a statement label, and

i is an integer.

This statement causes control to be transferred to the statement whose label is k_j where j is the integer value of the variable i for $1 \leq j \leq n$.

Example:

STATEMENT	VARIABLE	TRANSFER TO
GO TO (98,12,405,3),N	3	405

ASSIGN Statement

The ASSIGN statement, used to assign a label to a variable, has the form

ASSIGN k TO v

where:

k is a statement label and
v is an integer variable.

Examples:

```
ASSIGN 5 TO JUMP
ASSIGN 222 TO M
```

Arithmetic IF Statements

The format for arithmetic IF statements is

IF (e) k_1, k_2, k_3

where:

e is an expression of integer, real, double-precision, or complex mode, and

k_1, k_2, and k_3 are statement labels.

The arithmetic IF statement is interpreted to mean

IF $e < 0$, GO TO k_1

IF $e = 0$, GO TO k_2

IF $e > 0$, GO TO k_3

Note that if e is a real or double precision expression, a test for exact zero may not be meaningful on a binary machine. If the expression involves any amount of computation, a very small number is more likely to result than an exact zero.

Examples:

Statement	Expression Value	Transfer To
IF(K) 1,2,3	47802	3
IF(3*M(J)-7)76,4,3	-6	76
IF(C(J,10)+A/4)23,12,12	0.0002	12
IF(K*N**2-14* LIMIT) 78, 444, 78	-1000	78
IF (Z-B-3. 1416 + SQRT(X-2))3,3,7	23.40669	7

If the expression is complex, then k_1 must be the same as k_3. Otherwise, an error will result. If k_1 is the same as k_3, then the branch to k_2 will only be taken if <u>both</u> the real and imaginary portions of the expression value are zero.

2-6 CONTROL STATEMENTS (CONT'D.)

Arithmetic IF Statements (CONT'D.)

It is also possible to omit one or two of the k_i's. In this case, if the condition associated with the omitted k_i occurs, then execution proceeds with the statement following the IF statement. If k_3 is omitted, then the trailing comma may also be omitted; if k_2 and k_3 are omitted, then both trailing commas may be omitted.

Examples:

```
IF ( . . . )     1,2,3
IF ( . . . )     ,2,3
IF ( . . . )     ,,3
IF ( . . . )     1,,3
IF ( . . . )     1,,
IF ( . . . )     1
IF ( . . . )     1,2
```

Logical IF Statement

The logical IF statement is represented as

IF (e) s

where:

e is a logical expression, and

s is any executable statement other than a DO or another logical IF.

The statement s is executed if the expression e has the value "true"; otherwise, the next executable statement following the logical IF statement is executed. The statement following the logical IF will be executed in any case after the statement s, unless the statement s causes a transfer.

Examples:

```
IF (FLAG .OR. L) GO TO 3135

IF (W .OR. N. LT. U/S + X3 (J,K)) R (J-8) = Q * ABS(X)

IF (OCTT * TRR .LT. 5.334E4) CALL  THERML(N,Y(L, 5))

IF (ITB.EQ.1.AND.NS.ROTAT.-1.LT.0) IF (N-1) 43, 53, 63
```

CALL Statement

The CALL statement is used to call or transfer control to a SUBROUTINE subprogram, SUBROUTINE ENTRY point or a FUNCTION ENTRY point (refer to "2-9 PROGRAMS AND SUBPROGRAMS - SUBROUTINE Subprograms") and may take either of the following forms:

CALL p

CALL p $(a_1, a_2, a_3, \ldots, a_n)$

where:

p is the identifier of the subroutine and a_i is an argument. Arguments may be constants, variables, expressions, statement labels, or array or subprogram names (refer to "2-9 PROGRAMS AND SUBPROGRAMS - Arguments and Dummies").

A subroutine is similar to a function except that it does not necessarily return a value and must not, therefore, be used in an expression. Also, a function must have at least one argument, a subroutine may have none. For example,

CALL CHECK

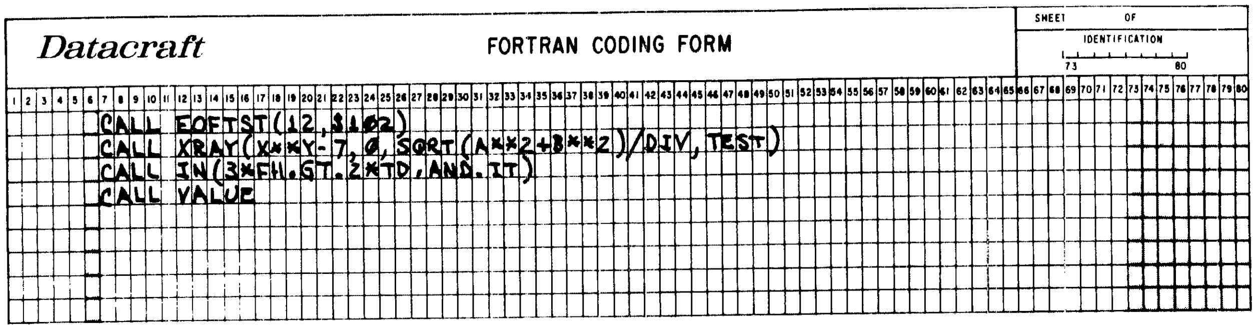
CALL Statements (Cont'd.)

Arguments that are scalars, array elements, or arrays may be modified by a subroutine effectively returning as many results as desired.

A complete discussion of the usage and forms of arguments to subprograms is contained in Section 2-9.

A subroutine name has no type (e.g., real, integer) associated with it; it merely identifies the block of instructions to be executed as a result of the CALL. Therefore, the appearance of a subprogram name in a CALL statement does not cause it to take on any implicit type.

Other examples of CALL statements are given below. (Note that statement labels are identified by being preceded by a dollar sign ($).


Datacraft FORTRAN CODING FORM

```
      CALL EOFTST(12,$1Ø2)
      CALL XRAY(X**Y-7,Ø,SQRT(A**2+B**2)/DIV,TEST)
      CALL IN(3*FH.GT.2*TD,AND.IT)
      CALL VALUE
```

RETURN Statement

The RETURN statement causes an exit from a subprogram. It takes the form:

RETURN

or

RETURN n (Extended Compiler Only)

Where n is an integer variable dummy (refer to Section 2-9, Arguments and Dummies) and n has been assigned a statement number by the calling program using an ASSIGN statement (Section 2-6) or n corresponds to a statement number argument (i.e., $mmmm). This form of the RETURN statement gives the user a non-normal return capability.

A RETURN statement must be logically the last statement executed in any subprogram;it need not be physically the last. There may be any number of RETURN statements in a subprogram. A RETURN statement in a main program will be treated as an error.

In a subroutine, the RETURN statement returns control from the subroutine to the first executable statement following the CALL statement that called the subroutine. In a function it causes the latest value assigned to the function name to be returned, as the function value, to the expression in which the function reference appeared (see Section 2-9, PROGRAMS AND SUB-PROGRAMS).

DO Statement

The DO statement may be written in two ways:

DO k v = i_1, i_2, i_3

DO k v = i_1, i_2

where

k is a statement label,

v is a scalar variable of integer type,

i_1, i_2, and i_3 are integer variables or constants.

Integers i_1 and i_2 must appear. If i_3 is not present, it is assumed to have the value 1.

A DO statement indicates that the block of statements following it are to be executed repetitively. Such a block is called a DO loop, and all statements within it, except for the opening DO statement, constitutes the range of the DO statement. The last statement in a DO loop is the terminus and bears the statement label k.

The execution of a DO loop proceeds in the following manner:

1. The variable v is assigned the value of i_1.

2. If the incremental value (i_3) is an integer variable or positive integer constant, the variable C is compared to the terminal value (i_2). If v is greater than i_2, control is passed to the statement following the one whose label is k. If v is less than or equal to i_2, continue to step 3.

   If the incremental value (i_3) is a negative integer constant, the variable v is compared to the terminal value (i_2). If v is less than i, control is passed to the statement following the one whose label is k. If v is greater than or equal to i_2, continue to step 3.

3. The range of statements is executed for one iteration.

4. The value of v is incremented by the value of i_3.

5. The process is repeated from Step 2.

The actual number iterations defined by the DO statements is given by

$$\max \left[\frac{i_2 - i_1}{i_3} \right] +1,0$$

where the brackets represent the largest integral value not exceeding the value of the expression.

DO Statement (Cont'd.)

Note that if conditions for termination are met initially, the entire range of the DO loop will not be executed.

The terminal statement of a DO range (i.e., the statement whose label is k) may be any executable statement other than one of the following:

DO statement

GO TO statement

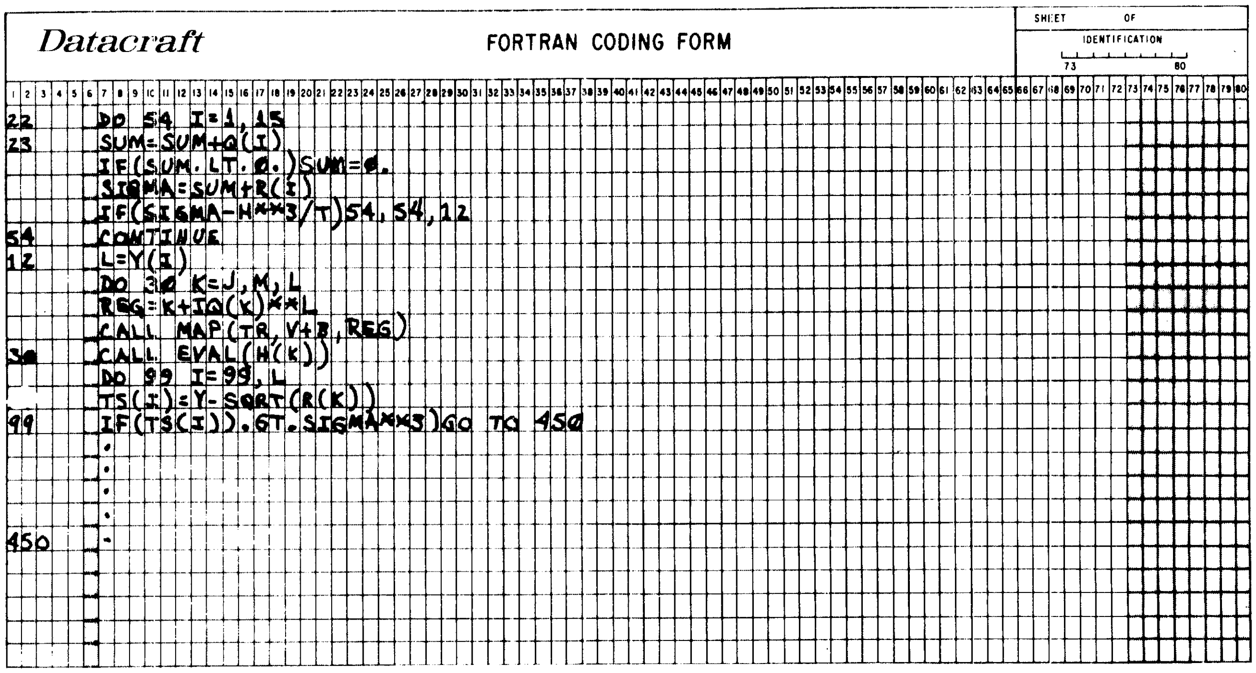
Arithmetic IF statement

RETURN statement

STOP statement

Note that logical IF statements are allowed as terminal statements of a DO range.

Examples:



Datacraft FORTRAN CODING FORM

SHEET OF

IDENTIFICATION 73 80

```
22    DO 54 I=1,15
23    SUM=SUM+Q(I)
      IF(SUM.LT.0.)SUM=0.
      SIGMA=SUM+R(I)
      IF(SIGMA-H**3/T)54,54,12
54    CONTINUE
12    L=Y(I)
      DO 30 K=J,M,L
      REG=K+IQ(K)**L
      CALL MAP(TR,V+B,REG)
30    CALL EVAL(H(K))
      DO 99 I=99,L
      TS(I)=Y-SQRT(R(K))
99    IF(TS(I)).GT.SIGMA**3)GO TO 450
      .
      .
      .
      .
450   -
```

In the example that begins with statement 22, the range of statements 23 through 54 will be executed 15 times, unless the arithmetic IF statement causes a transfer to statement 12. If all 15 iterations are completed, control is passed to statement 12 at the end of the fifteenth iteration

DO Statements (Cont'd.)

The value of the variable v appearing in a DO statement depends on the number of iterations completed. The value of v during any one iteration is

$$i_1 + (i - 1) * i_3$$

where i is the number of the current iteration, and i_1 and i_3 have the meanings discussed above. If a transfer is made out of the range of a DO before all iterations have been completed, the value of v will be that of the iteration during which the transfer occurred. However, should the entire number of iterations be executed, the value of v is

$$i_1 + n * i_3$$

where n is the total number of iterations specified by the DO statement.

Thus, in the example beginning with statement 22, if all iterations are completed, statement 12 will be equivalent to

```
12    L = Y (16)
```

However, if the arithmetic IF statement causes a transfer to statement 12 during the eighth iteration, the statement will mean

```
12    L = Y (8)
```

The value of the variable v may not be modified within the range of the DO, nor may it be modified by a subprogram called within the range of the DO.

```
5     DO 10 I = 2,-2,-1
      WRITE (6,1) I
1     FORMAT (X,I3)
10    CONTINUE
```

Will Print:

```
2
1
0
-1
-2
```

DO Statement (Cont'd.)

A transfer into the range of a DO may only occur if there has been a prior transfer out of the DO range. For example:

```
        DO 25 I = 1,9
        .
        .
        .
        GO TO 8605
        .
        .
        .
24      A = I/8
25      JGU = Y (I) **E
        .
        .
        .
8605    R = SIN (G(I)) + JSU
        .
        .
        .
8606    GO TO 24
```

is permissible; in fact, the statements 8605 through 8606 are considered part of the DO range.

The sequence

```
        GO TO 11
        .
        .
        .
        DO 32 J = 2,36,2
11      R(J) = 47.E-7*T(J)
32      T(J) = Q
        .
        .
        .
```

is not valid because no transfer could possibly occur out of the DO range.

A DO loop may include another DO loop. That is, DO loops may be nested; however, they may not be overlapped. In a nest of DO loops the same statement may be used as the terminal statement for any number of DO ranges; transfers to this statement may be made only from the innermost DO loop. There is no limit to the number of DO ranges that may be nested.

CONTINUE Statement

This statement is written as

```
CONTINUE
```

and must appear in that form. The CONTINUE statement does not cause the compiler to generate machine instructions, and consequently has no effect on a running program. The purpose of this statement is to allow the insertion of a label at any point in a program.

For example:

```
        DO 72 I = 1,10
        .
        .
        .
        IF (X **I+0.9999E-5) 72,72,88
72      CONTINUE
88      H(33)=T(3,R,L,E)/22.5
        .
        .
        .
```

CONTINUE statements are most often used as the terminal statement of a DO range, as in the above example.

PAUSE Statement

PAUSE statements are written as

```
PAUSE
PAUSE a
```

where

a is a set of 1 to 5 alphanumeric characters

This statement causes the program to cease execution temporarily, presumably for the purpose of allowing the computer operator to perform some specified action. The operator can signal the program to continue execution, beginning with the statement immediately after the PAUSE. (See HOLD CONDITIONS in the Operating Systems, General Specifications).

The literal constant, a, will be displayed on the operator communications device when the program pauses.

2-6 CONTROL STATEMENTS (CONT'D.)

STOP Statement

STOP statements are written in the form

STOP

STOP a

where:

a is a set of 1 to 5 alphanumeric characters.

This statement terminates the execution of a running program. A message indicating execution of a STOP statement and the literal a (if present) will be output to the list output device.

END Statement

An END statement is used to inform the FORTRAN compiler that the physical end of a program has occurred. The statement must appear in one of the forms:

END

END$

If control reaches an END statement, the effect is that of a STOP statement except that no message is output. An END statement may be labeled.

The following restriction applies to any statement that begins with the character string E N D:

If, at the end of any FORTRAN line (which may be a continuation line) the compiler has encountered only the characters E N D, the compiler assumes that the statement is an END statement and will act accordingly.

The END$ form is used by the FORTRAN compiler in the same manner as the DC 6024 Assembler, i.e., it is processed exactly like an end card after which all logical I/O files are closed and control is returned to the system.

2-7 INPUT/OUTPUT

FORTRAN input/output statements cause transmission of data between memory and peripheral devices at program execution time. These statements specify a logical file number that is associated with the peripheral device, and may contain an input/output list specifying the data to be transferred, or a reference to a FORMAT statement which controls conversion and editing of the transferred data.

I/O Lists

An input/output list defines the data that is to be processed by the input/output statement in which the list appears.

Simple Lists

A simple list has the form

$e_1, e_2, e_3, \ldots, e_4$

where each e_i may be

a scalar or array element,

an array name,

another input/output list enclosed in parentheses.

Note that the last item implies that input/output lists may be nested to any level; furthermore, redundant parentheses are permissible. However, parentheses are mandatory only to enclose DO-implied lists.

Examples:

```
A

MATRIX (25,L)

MATRIX, T

RY, Y(N,M), (X23A, HB, XKE)
```

When an unsubscripted array name appears in an input/output list, it refers to all of the elements in the array in storage order (see Section 2-3, DATA - Arrays in Storage). This means that the elements are accessed beginning with the lowest subscript value for each dimension and ending with the maximum subscript value for each dimension. In between the first subscript varies most rapidly and the last subscript varies least rapidly. This is also called "columnwise" ordering.

For example, if V is a 2x3x2 array, then the list item V is equivalent to the following elements in the order shown.

```
V(1,1,1)
V(2,1,1)
V(1,2,1)
V(2,2,1)
V(1,3,1)
V(2,3,1)
V(1,1,2)
V(2,1,2)
V(1,2,2)
V(2,2,2)
V(1,3,2)
V(2,3,2)
```

2-7 INPUT/OUTPUT (CONT'D.)

DO-Implied Lists

A DO-implied list is a simple list followed by a comma and then by a DO-control of the form

$v = i_1,i_2,i_3$

where

v is a DO-control integer variable,

i_1, i_2, and i_3 are DO-parameters as described in Section 2-6.

The meaning of a DO-control is similar to that of a DO statement, that is, all the items in the simple list preceding the DO-control are repeated over and over while v is incremented from i_1 to i_2 in steps of i_3.

A DO-implied list enclosed in parentheses becomes a simple list item. Thus, an input/output list may contain any number of nested DO-implied lists, with the provision that all the ne lists must be enclosed in parentheses.

Examples:

DO-implied lists	Equivalent Simple Lists
(X(I), I = 1,4)	X(1), X(2), X(3), X(4)
(A(J), B(J), J = 1,3)	A(1), B(1), A(2), B(2), A(3), B(3)
(G(2*N), N = 3,9,2)	G(6), G(10), G(14), G(18)
T,(C(J), J = 3,5), E,LENGTH	T,C(3), C(4), C(5), E, LENGTH
((A(I,J), I = 7,9), J = 1,3)	A(7,1), A(8,1), A(9,1), A(7,2), A(8,2), A(9,2), A(7,3), A(8,3), A(9,3)
(R,T(K), K = 2,3)	R, T(2), R, T(3)

The DO-control variable in an input/output list is available and may also be used as a list item. The output list

```
(K, A(K), K = 1, 3), G(K)
```

is equivalent to the simple list

```
1, A(1),2,A(2), 3,A(3), G(4)
```

In Harris FORTRAN, a DO-implied list functions in the same manner as a DO statemen If the terminal conditions of a loop are met initially, it will be done "no times". Thus, the lists

2-7 INPUT/OUTPUT (CONT'D.)

I/O List - DO-Implied Lists (Cont'd.)

```
(G(K), K = 34, 22)
(J, X(J), J = 10,9,1)
```

will not cause data to be read or written.

Free Format READ/WRITE

The free format I/O statements permit I/O operation within defined limits. Data conversion takes place based on the variable type encountered in the list.

There are three statement forms available for use with the free format feature. They are:

```
READ(u,-)   list
WRITE(u,-)

READ(u,)    list
WRITE(u,)

READ,                 (implied unit is 7)
PRINT,      list      (implied unit is 6)
PUNCH,                (implied unit is 8)
```

Data to be read under free format consists of one or more records. Data values in the input stream must be separated from each other by a comma and/or one or more blanks. Data will be read until the list is satisfied.

The input data must be of the same type as the variable. Therefore, an integer must not contain a decimal point and a logical variable must begin with a "T" to set the value TRUE and any other character to set the value FALSE. Real, double precision and complex data must be input in any of the forms acceptable under a formatted READ statement and if an exponent is present in the data it must begin with an E or a D. The decimal point is optional and, if it is omitted, it is assumed to be to the right of the last digit.

A printed line of free-format consists of 120 print positions. Since the formats used for free-format output provide for at least two leading blanks, column 1 of the output line will always be blank. Thus, free-format output will be single spaced. Additional carriage control must be provided by formatted output if it is desired.

The output field widths of the various variable types are as follows:

Integer -- I10

Logical -- L3

Real -- 1PE14.5

Double precision -- 1PD19.10

Complex -- 1P2E14.5

2-7 INPUT/OUTPUT (CONT'D.)

Free Format READ/WRITE (Cont'd.)

Note: The output accuracy for real, double precision and complex values is such that all full digits of accuracy for the value are output. Partial digits (least significant bits) are not output. This avoids output of numbers such as 1.9999999999, where the value is effectively 2.0000.

If the current line will not accommodate the list variable field width to be printed, a new line will be started, e.g., five complex values cannot be printed on the same line, since the total field width required is 140 columns. Therefore, only four complex variables will be printed on the line.

The definition of the abbreviations used in the examples which follow are:

D = Double Precision Variable

C = Complex Variable

X = Real Variable

L = Logical Variable

I = Integer Variable

Example: The statement

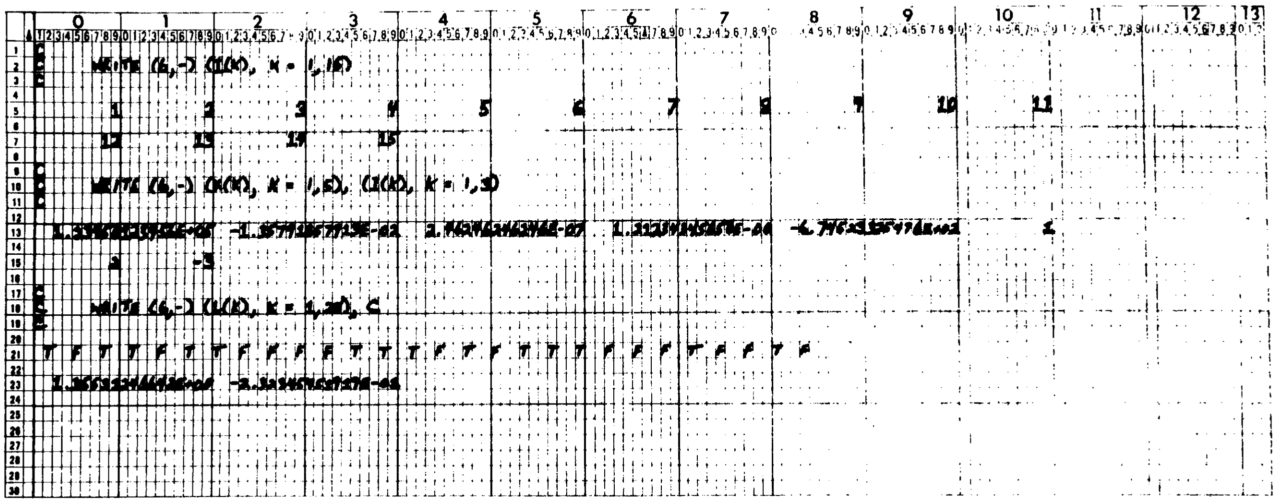
```
READ (7,) (D(J), C(J), X(J), L(J), I(J), J = 1,3)
```

will read any of the following data sets:

```
1.   1.2,2.,3.,3.1,F,1,1.,6.,7.,4.,T,2,6.1,2.2,3.3,4.4,FALSE,3

2.   1.0D0  0.0,3., 7.0, TRUE, 44
     -3.14D1  11.692,4.52, 3.81, FALSE, 88000
     1.23E17  5.65  2.03  1.1198, TO, -786

3.   1.2
     3.4
     5.6
     7.8,T,25
     13.8, 88.8, 91.6, 17.8
     THISISATRUESTATEMENT    5
     -900000000, .000001
     -.0000023, 1.5, FALSE
     8388607
```

The coding sheet which follows shows several examples of WRITE statements and the resulting output format. All values shown are arbitrarily chosen for the sake of example.

Free Format READ/WRITE (Cont'd.)



```
C
C       WRITE (6,-) (I(K), K = 1,15)
C

          1        2        3        4        5        6        7        8        9        10       11

          12       13       14       15

C
C       WRITE (6,-) (A(K), K = 1,5), (I(K), K = 1,3)
C

  1.[illegible]E-05  -1.[illegible]E-02  2.[illegible]E-07  1.[illegible]E-04  -6.[illegible]E+02  2

          2       -3

C
C       WRITE (6,-) (L(K), K = 1,20), C
C

 T  F  T  T  F  T  T  F  F  F  F  T  T  T  F  T  F  T  T  T  F  F  F  T  F  F  T  F

  1.[illegible]E-06  -2.[illegible]E-01
```

READ Statement

The READ statement forms are:

READ f READ f,k READ (L,f)k READ (L,f) READ (L,f,END=S1,ERR=S2) k	formatted
READ (L) k READ (L) READ (L,END=S1,ERR=S2)k	unformatted

where:

L is an integer variable or constant specifying a logical file. The logical file for the card reader (LFN 7) is implied for the forms READ f and READ f,k.

f is the statement label of a FORMAT statement or a variable FORMAT specification (see FORMAT - Stored in Arrays), and

k is an Input/Output list.

READ Statement (Cont'd.)

S1 is the statement number to be executed when an End-of-File is detected.

S2 is the statement number to be executed on a format error.

Either or both parameters may be present in any order.

The formatted form of the READ statement causes external symbolic data to be read and converted into internal form under control of the FORMAT statement specified by f. If no list is specified, the READ statement may cause a record to be skipped, or cause data to be read directly into the FORMAT statement.

The unformatted form of the READ statement causes external binary data to be read and placed directly into the locations specified by the list, k. If no list is specified, the unformatted read has the effect of skipping a record.

WRITE Statement

The Write statement forms are:

```
PRINT f         ⎫
PRINT f,k       ⎪
PUNCH f         ⎬   formatted
PUNCH f,k       ⎪
WRITE (L,f)k    ⎪
WRITE (L,f)     ⎭

WRITE (L)k      }   unformatted
```

where:

L, f, and k have the same meaning as in the READ statement. The logical file for the line printer (LFN 6) is implied for the PRINT statements, also the logical file for the card punch (LFN 8) is implied for the PUNCH statements.

The formatted form of the WRITE statement causes internal data to be converted and output under control of the FORMAT statement specified by f. If no list is specified, data may be directly output from the FORMAT statement.

The unformatted form of the WRITE statement causes internal binary data located in locations specified by the list, k, to be directly output to the logical file, L. Note that the unformatted form of the WRITE statement must specify a list.

FORMAT Statement

The FORMAT statement is used to specify the conversion to be performed on data being transmitted during formatted input/output operations. It is nonexecutable and may be placed anywhere in the program. In general, conversion performed during output is the reverse of that performed during input. FORMAT statements are expressed as

FORMAT $(S_1, S_2, S_3, \ldots, S_n)$

2-7 INPUT/OUTPUT (CONT'D.)

FORMAT Statement (Cont'd.)

where

S_i is either a format specification of one of the forms described below or a repeated group of such specifications in the form

$r(S_1, S_2, S_3, \ldots, S_n)$

where

r is a repeat count as described below, and

S_i is as described above; in other words, repetitions may be nested. The commas between the S_i are mandatory, except when the S_i is of the X of H form, in which case the comma is optional.

Every FORMAT statement should be labeled so that references may be made to it by formatted input/output statements. An entire FORMAT (the parentheses and the items they enclose) may be stored in an array variable. In this case, the array itself is referenced by the input/output statements (see FORMATS Stored in Arrays, Section 2-7, page 2-51).

Format specifications describe the type of conversion to be performed, specific data to be generated, scaling of data values, and editing to be executed. Each integer, real, double precision, or logical datum appearing in an input/output list is processed by a single format specification, while complex data are operated on by two consecutive format specifications. Format specifications may be any of the following forms:

rFw.d	rIw	nHs	wX
rEw.d	rLw	'S'	Tw
rDw.d	rAw	"S"	iP
rGw.d	rOw		
	rRw		

where:

The characters F, E, D, G, I, L, A, H, X, T, P, and (/) define the type of conversion, data generation, scaling, editing, and FORMAT control.

r is an optional, unsigned integer that indicates that the specification is to be repeated r times. When r is omitted, its value is assumed to be 1.

w is an unsigned integer that defines width in characters (including digits, decimal points, algebraic signs, and blanks) of the external representation of the data being processed.

d for F, E, D, and G input specifications, d is an unsigned integer that specifies the number of fractional digits appearing in the magnitude portion of the external field.

2-7 INPUT/OUTPUT (CONT'D.)

FORMAT Statement (Cont'd.)

For G output specification, d is also an unsigned integer; but in this context it is used t define the number of significant digits that appear in the external field. Therrefore, its value should not be zero.

n is an unsigned, decimal integer that defines the number of characters being processed

i is a signed integer (plus signs are optional). The function of i is described under X and P specifications.

s is a character string (see H format)

F Format (Fixed Decimal Point)

F format specifications are expressed as:

rFw.d

Real, double precision, or either part of complex data may be processed by this form of conversion. Double-precision values are converted with full precision if sufficient width is specified by w, and the value of d allows for the appropriate number of digits in the fractional portion of the field.

OUTPUT - Internal values are converted to real constants, rounded to d decimal place with an overall length of w. The field is right justified with as many leading blanks as necessai Negative values are preceded with a minus sign. Consequently, for the specification F11.4,

273.4	is converted to	273.4000
7	is converted to	7.0000
-.003	is converted to	-.0030
-442.30416	is converted to	-442.3042

If a value requires more positions than are allowed by the magnitude of w, the entire output field will be filled with asterisks. If an integer or logical value is output with this format, the entire output field will be filled with question marks.

In order to insure that such a loss of digits does not occur, the following relation must hold true:

$w \geq d+2+n$

where n is the number of digits to the left of the decimal point.

2-7 INPUT/OUTPUT (CONT'D.)

F Format (Cont'd.)

INPUT - Input strings may take any of the integer, real, or double-precision constant forms discussed under "Numeric Input Strings". Each string will be of length w with d characters in the fractional portion of the value. If a decimal point is present in the input string, the value of d is ignored, and the number of digits in the fractional portion of the value will be explicitly defined by that decimal point. F-input data may optionally have exponent specified. For the specification F10. 3,

33	is converted to	.033
902142	is converted to	902. 142
. 34562	is converted to	. 34562
-7. 001	is converted to	-7. 001
2. 3E-2	is converted to	.023

NOTE

During F format input, any blanks within the input field will be processed as zeros. Thus, "1 . 2" will be interpreted as 10. 02. This may be overridden so that blanks are totally ignored thru an execution time flag. Under VULCAN, this is done with a Vulcanizer MODE IB statement. Under DMS/ROS/TOS/DOS, this is done by setting option 16 at execution time.

E Format

E format specifications are expressed as:

rEw. d

Real, double-precision, or either part of complex data may be processed by this form of conversion. Double-precision values are converted with full precision if sufficient width is specified by w and the value of d allows for the appropriate number of digits in the fractional portion of the field.

OUTPUT - Internal values are converted to real constants of the forms

±. dddd. . . dE±ee

where dddd. . . d represents d digits and E±ee is interpreted as a multiplier of the form:

$10^{\pm ee}$

The leading sign (before the decimal point) is omitted if positive.

2-7 INPUT/OUTPUT (CONT'D.)

E Format (Cont'd.)

Internal values are rounded to d digits, and negative values are preceded by a minus sign. The external field is right justified and preceded by the appropriate number of blanks. The following are examples for the specification E15.8:

90.4450	is converted to	0.90445000E+02
-435739015.	is converted to	-0.43573902E+09
.000375	is converted to	0.37500000E-03
-1	is converted to	-0.10000000E+01
.2	is converted to	0.20000000E+00
0.0	is converted to	0.00000000E+00

The field includes the exponent digits, an exponent sign, the letter E, the mantissa digits, the decimal point, and the sign of the value (minus or space). If an integer or logical value is output with this format, the entire output field will be filled with question marks. If a value requires more positions than are allowed by the magnitude of w, the entire output field will be filled with asterisks. To prevent this from occurring, the value for w should be at least 6 more than the value for d.

$w \geq d+7$

is satisfied by the specification.

INPUT - Forms permissible for strings of input characters are discussed under the headin "Numeric Input Strings" in Section 2-7. Conversion is identical to F format conversion. In particular, input fields for conversion in E format need not have exponents specified.

Examples:

Input Value	Specification	Converted to
-113409E2	E11.6	-11.340900
849935E-02	E10.5	.0849935
23.5+2	E8.1	2350.0

First, the decimal point is positioned according to the specification; then, the value of the exponent is applied to determine the actual position of the decimal point. In the first example, -113409E2 with a specification of E11.6 is interpreted as -.113409E02; which, when evaluated (i.e., -.113409 x 10^2), becomes -11.340900.

D Format

D format specifications are expressed as:

rDw.d

2-7 INPUT/OUTPUT (CONT'D.)

D Format (Cont'd.)

OUTPUT - This format is similar to E format, with the exception that for output, the character D will be present instead of the character E. For example,

for E12.6, -667.334 is converted to -.667334E 03

and

D12.6, -667.334 is converted to -.667334D 03.

INPUT - Input under D format is the same as for E and F formats.

G Format

G format specifications are expressed as:

rGw.d

OUTPUT - The method of representation in the external output string is a function of the magnitude of the value being converted. The following table shows the correspondence between the magnitude of the value and the equivalent method of conversion that will be effected:

Magnitude of Datum	Equivalent Conversion Effected
$0.1 \leq N\ 10^{(w-d-4)}$	F(w-4).d, 4X
Otherwise	Ew.d

Note that the effect of the scale factor (see P Specification) is suspended unless the magnitude of the datum to be converted is outside of the range that permits effective use of F conversion.

INPUT - The G input conversion is identical to the F input conversion. The numeric field descriptor Gw.d indicates that the external field occupies w positions with d significant digits. The value of the list item appears internally as a real or double precision value.

I Format (Integer)

I format specifications are expressed as:

rIw

OUTPUT - Internal values are converted to integer constants. The integers may contain as many digits as are specified by w (or w-1 if negative). Negative values are preceded by a minus sign, and the field will be right justified and preceded by the appropriate number of blanks.

If a value requires more positions than are allowed by the magnitude of w, the entire output field will be filled with asterisks. If the value being output with this format is not an integer, the entire output field will be filled with question marks.

2-7 INPUT/OUTPUT (CONT'D.)

I Format (Cont'd.)

INPUT - External input strings must take the form of an integer constant or signed integer constant in the external input field.

NOTE

During I format input, any blanks within the input field will be processed as zeros. Thus "1 2 3" will be interpreted as 10203. This may be overridden so that blanks are totally ignored thru an execution time flag. Under VULCAN, this is done with a Vulcanizer MODE IB statement. Under DMS/ROS/TOS/DOS, this is done by setting option 16 at execution time.

O Format (Octal)

O format specifications are expressed as:

rOw

OUTPUT - Internal values are output as octal constants. The numbers may contain as many digits as specified by w. The field will be filled with zeros up to 8. Over 8, the field is right justified and filled with blanks. If a value requires more positions than are allowed by the magnitude of w, the entire output field will be filled with asterisks.

INPUT - The input string must be less than 8 integer digits from zero (0) to seven (7), inclusive.

NOTE

During O format input, any blanks within the input field will be processed as zeros. Thus, "1 2 3" will be interpreted as '10203. This may be overridden so that blanks are totally ignored thru an execution time flag. Under VULCAN, this is done with a Vulcanizer MODE IB statement. Under DMS/ROS/TOS/DOS, this is done by setting option 16 at execution time.

L Format

L format specifications are expressed as:

rLw

Only logical data may be processed with this format specification.

L Format

L format specifications are expressed as:

rLw

Only logical data may be processed with this format specification.

OUTPUT - Logical values are converted to either a T or an F character for the values "true" and "false", respectively. The T and F characters are preceded by w-1 blanks.

For example, using the specification L4,

.TRUE.	is converted to	b̸b̸b̸T
.FALSE.	is converted to	b̸b̸b̸F

where b̸ represents the character blank.

If the value being output with this format is not logical, the entire output field will be filled with question marks.

INPUT - The first non-blank character encountered in the next w characters determines whether the value is "true" or "false". If the first non-blank character is a "T" the value is "True". If the first non-blank character is not a "T", the value is "false". For example, the following input fields, processed by an L7 format, have the indicated values:

True	False
T	F
TRUE	FALSE
T42	(blank)
TILT	ZILCH

A Format

A format specifications are expressed as:

rAw

where r is the repeat specification and w is the external field width. A-format data is represented internally as 8 bit ASCII characters, packed three to a word. Hence, the maximum number of characters that can be stored in an integer element is 3, and in a real or double precision element is 6. However, the field width, w, may be greater or less than these maximum internal values. The following tables show how conversion is accomplished.

A Format (Cont'd.)

OUTPUT - The output consists of the following:

Internal Representation		Format	External Field	
Integer	Real	Spec.	Integer	Real
ABC	ABCDEF	A1	A	A
ABC	ABCDEF	A2	AB	AB
ABC	ABCDEF	A3	ABC	ABC
ABC	ABCDEF	A4	bABC	ABCD
ABC	ABCDEF	A5	bbABC	ABCDE
ABC	ABCDEF	A6	bbbABC	ABCDEF
ABC	ABCDEF	A7	bbbbABC	bABCDEF

INPUT - The input consists of the following:

External		Internal Representation	
Field	Format Spec.	Integer	Real
ABCDEFG	A1	A	A
ABCDEFG	A2	AB	AB
ABCDEFG	A3	ABC	ABC
ABCDEFG	A4	BCD	ABCD
ABCDEFG	A5	CDE	ABCDE
ABCDEFG	A6	DEF	ABCDEF
ABCDEFG	A7	EFG	BCDEFG

Example:

```
   DIMENSION B(2)

   READ (7,1) B,I

1  FORMAT (1A6,A2,I2)

   WRITE (6,2), B,I

2  FORMAT (1X,2A6,I2)
```

card input starting in column 1

```
ABCDEFGH12
```

print out

```
ABCDEFGHb̸b̸b̸b̸12
```

A Format (Cont'd.)

If an array name is used in the I/O list, the total number of words may be used to contain A format data. For instance, if M is an integer array with dimensions M(4), then an A specification of 4A3 may be used to read or write 12 consecutive characters from that array.

R Format

The R specification is similar to the A formatting specification. If the width specification for the R format is not less than the number of characters which can be packed in an element (3 for integer, 6 for real/double precision), then the R format is identical to the A format.

If the R format width specification is less than the maximum number of characters that can be packed in the element, then the number of characters specified are processed right-justified within the element. On input leading characters are filled with binary zeros; on output, they are ignored.

Example:

External field	Format width	A format input	R format input
ABC	3	'20241103	'20241103
ABC	2	'20241040	'00040502
ABC	1	'20220040	'00000101

where the characters have the following octal representations:

"A"	=	'101
"B"	=	'102
"C"	=	'103
" "	=	'040

H Format (Hollerith)

H format specifications are expressed as

nHs

2-7 INPUT/OUTPUT (CONT'D.)

H Format (Hollerith) (Cont'd.)

OUTPUT - The n characters in the string s are transmitted to the external record. For instance,

Specification	External String
1 HE	E
8Hb̸b̸VALUE:	b̸b̸VALUE:
5H$3.95	$3.95
9HX(2,5)b̸=b̸	X(2,5)b̸=b̸

where b̸ represents the character blank.

Care should be taken that the character string s contains exactly n characters, so that the desired external field will be created, and so that characters from other format specifications are not used as part of the string.

INPUT - The n characters in the string s are replaced by the next n characters from the input record. This replacement occurs as shown in the following examples:

Specification	Input String	Resultant Specification
3H123	ABC	3HABC
10HNOWb̸ISb̸THE	b̸TIMEb̸FORb̸	10Hb̸TIMEb̸FORb̸
5HTRUEb̸	FALSE	5HFALSE
6Hb̸b̸b̸b̸b̸b̸	RANDOM	6HRANDOM

where

b̸ = the character blank.

This feature can be used to change the titles, dates, column headings, etc., that are to appear on an output record generated by the H specification.

Harris FORTRAN provides an alternate to the H specification for outputting alphanumeric data. Literal strings are accepted within Format statements and are only accepted for output statements. The literal string takes either of the following forms:

"s"

's'

2-7 INPUT/OUTPUT (CONT'D.)

H Format (Hollerith) (Cont'd.)

where s is a string of ASCII characters. When enclosed in quotation marks ("), or apostrophes ('), a quotation mark or apostrophe may be represented by two successive quotation marks or apostrophes, respectively.

Examples:

```
"THIS IS A 'LITERAL STRING' "

'THIS IS A "LITERAL STRING"'

' ' ' '  is equivalent to  " ' "
```

X Specification (Skip)

X specifications are expressed as:

wX

This specification causes no conversions to occur. Instead, it causes w positions of the external field to be skipped or ignored.

OUTPUT - The next w positions in the output record will be blanks.

INPUT - The next w characters from the input string are ignored (that is, they are skipped).

For example, with the specification

F5. 3, 6X, I3

and the input string

76.41IGNORE697

the characters

IGNORE

will not be processed.

The field width specified for the "X" specification may be negative. In that case, then next column from or to which data is to be transferred will be the specified number of columns "backwards" of the current column. This specification can be used similarly to the "T" specification (see next section). Note however, that the negative "X" specification references a column number relative from the current column while the "T" specification references an absolute column number.

T Specification (Tab)

The T specification has the form:

Tw

where:

w specifies a character position within the input or output record.

The T specification causes no transfer of data, but merely resets the character position at which the next processing will begin. For example, the following three FORMAT statements are equivalent:

```
1       FORMAT (10X,F10.3,10X,I10)
2       FORMAT (T11,F10.3,T31,I10)
3       FORMAT (T31,I10,T11,F10.3)
```

Note that it is permissible to tab either forward or backward.

P Specification (Scale Factor or Power of 10)

The P specification has the form:

iP

where $-37 \le P \le 37$.

The P specification causes the value of the scale factor to be set to i, where the scale factor is treated as a multiplier of the forms

10^i for output, and

10^{-i} for input.

The scale factor is set to zero at the beginning of each formatted input/output operation. Once a scale factor has been encountered it applies to all subsequently encountered F, E, D and G field descriptions. Any number of P specifications may be present in a FORMAT statement, causing the value of the scale factor to be changed several times during a formatted input/output operation. If a FORMAT is re-scanned within a single input/output operation due to the number of items in a list (see Section 2-7 "FORMAT and List Interfacing"), the value of the scale factor is not reset to zero.

OUTPUT - The value of the list item is scaled by the multiplier 10^i. This causes the decimal point to be shifted right i places. On D- and E- type conversions, the exponent field (±ee) is correspondingly reduced by i. Thus, for D- and E-type output, the external number is equal to the internal value (except for rounding), while for F format output is not equal to the internal value (unless i is 0). The following examples illustrate output scaling:

P Specification (Scale Factor or Power of 10) (Cont'd.)

Format	External Field when Internal Value is:	
	2.71828	-2.71828
-2PF10.3	.027	-.027
-1PF10.3	.272	-.272
F10.3	2.718	-2.718
1PF10.3	27.183	-27.183
2PF10.3	271.828	-271.828
-2PE14.3	0.003E 3	-0.003E 3
-1PE14.3	0.027E 2	-0.027E 2
E14.3	0.272E 1	-0.272E 1
1PE14.3	2.718E 0	-2.718E 0
2PE14.3	27.183E -1	-27.183E -1

The examples for E conversion above are similar to those that would result from D conversion and E-type G conversion. When G conversion uses the F form, however, scale factors do not apply. Thus, a number output in G format always represents the internal value.

Note that when a scale factor is in effect, output rounding takes place after the scaling has been performed.

INPUT - During F, E, D, and G input conversions, if the input string contains an exponent field, the scale factor has no effect. However, when the input string does not contain an exponent field, the value of the external field is scaled by 10^{-i}; that is, the decimal point is moved left i places. The following examples indicate the effect of scaling during an input operation:

External Field	Scale Factor	Effective Value
-71.436	0P	-71.436
	3P	-.071436
	-1P	-714.36
-71.436E 00	3P	-71.436
	-1P	-71.436

It can be seen that, on both input and output, if the external number has an exponent specified, it is equal to the internal value; if it does not, then

external value = internal value x 10^i

Once a scale factor has been established during an input/output operation, it remains in effect throughout the operation, unless redefined by an additional P specification. To reset the scale factor to zero, it is necessary to write a 0P specification.

/Specification (Record Separator)

The form of the / specification is

/

Each slash (/) specified causes another record to be processed. In the case of contiguous slash specifications (i.e., ////.../), since no conversion occurs between each of the slash specifications, records are ignored during input, and blank records are generated during output operations.

OUTPUT - Whenever a slash specification is encountered, the current record being processed is output, and another record is begun. If no conversion has been performed when the slash is encountered, a blank record is created.

INPUT - The effect of slash specifications during input operations is similar to the effect for output, except that for input, records are ignored in the cases where blank records are created during output.

Parenthesized Format Specifications

Within a FORMAT statement any number of specifications may be repeated by enclosing them in parentheses, preceded by an optional repeat count, in the form shown below.

$r(S_1,S_2,S_3...,S_m)$

where

r is an optional, unsigned integer that indicates that the specification is to be repeated r times. When r is omitted, its value is assumed to be 1,

S_i is a FORMAT specification and

$m > 0$.

For example, the statement

3 FORMAT (3(A3,F6.2,3X), 3I2)

is equivalent to

3 FORMAT (A3, F6.2, 3X, A3, F6.2, 3X, A3, F6.2, 3X, 3I2)

There is no limit to the number of repetitions of this form that can be present in a FORMAT statement.

During input/output processing each repetitive specification is exhausted in turn, as is each singular specification.

The following are additional examples of repetitive specifications:

Parenthesized Format Specification (Cont'd.)

```
34 FORMAT (4X, 2(A8, X, 7G, 6.3), I4, 3 (D12.4,L5))

8 FORMAT (2(I8, 2(3X, F12.9), F12.9), A16)
```

In the latter example, repetitions are nested. Nesting of this type is permissible to a depth of ten levels.

The presence of parenthesized groups within a FORMAT statement affects the manner in which the FORMAT is re-scanned if more list items are specified than are processed the first time through the FORMAT statement. In particular, when one or more such groups have appeared, the rescan begins with the group whose right parenthesis was the last one encountered prior to the final right parenthesis of the FORMAT statement. A more complete discussion of this process is contained in Section 2-7, INPUT/OUTPUT STATEMENTS - Format and List Interfacing.

Numeric Input Strings

The permissible kinds of input strings that may be processed by numeric conversions are exactly the same for F, E, D, and G conversion. Any field that can be read using one of these formats can be read using any of the others. In other words, numbers for input with E format need not have exponents, and numbers input with F format may have exponents.

A numeric input string consists of a string of digits with or without a leading sign, a decimal point, and/or a trailing exponent. An exponent is normally specified as

E±e

where the plus sign is optional and e is a one- or two-digit number. The form ±e is also accepted (without the E), in which case the plus sign is not optional. Thus, a variety of forms may be used to express data for numeric input:

±n	±n.m	±n	±.m
±nE±e	±n.mE±e	±n.E±e	±.mE±e
±n±e	±n.m±e	±n.±e	±.m±e

where the plus signs are optional except in an exponent field without an E.

A D may be substituted for the E in an exponent field, with no change in meaning or value. It is not necessary to indicate that data is double precision, nor is it necessary to use a D format. Regardless of the format used or the form exponent (if any), a numeric string will be converted with full double precision if the input list item to which it is to be assigned is double precision.

Any numeric type of list item may be used with any numeric type of format specification, with the exception of I format. In this case the input field must contain an integer constant, refer to Section 2-3, DATA - Constants.

Leading, embedded and trailing blanks are ignored. The field terminates only when the width specification is exhausted.

FORMAT and List Interfacing

Formatted input/output operations are controlled by the FORMAT requested by each READ or WRITE statement. Each time a formatted READ or WRITE statement is executed, control is passed to the FORMAT processor. The FORMAT processor operates in the following manner:

1. When control is initially received, a new input record is read, or construction of a new output record is begun.

2. Subsequent records are started only after a slash specification has been processed (and the preceding record has been terminated) or the final right parenthesis of the FORMAT has been sensed. Attempting to read (or write) more characters on a record than are (or can be) physically present does not cause a new record to be begun; on output the extra characters are lost, on input they are treated as blanks.

3. During an input operation, processing of an input record is terminated whenever a slash specification or the final right parenthesis of the FORMAT is sensed, or when the FORMAT processor requests an item from the list and no list items remain to be processed. Construction of an output record terminates, and the record is written on occurrence of the same conditions.

4. Every time a conversion specification (i.e., F, E, D, G, I, L, or A specification) is to be processed, the FORMAT processor requests a list item. If one or more items remain in the list, the processor performs the appropriate conversion and proceeds with the next field specification. (If conversion is not possible because of a conflict between a specification and a data type, an error occurs.) If the next specification is one that does not require a list item (i.e., H,X,P,T, or /), it is processed <u>whether or not</u> another list item exists. For example, the statement

   ```
   WRITE (6, 12)

   12 FORMAT (///4HABCD)
   ```

   would produce three blank records and one record containing ABCD before reaching the final right parenthesis. When there are no more items remaining in the list and the final right parenthesis has been reached or a conversion specification has been found, the current record is terminated, and control is passed to the statement following the READ or WRITE statement that initiated the input/output operation.

5. When the final parenthesis of a FORMAT statement is encountered by the FORMAT processor, a test is made to determine if all list items have been processed. If the list has been exhausted, the current record is terminated, and control is passed to the statement following the READ or WRITE statement that initiated the input/output operation. However, if another list item is present, an additional record is begun, and the FORMAT statement is rescanned. The rescan takes place as follows:

   a. If there are no parenthesized groups of specifications within the FORMAT statement, the entire FORMAT is rescanned.

FORMAT and List Interfacing (Cont'd.)

b. If, however, one or more parenthesized groups do appear, the rescan is started with the group whose right parenthesis was the last one encountered prior to the final right parenthesis of the FORMAT statement. In the following example, the rescan begins at the point indicated:

```
                          ┌──────────────────────────┐
           ┌────┐         │      ┌────────────────┐  │
                          │      │     ┌──────┐   │  │
FORMAT (3X, (F7.2, A3),  (3HABC (3I4,  (G15.7//), A3)), E29. 12, 3HXYZ)
                          ↑                          ↑                ↑
                    rescan begins           last Internal       final right
                        here                   closing          parenthesis
                                             parenthesis        of FORMAT.
```

6. Each list item to be converted is processed by one specification or one iteration of a repeated specification, with the exception of complex data, which are processed by two such specifications.

7. Each READ or WRITE statement containing a non-empty list must refer to a FORMAT statement that contains at least one conversion (see step 4 above) specification. If this condition is not met, the FORMAT statement will be processed, but an error will occur.

FORMAT - Stored in Arrays

As mentioned previously, a FORMAT, including the beginning left parenthesis, the final right parenthesis, and the specifications enclosed therein, may be stored in an array. The FORMAT must be stored as a Hollerith string (i. e., a string of characters). The string may be "loaded" to the array by any of the methods that are normally used with numeric data. The FORMAT may also be stored in a scalar variable. This should be done only if the FORMAT is extremely short or if the ordering of other scalars (or arrays) is known (through EQUIVALENCE).

The format need not be stored starting in the first element of the array, however, it must be stored in consecutive elements. If the FORMAT is not stored in the first element of the array, then the starting element must be specified as the FORMAT specification.

Examples:

FORMAT stored in scalar X: WRITE (6,X) list

FORMAT stored in array A, starting in first element:
```
WRITE(8,A) list or
PRINT A, list
```

Format stored in array FMT, starting at fifth element:
```
READ (7,FMT(5) ) list
```

Format stored in array WWW, starting at a calculated element:
```
READ WWW(I-J+7), list
```

FORMAT - Stored in Arrays (Cont'd.)

if the variable M is an integer array, the following method may be used to store a FORMAT in M:

```
        M(1) = 3H(F8
        M(2) = 3H.5,
        M(3) = 3H3HS
        M(4) = 3HAM,
        M(5) = 3HI3)
             OR
        READ (N,90) (M(I), I = 1,5)
   90   FORMAT (5A3)
          External Input is: (F8.5, 3HSAM, I3)
```

Auxiliary Input/Output Statements

The following set of statements enable the user to manipulate magnetic tapes and sequential disc files.

REWIND Statement

This statement is expressed as:

REWIND i

where i is an integer variable or constant.

Execution of a REWIND statement causes the units whose logical file number is the integer value i to be rewound, or repositioned to the beginning of the file.

BACKSPACE Statement

The BACKSPACE statement has the form:

BACKSPACE i

where i is an integer variable or constant.

BACKSPACE Statement (Cont'd.)

When a BACKSPACE statement is executed, the file referenced by the integer value of i is backspaced one logical record. In the case of binary tapes, a logical record may consist of more than one physical record. A logical record is interpreted as all the information output by one binary WRITE statement. Because of this, the BACKSPACE statement may not be used to backspace over binary records that have not been produced by a DC 6024 FORTRAN binary WRITE statement.

REWIND and BACKSPACE statements that are executed for files already positioned at the beginning of the file have no effect.

ENDFILE Statement

This statement causes an end-of-file to be written on the specified file and has the form of:

```
ENDFILE i
```

where i is an integer variable or constant whose value determines the unit on which the end-of-file record is to be written.

Sometimes it is desirable to take a program that has been written for output on magnetic tape and assign that logical unit number to some other device, such as a line printer. Since such programs often write end-of-file and rewind their tapes at the end of the job, it is permissible to specify an END FILE or REWIND operation on any device. Refer to the appropriate operating system specification for the effect of these commands on various devices.

OPEN Statement
CLOSE Statement

These statements are expressed as:

```
OPEN i   PASSWD      or OPEN i, PASSWD
CLOSE i
```

where i is an integer variable or constant whose value determines which file is opened or closed. PASSWD is an optional 6 character name required to open files created with a password.

CALL EOFTST

A subroutine, EOFTST, has been added to the library to allow the FORTRAN user the ability to test for end-of-file. It is used as follows:

```
CALL EOFTST (file number, $stno)
```

CALL EOFTST (Cont'd.)

Where the first argument is an integer variable, constant, or expression whose value is the logical file number to be tested, and the second argument is a statement number (preceded by $) indicating the statement to which control will be transferred if an end-of-file has been encountered during the last READ. If no end-of-file was encountered, control is returned to the next sequential statement following the CALL.

CALL SSWTCH

A subroutine, SSWTCH, reads the status of the hardware sense switches. It is used as follows:

CALL SSWTCH (number, status)

where:

number - is an integer variable or constant specifying which sense switch is to be tested.

status - is an integer variable into which will be returned the value 1 if the sense switch is on, or 2 if the sense switch is off.

CALL BTIME

A subroutine, BTIME, reads the computer clock and saves the time. The routine is used as follows:

CALL BTIME

The routine has no arguments.

CALL ETIME

A subroutine, ETIME, reads the computer clock, computes the elapsed time since the last call to BTIME and outputs the time differential to the logical file assigned to the list out file in the following format:

```
RUN TIME = XXHRS XXMIN   XX.XXX000SEC
```

List Output Carriage Control

When formatted records are prepared for list output, the first character of the record is not printed. Carriage control for List Output is performed according to the first character of the record (see Table 2-5).

Table 2-5. List Output Device Carriage Control Characters

First Character	Action before Printing
"@" or "+"	0 line advance
"A" or "b̸"	1 line advance
"B" or "0"	2 line advance
"C"	3 line advance
. . .	
"O"	15 line advance
"P" or "1"	Channel 1 advance (Top of Form)
"Q"	Channel 2 advance
"R"	Channel 3 advance
. . .	
"W"	Channel 8 advance

The "+", "b̸", "0" and "1" are generally honored. Only some Line Printers honor the complete list.

Input/Output Logical File Assignments

The value of the parameter L in a READ or WRITE statement refers to a logical file to which various physical devices can be assigned. See "ASSIGN" statement in the appropriate operating system specification. The value of L is such that $0 \leq L < 64$. Table 2-6 lists the assignments usually associated with the value L.

Table 2-6.. Logical File Assignments

File	File Name	Assignment
1	Operator Communications	Console typewriter
2		Console Tape Reader*
3		Console Tape Punch*
4	Binary Input	High Speed Paper Tape Reader
5	Binary Output	High Speed Paper Tape Punch
6	List Output	Line Printer
7	Symbolic Input	Card Reader
8	Symbolic Output	Disc File

* ROS, TOS and DOS Only. Will not operate with unformatted read/write.

Random Access I/O (Extended Compiler Only)

This section applies only to those systems operating under disc oriented operating systems.

FORTRAN random access I/O uses a fixed sector size of 112 words. The record length (L) in the define file statement determines the users record length and may be any integer value. Random I/O packs or divides the users records into groups of 112 words, overflowing in the middle of the user's record to the next sector where necessary. This eliminates any wasted space on the disc. If L is 112, exactly one record per sector is read or written. If L is 28 four records are put in each sector. If L is 100, to get the second record will require one read for the first 12 words from the first sector and another read for the remaining 88 words from the second sector. If speed is essential to the user, it might help to make L some multiple or even divisor of 112, such as 2, 4, 7, 8, 14, 16 28, 56, 112, 224, 336, etc.

DEFINE FILE Statement

General Form:

DEFINE FILE a_1 (m_1, L_1, U, v_1), a_2 (m_2, L_2, U, v_2).....

or:

DEFINE FILE a_1 (m_1, L_1, v_1), a_2 (m_2, L_2, v_2).....

where:

a is an integer variable or constant corresponding to the logical file to be used. (Note that DMS and DOS logical file numbers are assumed to be octal values, hence a value of a = 14 is equivalent to DOS logical file number 16_8).

m is an integer variable or constant that defines the maximum record number in this file. The record number may vary from 0 to and including m.

L is an integer variable or constant that defines in words the length of each record in this file.

U is a fixed letter. This parameter may be omitted completely. DC 6024 FORTRAN accepts it merely for compatibility with other systems.

v is an integer variable name. This variable is set to the next available record number at the conclusion of each READ or WRITE statement.

NOTE: The DEFINE FILE statement must be executed prior to any READ, WRITE, or FIND statement referencing that file.

Disc I/O Statements

General Form:

READ (a' b) list

WRITE (a' b) list

FIND (a' b)

where:

a is an integer variable or constant corresponding to the logical file number,

b is an integer variable or constant corresponding to the record number where transmittal will start and its value may vary from 0 to and including m and list is any FORTRAN I/O list.

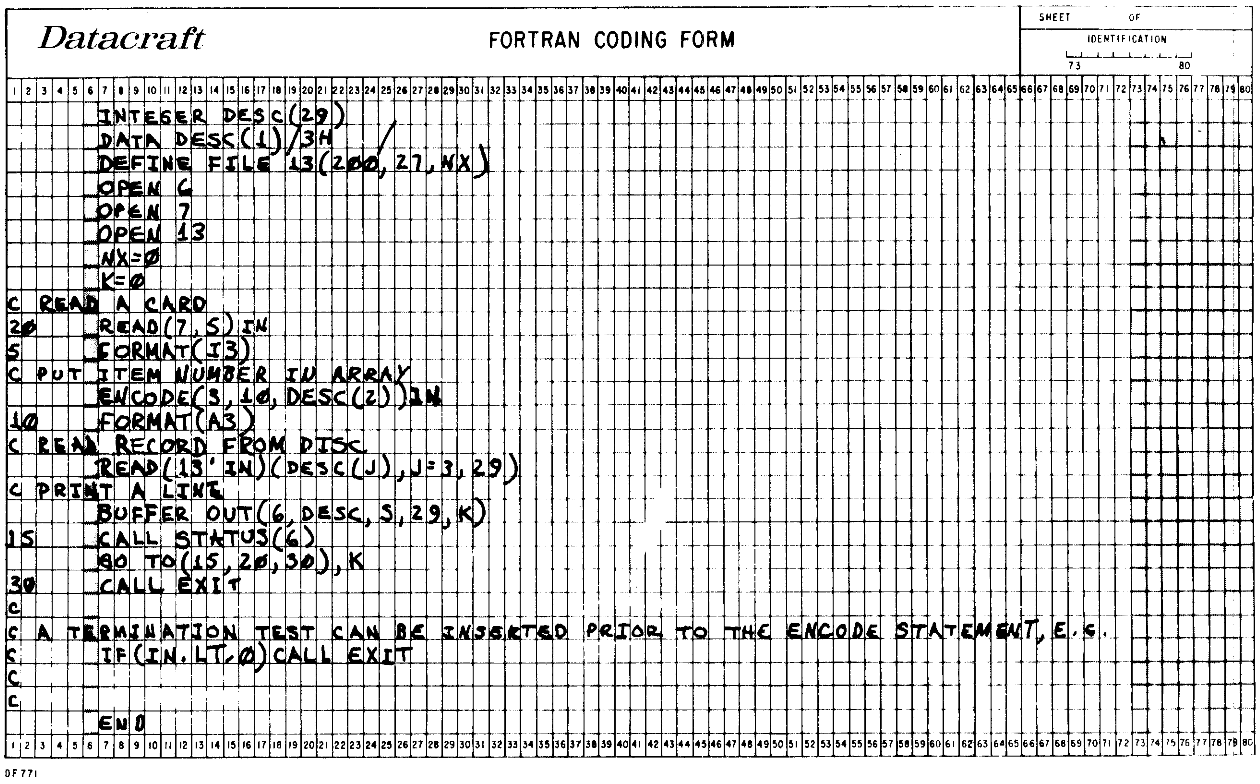
list is a standard I/O list.

2-7 INPUT/OUTPUT (CONT'D.)

Disc I/O Statements (Cont'd.)

Example:

Assume a disc file exists containing descriptions of items. The record number in the file corresponds to the item number. The following program reads a card containing the item number, reads the record from disc and prints the item number and description on the line printer. The file data will be input as unformatted data, and output via a BUFFER OUT Statement (the data is assumed to be packed 3 characters/word).


Datacraft FORTRAN CODING FORM

SHEET OF
IDENTIFICATION

```
      INTEGER DESC(29)
      DATA DESC(1)/3H   /
      DEFINE FILE 13(2ØØ,27,NX)
      OPEN 6
      OPEN 7
      OPEN 13
      NX=Ø
      K=Ø
C READ A CARD
2Ø    READ(7,5)IN
5     FORMAT(I3)
C PUT ITEM NUMBER IN ARRAY
      ENCODE(3,1Ø,DESC(2))IN
1Ø    FORMAT(A3)
C READ RECORD FROM DISC
      READ(13'IN)(DESC(J),J=3,29)
C PRINT A LINE
      BUFFER OUT(6,DESC,5,29,K)
15    CALL STATUS(6)
      GO TO(15,2Ø,3Ø),K
3Ø    CALL EXIT
C
C A TERMINATION TEST CAN BE INSERTED PRIOR TO THE ENCODE STATEMENT, E.G.
C     IF(IN.LT.Ø)CALL EXIT
C
C
      END
```

DF 771

Random Access I/O - Disc I/O Statements

Note that file number and record number are separated by an apostrophe. No data conversion is done on information transmitted via these statements.

ENCODE/DECODE Statements

The Encode/Decode statements provide memory-to-memory data conversion capability. The statements operate on an input/output list, a format statement and a program-defined internal buffer area.

The Encode statement is analagous to a WRITE statement and the Decode statement is analagous to a READ statement.

The statements take the form:

```
ENCODE
          (n,f,a) L
DECODE
```

where:

n	=	number of characters in the buffer.
f	=	the FORMAT statement number or the name of the array or the array element or the scalar variable which contains the format statement.
a	=	the "buffer" which contains the record to be ENCODE'd or DECODE'd. The buffer must be an array name, an array element, or a scalar variable. The record begins with the left most character position of "a" and continues for "n" characters.
L	=	List

There are 3 ANSCII characters per word.

Examples:

```
ENCODE(3, 7, BUF) list
DECODE (28, IFMT, JBUF(27) list
DECODE (6, 13, XYZ) list
ENCODE (NUM, ALPHA( I  ), BETA( (I+7)/3) list
```

where:

BUF, JBUF, and BETA are data buffer arrays, XYZ is a real variable

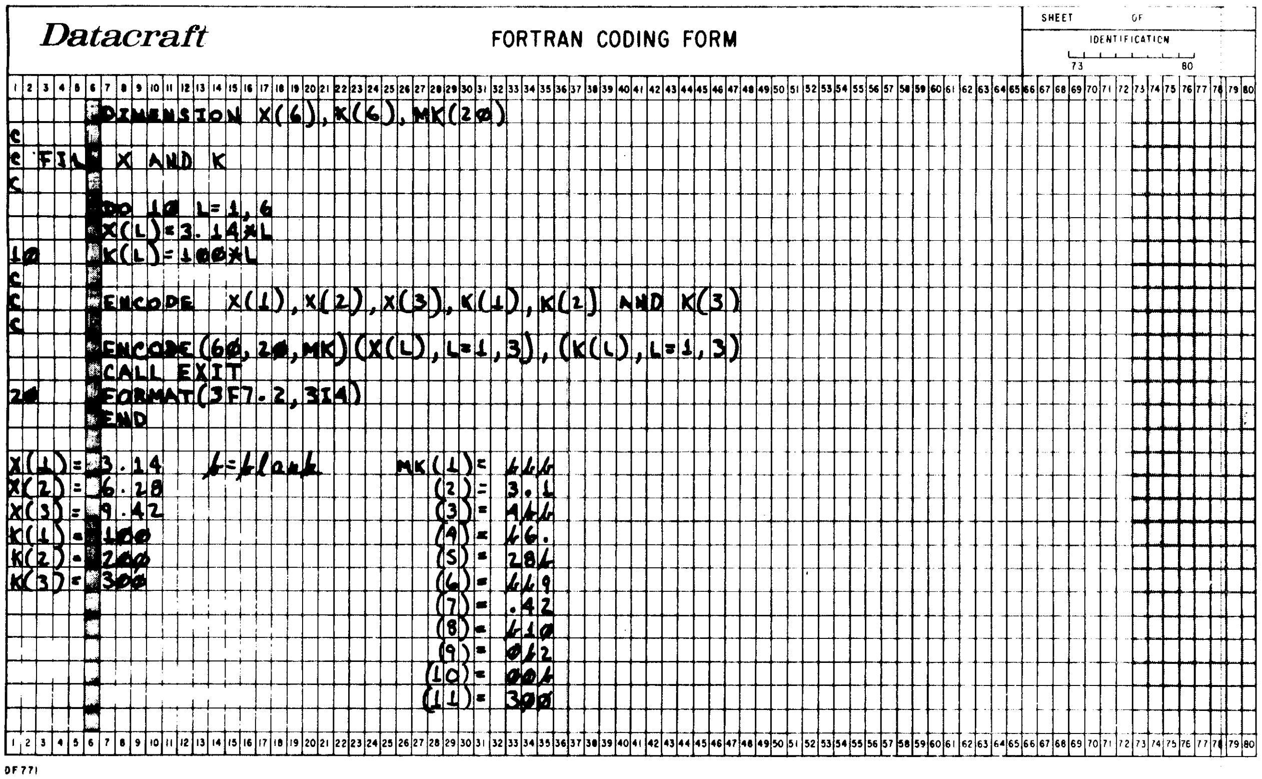
IFMT is an array containing a format

ALPHA is an array containing a format at the Ith element.

ENCODE/DECODE Statements (Cont' d.)

The ENCODE statement will transfer the list elements into the internal buffer while performing the specified format conversion.

The following example illustrates the action of the ENCODE statement. The data from arrays X and K are encoded into array MK. The example shows the arrangement of the character string in MK.



Datacraft FORTRAN CODING FORM

SHEET OF

IDENTIFICATION

73 80

```
      DIMENSION X(6),K(6),MK(2Ø)
C
C FILL X AND K
C
      DO 1Ø L=1,6
      X(L)=3.14*L
1Ø    K(L)=1ØØ*L
C
C     ENCODE  X(1),X(2),X(3),K(1),K(2) AND K(3)
C
      ENCODE(6Ø,2Ø,MK)(X(L),L=1,3),(K(L),L=1,3)
      CALL EXIT
2Ø    FORMAT(3F7.2,3I4)
      END

X(1)= 3.14   b̸=blank      MK(1)= b̸b̸b̸
X(2)= 6.28                  (2)= 3.1
X(3)= 9.42                  (3)= 4b̸b̸
K(1)= 1ØØ                   (4)= b̸6.
K(2)= 2ØØ                   (5)= 28b̸
K(3)= 3ØØ                   (6)= b̸b̸9
                            (7)= .42
                            (8)= b̸1Ø
                            (9)= Øb̸2
                           (1Ø)= ØØb̸
                           (11)= 3ØØ
```

DF771

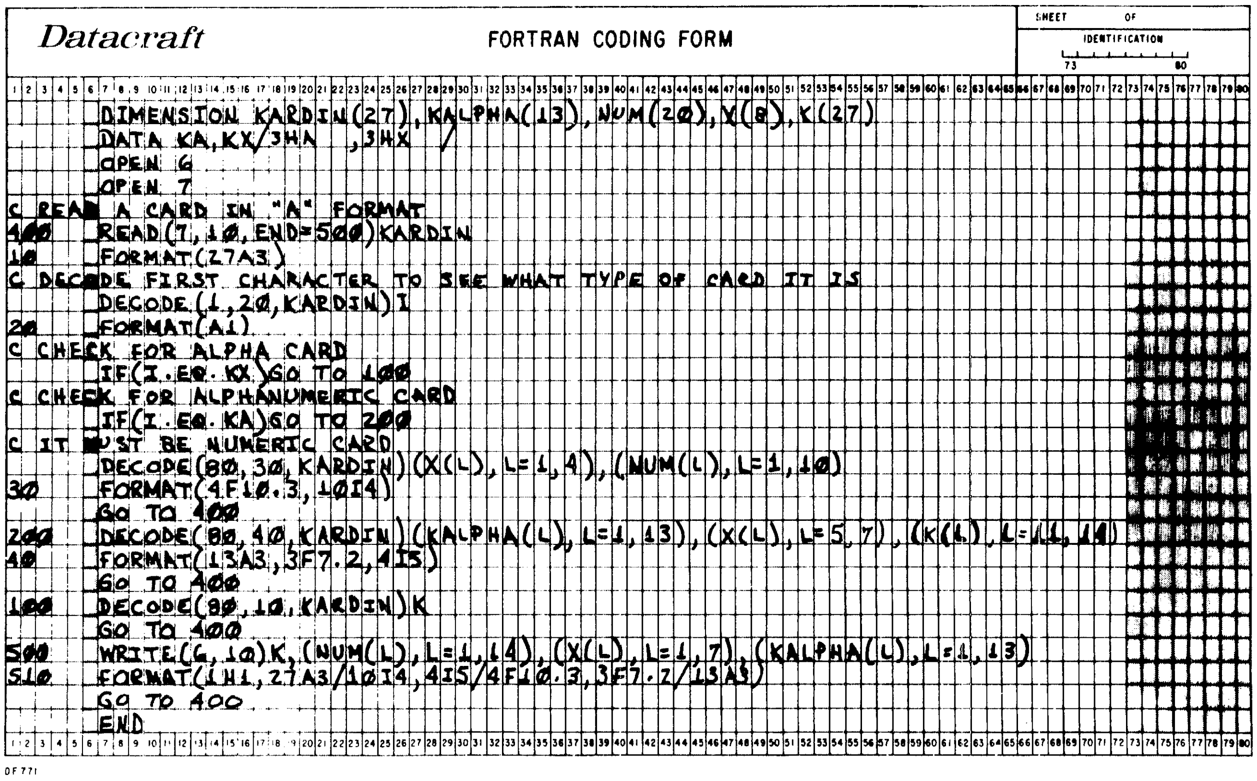
The DECODE statement will transfer the list elements from the internal buffer area to the variables specified in the list while performing the conversion specified in the format statement.

The DECODE feature permits the program to read the card (or other input record) more than once. For example, if a card deck consists of randomly mixed alphabetic, alphanumeric and numeric data cards, it is possible to determine, via a DECODE statement, what format conversion is required for the card just read. The following example has provision for one of each type card, input in any order.

ENCODE/DECODE Statements (Cont'd.)

Example: The card formats are:

ALPHA	First character will always be X. Format will be 27A3
ALPHA-NUMERIC	First character will always be A . Format will be 13A3, 3F7.2,4I5
NUMERIC	First character will be blank, sign or digit. Format will be 4F10.3,10I4



Datacraft FORTRAN CODING FORM

SHEET OF

IDENTIFICATION 73 80

```
      DIMENSION KARDIN(27),KALPHA(13),NUM(20),X(8),K(27)
      DATA KA,KX/3HA  ,3HX  /
      OPEN 6
      OPEN 7
C READ A CARD IN "A" FORMAT
400   READ(7,10,END=500)KARDIN
10    FORMAT(27A3)
C DECODE FIRST CHARACTER TO SEE WHAT TYPE OF CARD IT IS
      DECODE(1,20,KARDIN)I
20    FORMAT(A1)
C CHECK FOR ALPHA CARD
      IF(I.EQ.KX)GO TO 100
C CHECK FOR ALPHANUMERIC CARD
      IF(I.EQ.KA)GO TO 200
C IT MUST BE NUMERIC CARD
      DECODE(80,30,KARDIN)(X(L),L=1,4),(NUM(L),L=1,10)
30    FORMAT(4F10.3,10I4)
      GO TO 400
200   DECODE(80,40,KARDIN)(KALPHA(L),L=1,13),(X(L),L=5,7),(K(L),L=11,14)
40    FORMAT(13A3,3F7.2,4I5)
      GO TO 400
100   DECODE(80,10,KARDIN)K
      GO TO 400
500   WRITE(6,10)K,(NUM(L),L=1,14),(X(L),L=1,7),(KALPHA(L),L=1,13)
510   FORMAT(1H1,27A3/10I4,4I5/4F10.3,3F7.2/13A3)
      GO TO 400
      END
```

DF 771

ASYNCHRONOUS INPUT/OUTPUT (Extended Compiler Only)

The asynchronous input/output feature provides a method of performing double buffered input/output operations in either a symbolic or binary mode.

There are three statements which implement this function. They are:

```
BUFFER IN
                (u,a,m,w,s,n)
BUFFER OUT

CALL STATUS     (u)
```

where:

u is the logical unit number

a is an array name, an array element, or a scalar variable specifying the starting address for the operation.

m is the mode of operation

S = Symbolic

B = Binary

w specifies the number of words to be input or output (in the symbolic mode

3*w characters will be transferred)

s is an integer variable which will be set by a call STATUS request.

NOTE

Under DMS and VULCAN operating systems, the value of S = 1 will never occur. A call to the STATUS routine will cause the system to wait for I/O completion before returning to the user program.

The variable s is set as follows:

S = 1 operation incomplete

S = 2 successful completion

S = 3 EOF encountered

n is an optionally present integer variable which will be set with the actual number of words transferred when the operation is completed.

Examples:

```
BUFFER IN (8, BUF,S, NUM, ISTAT, IWORDS)
BUFFER OUT (LFN, IBUF,B,112,JSTAT)
BUFFER OUT ('17,JBUF(I-J+3), B,N,II)
BUFFER IN (3,KBUF(5),S,KK,LL)
```

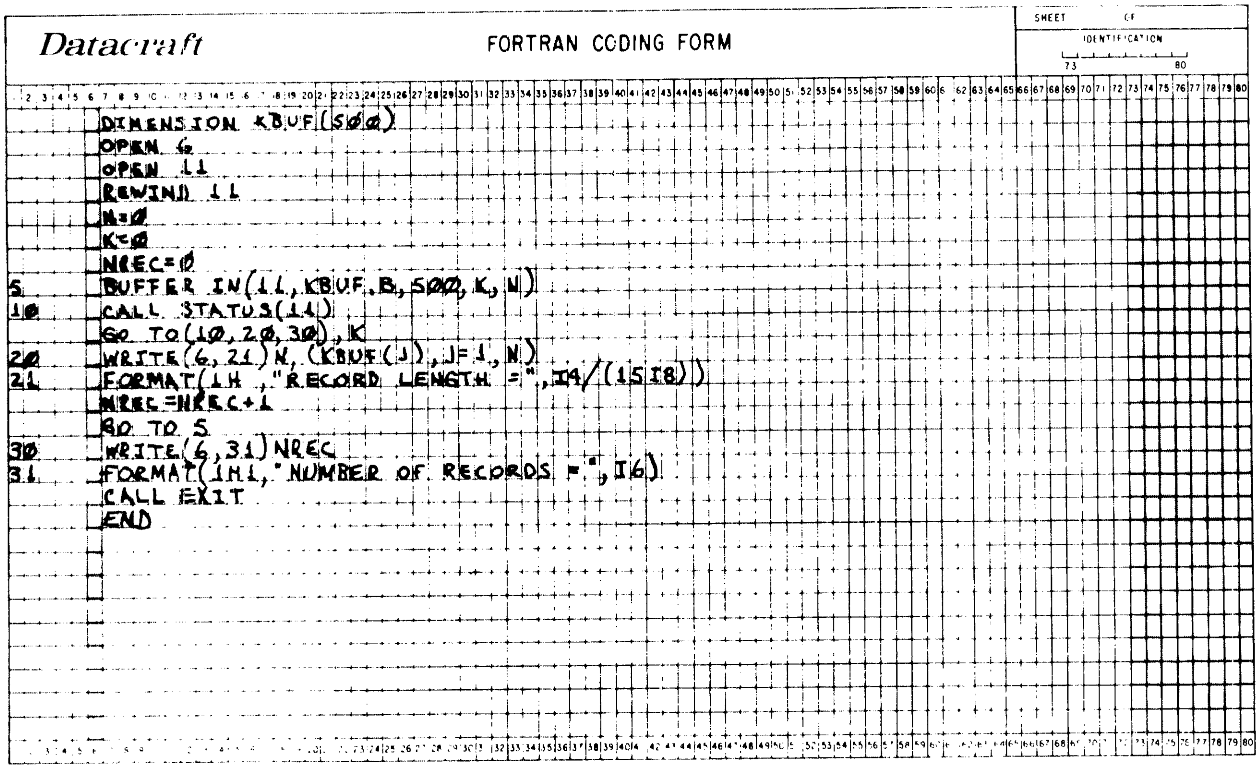
ASYNCHRONOUS INPUT/OUTPUT (Extended Compiler Only) (Cont'd.)

The BUFFER IN/ BUFFER OUT statements permit processing, both on input and output, records of arbitrary length and format, without regard to the usual restrictions. It permits complete program control of the data and enables such functions as: interpretating binary tapes produced on other machines or by other programs, reading and writing binary cards, and in conjunction with the ENCODE/DECODE functions, processing long formatted records.

Example:

Read a binary tape (9 track, 3 C/W, 800 bpi) of variable length records none of which exceeds 500 words, terminated by End-of-File.

The program will output the records 15 words per line in decimal. Each record printed will be preceded by the record length. After all records have been printed, the number of records in the file will be printed.



Datacraft FORTRAN CODING FORM

```
      DIMENSION KBUF(5ØØ)
      OPEN 6
      OPEN 11
      REWIND 11
      N=Ø
      K=Ø
      NREC=Ø
5     BUFFER IN(11,KBUF.B,5ØØ,K,N)
1Ø    CALL STATUS(11)
      GO TO(1Ø,2Ø,3Ø),K
2Ø    WRITE(6,21)N,(KBUF(J),J=1,N)
21    FORMAT(1H ,"RECORD LENGTH =",I4/(15I8))
      NREC=NREC+1
      GO TO 5
3Ø    WRITE(6,31)NREC
31    FORMAT(1H1,"NUMBER OF RECORDS =",I6)
      CALL EXIT
      END
```

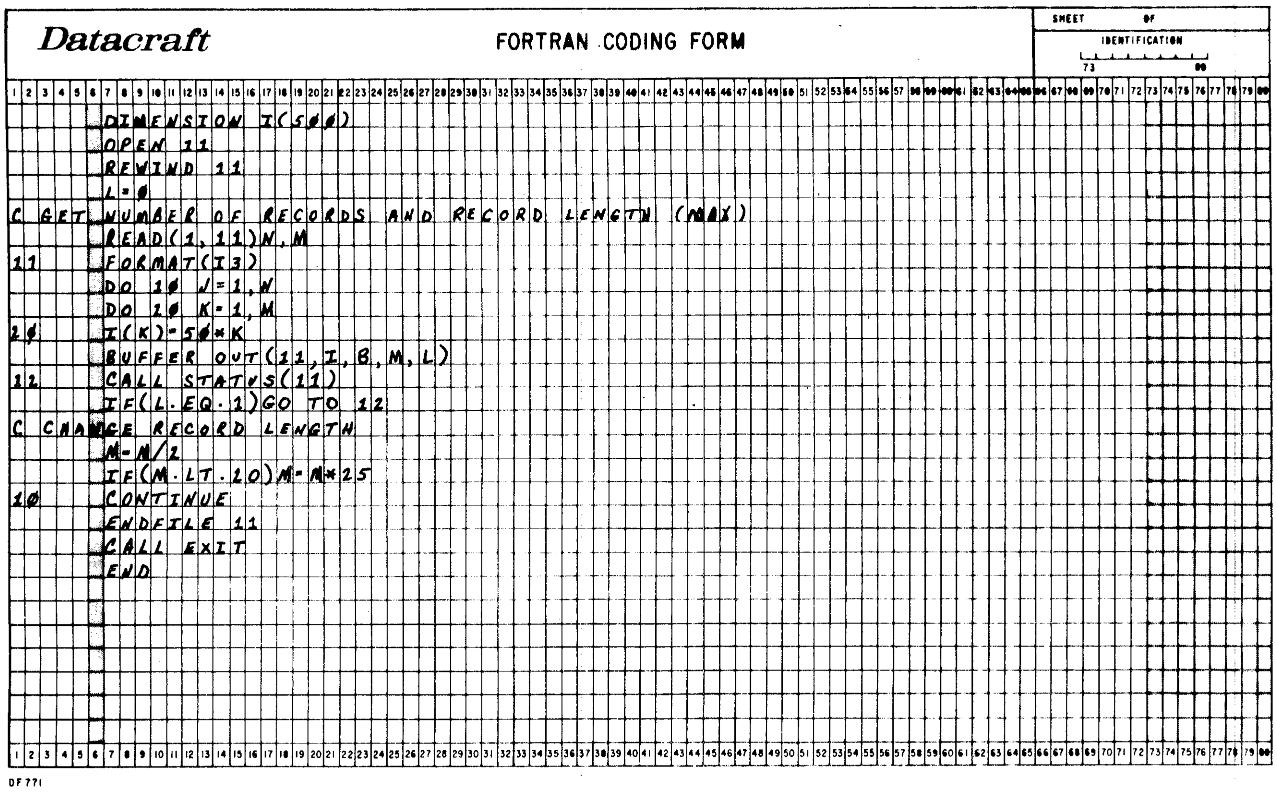
DF 77

Example:

A test program to generate the binary tape used in the preceeding example.

2-7 INPUT/OUTPUT (CONT'D.)

ASYNCHRONOUS INPUT/OUTPUT (Extended Compiler Only) (Cont'd.)



Datacraft FORTRAN CODING FORM

```
      DIMENSION I(5ØØ)
      OPEN 11
      REWIND 11
      L=Ø
C GET NUMBER OF RECORDS AND RECORD LENGTH (MAX)
      READ(1,11)N,M
11    FORMAT(I3)
      DO 1Ø J=1,N
      DO 2Ø K=1,M
2Ø    I(K)=5Ø*K
      BUFFER OUT(11,I,B,M,L)
12    CALL STATUS(11)
      IF(L.EQ.1)GO TO 12
C CHANGE RECORD LENGTH
      M=M/2
      IF(M.LT.10)M=M*25
1Ø    CONTINUE
      ENDFILE 11
      CALL EXIT
      END
```

DF771

2-8 DECLARATION STATEMENTS

Declaration statements are used to define the data type of variables and functions, the dimensions of arrays, storage allocation, initial values of variables, and to provide similar information.

Classification of Identifiers

An identifier may be classified as referring to any of the following:

scalar

array

subprogram

dummy

COMMON block

Classification of Identifiers (Cont'd.)

The category into which an identifier is placed and the type (if any) associated with it depend on the contexts in which the identifier appears in the program. These appearances constitute explicit or implicit declarations of the way the identifier is to be classified.

Implicit Declarations

Unless specifically declared to be in a particular category or type, identifiers that appear in executable or DATA statements are implicitly classified according to the following set of rules.

1. When applicable an identifier is integer if it begins with I, J, K, L, M, or N. If it begins with any other letter, it is real.
2. An identifier that is called with a CALL statement is a subprogram.
3. An identifier that appears in an expression, followed by an argument list enclosed in parentheses, is a subprogram, unless it has been explicitly declared as an array.
4. An identifier that appears to the left of an equal sign, followed by a dummy list enclosed in parentheses, is a statement function definition if it also compiles with the rules given in Section 2-9 "PROGRAMS AND SUBPROGRAMS - Statement Functions". Otherwise it is an error. This does not apply to declared arrays.
5. Any other appearance of an identifier in a executable or DATA statement (i.e., other than followed by a left parenthesis or in a CALL statement) causes it to be classified as a scalar variable.
6. An identifier that appears in no executable or DATA statement, but does appear in a COMMON or EQUIVALENCE statement, is classified as a scalar variable unless explicitly declared as an array.
7. Intrinsic and basic external library functions have an inherent type associated with them, as shown in Table 2-8, Library Functions. Inherent type is not equivalent to implicit type. Section 2-9 contains a complete description of library functions.

Explicit Declarations

All other declarations are explicit declarations. Explicit declarations are required in order to classify an identifier in any way other than those described above. Explicit declarations include:

array declarations

type declarations

Explicit Declarations (Cont'd.)

storage allocation declarations

subprogram declarations

subprogram definitions

Explicit declarations override implicit declarations. They must precede the first executable or DATA statement of the program.

Conflicting and Redundant Declarations

Except where specifically noted to the contrary, definitions and declarations of the classification of an identifier may not conflict. For example, an identifier may not be both a subprogram name and an array name, both integer and real type, defined as a subprogram in more than one place, etc.

Array Declarations

Array declarations explicitly define an identifier as the name of an array variable and have the form:

$v_1 (d_1), v_2 (d_2), v_3 (d_3), \ldots v_m (d_m)$

where:

each v_i is the identifier of the array, and
each d_i is a set of 1, 2, or 3 integer constants separated by commas.

Examples:

X(10)

ARRAY (5,15,10)

PLANE (25,25)

CUBE (10,10,10)

LINE (14000)

When v is a dummy array in a subprogram, the d_i may be integer variables instead of constants (see Section 2-9, PROGRAMS AND SUBPROGRAMS - Adjustable Dimensions).

Array declarations may appear in

DIMENSION statements

Explicit type statements

COMMON statements

Array Storage

Although an array may have up to three dimensions, it is placed in storage as a linear string. This string contains the array elements in sequence (from low address storage toward high address storage) such that the leftmost dimension varies with the highest frequency, the next leftmost dimensions varies with the next highest frequency, etc., i.e., 2-dimensional arrays are stored "column-wise". Figure 2-2 contains a pictorial example of array storage.

array A (3, 3, 2)

Item	Element
1	A(1,1,1)
2	A(2,1,1)
3	A(3,1,1)
4	A(1,2,1)
5	A(2,2,1)
6	A(3 2,1)
7	A(1,3,1)
8	A(2,3,1)
9	A(3,3,1)
10	A(1,1,2)
11	A(2,1,2)
12	A(3,1,2)
13	A(1,2,2)
14	A(2,2,2)
15	A(3,2,2)
16	A(1,3,2)
17	A(2,3,2)
18	A(3,3,2)

Figure 2-2. An Example of Array Storage for an Array Defined as A(3,3,2)

Reference to Array Elements

Reference to array elements must contain the number of subscripts that correspond to the number of dimensions declared for the array (except as discussed for EQUIVALENCE statements). References that contain an incorrect number of subscripts are treated as errors.

2-8 DECLARATION STATEMENTS (CONT'D.)

Reference to Array Elements (Cont'd.)

Furthermore, the value of each subscript should be within the range of the corresponding dimension, as specified in the array declaration. Otherwise, the references may not be to data belonging to the set of elements that comprise the array.

DIMENSION Statement

This statement is used only to define the dimensions of arrays and has the form:

DIMENSION $v_1, v_2, v_3, \ldots, v_n$

where: v_i are array declarations.

A DIMENSION statement does not affect the type or allocation of the arrays declared.

Example:

DIMENSION MGO (16), LTO (14), BB (36, 22, 34)

IMPLICIT Statement

These statements are used to define, implicitly, the type of an identifier by the first letter of the identifier. The IMPLICIT statement has the following form:

IMPLICIT $E_1, E_2, \ldots E_n$

where each E_i is a specification of the form:

type $S_1, S_2, \ldots S_n$

where type is one of five declarations: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL. And each S_i is a specification of the form:

$(I_1, I_2, \ldots I_n)$

where each I_i is either a single letter or is a range of letters $I_a - I_b$ where I_b alphabetically follows I_a.

Only the identifiers beginning with the first letter in an IMPLICIT statement are affected by the retyping. The typing of intrinsic, basic external functions or explicitly type identifiers are not affected by the IMPLICIT statement. The letters I, J, K, L, M and N are typed as INTEGER unless implicitly typed otherwise by an IMPLICIT statement.

An IMPLICIT statement may only be preceded with a program unit by a NAME, SUBROUTINE, FUNCTION, another IMPLICIT statement or a comment card.

Examples:

IMPLICIT INTEGER (A-D,E,Y,X),(Z)

IMPLICIT REAL (I),(M-Q),(B)

Identifiers beginning with the letters A,C-H,J-L,X-Z are typed as INTEGER. Identifiers beginning with the letters B,F-I,M-W are typed as REAL.

Storage Allocation Statements

These statements are used to arrange variable storage in special ways, as required by the programmer. If no storage allocation information is provided, the compiler allocates all variables within the program in the order in which they appear. The storage allocation statements are:

COMMON statement

EQUIVALENCE statement

Allocation of Variable Types

To make proper use of the storage allocation statements, it is necessary to know the amount of storage required by each type of variable. Table 2-7 indicates the size associated with each type.

Table 2-7. Storage Allocation Requirements

Type	Words
Integer	1
real	2
double-precision	2
complex	4
logical	1

COMMON Statement

The COMMON statement is used to assign variables to a region of storage called COMMON storage. COMMON storage provides a means by which more than one program or subprogram may reference the same data.

The COMMON statement has the form:

COMMON $w_1, w_2, w_3 \dots w_n$

where:

w_i has the form

$/c/\ v_1, v_2, v_3, \dots, v_m$

COMMON Statement (Cont'd.)

where:

c is either the identifier of a labeled COMMON block, or is absent, indicating blank COMMON, and

v_i is a scalar or array name or array declaration.

When w_1 (the first specification in the statement) is to specify blank COMMON, the slashes may be omitted. In all other places, blank COMMON is indicated by two consecutive slashes. For example:

```
COMMON MARKET, SENSE / GROUP3/X, Y, JUMP // GETIT, COLD
```

For each specification (w_i), the variables listed are assigned to the indicated COMMON block or to blank COMMON. The variables are assigned in the order they appear. Thus in the above example MARKET, SENSE, GETIT, and COLD are assigned to blank COMMON, while X, Y, and JUMP are placed in labeled COMMON block GROUP3.

Labeled COMMON

Labeled COMMON blocks are discrete sections of the COMMON region and as such are independent of each other and blank COMMON.

Any labeled COMMON block may be referenced by any number of programs or subprograms which comprise an executable program (see Section 2-9). References are made by block name, which must be identical in all references. All labeled COMMON blocks need not be defined in any one program; in fact, only those blocks containing data needed by the program require definition.

The variables defined as being in a particular labeled COMMON block do not necessarily have to correspond in type or number between the programs in which the block is referenced. However, the definition of the overall size of a labeled COMMON block must be identical in all the programs in which it is defined. For example:

```
SUBROUTINE A                  SUBROUTINE B

REAL T, V, X (10)             COMPLEX G, F(5)

COMMON /SET1/T, V, X          COMMON /SET1/G, F

.                             .
.                             .
.                             .
```

Both references to the COMMON block SET1 correspond in size. That is, both subprograms define the block SET1 as containing 24 words; the definition in subroutine A specifies 12 items of real type, and the definition in subroutine B declares 6 items of complex type.

Labeled COMMON (Cont'd.)

Reference may be made to the name of a labeled COMMON block more than once in any program. A multiple reference may occur in a single COMMON statement, or the block name may be specified in any number of individual COMMON statements. In both cases the processor links together all variables, defined as being in the block, into a single labeled COMMON block of the appropriate name.

Block names must be unique with respect to

1. Subprogram names defined, explicitly or implicitly, to be external references (see Section 2-8, DECLARATION STATEMENTS - External Statement).

2. Other block names

A labeled COMMON block may have the same name as an identifier in any classification other than the above; however, it is usually preferable to choose block names that are totally unique.

Blank COMMON

There is only one contiguous area of memory assigned to blank COMMON, and empty block name specifications always refer to it. Furthermore, as opposed to labeled COMMON, blank COMMON areas, defined in the various programs and subprograms that comprise an executable program (see Section 2-9), do not have to correspond in size. The only restriction is that the first blank COMMON definition in a set of programs and subprograms to be linked must be the largest block (DOS, TOS, ROS systems only).

References may be made to blank COMMON any number of times within a program. The multiple references may occur in a single COMMON statement or in several COMMON statements. In either case, all variables defined as being in blank COMMON will be placed together in the blank COMMON area. All COMMON statements must occur prior to the first executable or DATA statement of a program.

Variables in blank or labeled COMMON may not be initialized by the DATA statement except in a BLOCK DATA subprogram (see Section 2-9, PROGRAMS AND SUBPROGRAMS - BLOCK DATA Subprograms).

Arrangement of COMMON

Each labeled COMMON block and the blank COMMON area contain, in the order of their appearance, the variables declared to be in the labeled block or the unlabeled area. The variables in each section of the COMMON region are arranged from low address storage towards high address storage. That is, the first variable to be declared as being in a particular section is contained in the low address word or words of that section, while the last variable to be declared as being in the section is contained in the high address word or words of the section. Array variables are stored in their normal sequence (see Section 2-3, DATA - Arrays in Storage) within the COMMON block. For example, the statements:

Arrangement of COMMON (Cont'd.)

```
COMMON /E/W, X(3,3) //T, B, Q /E/J

COMMON K,M/E/Y//C(4),H,N(2),Z
```

cause the following arrangement of COMMON:

Item	Block E	Blank COMMON
1	W	T
2	X (1,1)	B
3	X (2,1)	Q
4	X (3,1)	K
5	X (1,2)	M
6	X (2,2)	C (1)
7	X (3,2)	C (2)
8	X (1,3)	C (3)
9	X (2,3)	C (4)
10	X (3,3)	H
11	J	N (1)
12	Y	N (2)
13		Z

Note that, since a segment of the COMMON region may be defined differently in each program, it may be quite important to be aware of which items in a segment contain certain variables.

For example:

```
SUBROUTINE DOG          SUBROUTINE CAT          SUBROUTINE PIG
COMMON /S/A,C,B (100)   COMMON /S/A,X (51)      COMMON /S/ALPHA (52)
                        COMMON /S/Y (50)        COMMON /S/Y (50)
   .                       .                       .
   .                       .                       .
   .                       .                       .
```

Arrangement of COMMON (Cont'd.)

will define the block S as follows:

Item	DOG	CAT	PIG
1	A	A	ALPHA (1)
2	C	X(1)	ALPHA (2)
3	B(1)	X(2)	ALPHA (3)
.	.	.	.
.	.	.	.
.	.	.	.
52	B(50)	X(51)	ALPHA (52)
53	B(51	X(1)	Y(1)
54	B(52)	Y(2)	Y(2)
.	.	.	.
.	.	.	.
.	.	.	.
102	B(100)	Y(50)	Y(50)

which allows the routines DOG and CAT to access the variable A by that identifier, the routines CAT and PIG to access the array variable Y by that identifier, and yet the integrity of the block S is maintained. (These examples assume A, B, C, X, Y, and ALPHA are of the same type.)

MONITOR COMMON Statement (VULCAN compiler only)

The MONITOR COMMON statement is used to declare a labelled common block to be a MONITOR COMMON block. A MONITOR COMMON block is a specialized type of file which can be randomly and simultaneously referenced by several users in a multi-programming environment. (See VULCAN system reference for a more detailed description.)

The form of the statement is:

MONITOR COMMON $c_1, c_2, c_3, \ldots c_n$

where each c_i is the name of a common block which is to be typed as a MONITOR COMMON block. (The name is not enclosed within slashes.)

There may be any number of MONITOR COMMON statements within a program. Any common block referenced within a MONITOR COMMON statement must also be defined within a regular COMMON statement, however, it need not be defined prior to the MONITOR COMMON statement.

MONITOR COMMON Statement (VULCAN compiler only) (Cont'd.)

Examples:

MONITOR COMMON BLOCK1, BLOK2,BLOK3

MONITOR COMMON ABC

SPECIAL COMMON Statement (VULCAN compiler only)

The SPECIAL COMMON statement is used to declare a common block to be a SPECIAL COMMON block. A SPECIAL COMMON block is allocated at or above 65K (200000_8) within the users logical address space. SPECIAL COMMON blocks may extend to the full addressable space within the user's logical address space ($256K_{10} = 1000000_8$ words).

When a common block is declared to be a SPECIAL COMMON block, then the code generated to reference data within this common block is automatically modified to correctly access the data. There is no further effort necessary on the part of the user. In particular, the "L" option is not necessary to reference this data, however, it may be used if other data not in SPECIAL COMMON but above 32K is to be accessed. (See Paragraph 3-4 for a further discussion of compile time options.)

The form of the statement is:

SPECIAL COMMON $c_1, c_2, c_3, \ldots c_n$

where each c_i is the name of a Common Block (or null if blank common is intended) which is to be typed as a SPECIAL COMMON block. (The name is <u>not</u> enclosed with slashes.)

There may be any number of SPECIAL COMMON statements within a program. Any common block referenced within a SPECIAL COMMON statement must also be defined within a regular COMMON statement, however, it need not be defined prior to the SPECIAL COMMON statement.

Examples:

SPECIAL COMMON BLOKA, BLOKB, BLOKC

SPECIAL COMMON XYZ

SPECIAL COMMON (Blank common)

EQUIVALENCE Statement

The EQUIVALENCE statement controls the allocation of variables relative to one another. Generally, it is used to assign more than one variable to the same storage location or locations. It is expressed as:

EQUIVALENCE $s_1, s_2, s_3, \ldots, s_n$

where: each s_i is an equivalence set of the form

$(v_1, v_2, v_3, \ldots, v_m)$

Each equivalence set specifies that all of the v_i are to be assigned the same storage location. The v_i may be a scalar name or an array element. As an example:

EQUIVALENCE Statement (Cont'd.)

```
REAL B, C, A(3,3), X(4,3,2)

EQUIVALENCE (A(1,3), B), (C, X(2,1,2))
```

would make B and A(1,3) equivalent, and, similarly, C and X(2,1,2) equivalent.

The EQUIVALENCE statement also permits an element of a multi-dimensional array to be expressed as its equivalent single dimensioned subscript value as defined by the "Array Element Successor Function" which states that:

$M(i,j,k) = M(i + A(j-1) + A \cdot B \cdot (k-1))$

where i, j, and k are subscript expressions, and A and B are declared dimensions of the array M (A, B, C)

For example, the following statements effect the same equivalence as in the previous example:

```
REAL B, C, A(3,3), X(4,3,2)

EQUIVALENCE (A(7), B), (C, X(14))
```

EQUIVALENCE references to variables of a type which occupy other than one word per element (e.g., complex) are interpreted as referring to the first word of the first element. The effect of the statements:

```
INTEGER M(8)

REAL A(4)

COMPLEX Z(2)

EQUIVALENCE (M(1), A(1), Z(1))
```

is to cause the indicated equivalence:

Word		Variables	
1	M(1)	$A(1)_1$	$Z(1)r_1$
2	M(2)	$A(1)_2$	$Z(1)r_2$
3	M(3)	$A(2)_1$	$Z(1)i_1$
4	M(4)	$A(2)_2$	$Z(1)i_2$
5	M(5)	$A(3)_1$	$Z(2)r_1$
6	M(6)	$A(3)_2$	$Z(2)r_2$
7	M(7)	$A(4)_1$	$Z(2)i_1$
8	M(8)	$A(4)_2$	$Z(2)i_2$

EQUIVALENCE Statement (Cont'd.)

No storage allocation declaration is permitted to cause conflicts in the arrangement of storage. Each COMMON, and EQUIVALENCE statement determines the allocation of the variables referenced in them. Therefore, no EQUIVALENCE set should contain references to more than one variable which has previously been allocated, and COMMON statements should contain references to any variable that has previously been declared to be in COMMON.

In an EQUIVALENCE statement, an unsubscripted array name appearing as an element of an equivalence group has the same effect as specifying the first element of that array.

Example:

```
DIMENSION A( 10, 10), B (50)

EQUIVALENCE (A, B (20))
```

has the same effect as:

```
DIMENSION A( 10,10), B (50)

EQUIVALENCE (A ( 1,1), B (20))
```

COMMON and EQUIVALENCE Interactions

In all cases, the storage allocation sequence specified in a COMMON statement takes precedence over any EQUIVALENCE specifications. Consequently, EQUIVALENCE statements are not allowed to define conflicting allocations of COMMON storage.

Storage allocation of arrays by use of the EQUIVALENCE statement may vary, depending upon whether or not the arrays are in a COMMON block. For example, if two arrays (not in COMMON) are equivalenced:

```
DIMENSION A(3), B(5), C(4)

EQUIVALENCE (A(3), C(2))
```

storage allocation is assigned as follows:

COMMON and EQUIVALENCE Interactions (Cont'd.)

Item	Variable
1	A(1)
2	A(2) = C(1)
3	A(3) = C(2)
4	C(3)
5	C(4)
6	B(1)
7	B(2)
8	B(3)
9	B(4)
10	B(5)

However, if the arrays are in common:

```
COMMON A(3), B(5)

DIMENSION C(4)

EQUIVALENCE (A(3), C(2))
```

The array C is equivalenced as expected, but the original allocation specified by the COMMON statement is not disturbed:

Item	Variable
1	A(1)
2	A(2) = C(1)
3	A(3) = C(2)
4	B(1) = C(3)
5	B(2) = C(4)
6	B(3)
7	B(4)
8	B(5)

The above examples assumes all variables are of the same data type.

COMMON and EQUIVALENCE Interaction (Cont'd.)

It is permissible for an EQUIVALENCE to cause a segment of a COMMON block to be lenghtened beyond the upper bound established by the last item defined to be in that block. However, it is not permissible for an EQUIVALENCE declaration to cause a block to be lengthened beneath the lower bound established by the first item declared to be in that block. Both conditions are demonstrated in the examples below:

```
COMMON /BLK1/A(5), B/BLK2/E (4), H, Y(4)

DIMENSION Z(10), V(5)

EQUIVALENCE (A(1), Z(1)), (V(4), E(2))
```

The first EQUIVALENCE set is a permissible extension of the block BLK1, whereas the second set illegally defines an extension of the block BLK2. The declared storage allocation would appear as shown below:

Item	BLK1	BLK2 (illegal extension)
-		V(1)
-		V(2)
1	A(1) = Z(1)	E(1) = V(3)
2	A(2) = Z(2)	E(2) = V(4)
3	A(3) = Z(3)	E(3) = V(5)
4	A(4) = Z(4)	E(4)
5	A(5) = Z(5)	H
6	B = Z(6)	Y(1)
7	Z(7)	Y(2)
8	Z(8)	Y(3)
9	Z(9)	Y(4)
10	Z(10)	

assuming all items are of the same data type.

EXTERNAL Statement

The EXTERNAL statement has the form:

EXTERNAL $p_1, p_2, p_3, \ldots, p_n$

This statement declares that the identifiers listed are subprogram identifiers so that they may be referenced externally.

The practical use of the EXTERNAL statement is that it declares, as subprograms, names which might otherwise be classified implicitly as scalars, so that they may be passed as arguments to other subprograms (see Section 2-9, PROGRAMS AND SUBPROGRAMS - Arguments and Dummies). For example, if the subprogram name F appears in the statement

```
CALL ALPHA(F)
```

but appears in no other context to indicate that it is a subprogram, it would be implicitly classified as a scalar. The EXTERNAL statement is used to avoid this.

DATA Statement

The DATA statement has the form:

DATA $S_1, S_2, S_3, \ldots, S_n$

where:

S_i is a data set specification of the form

variable-list / constant-list /

The primary purpose of the DATA statement is to give names to constants; instead of referring to e as 2.71828 at every appearance, the variable e can be given that value with a DATA statement and used instead of the larger form of the constant. This also simplifies modifying the program, if a more accurate value is required.

Giving e a value with a DATA statement is somewhat different from giving it a value with an assignment statement. With the DATA statement, the value is assigned when the program is loaded, whereas with the assignment statement, it is done at execution time.

The effect of the DATA statement is to initialize the variables in each data set to the values of the constants in the set, in the order listed. For example, the statement

```
DIMENSION IA(2)

DATA X, J, L/3.5,7, .TRUE./,ALPHA/9/,IA/6HABCDEF/
```

is equivalent to the assignment statements

DATA Statements (Cont'd.)

```
X = 3.5
J = 7
L = .TRUE.
ALPHA = 9
IA(1) = 3HABC
IA(2) = 'DEF'
```

except that the DATA statement is not executable; its assignments take place upon loading.

DATA Variable List

A data variable list is a list containing names of scalar and/or array elements. Array elements must have integer constant subscripts. Dummy arguments may not appear in a DATA list. If a list contains more than one entry, the entries must be separated by commas.

When an unsubscripted array name appears as an element of a DATA variable list, it has the same effect as if all of the elements of the array are specified in array element successor function order.

Example:

```
DIMENSION A(2,3)
DATA A/6*0.0/
```

has the same effect as:

```
DIMENSION A(2,3)
DATA A(1,1), A(2,1),A(1,2),A(2,2),A(1,3),A(2,3)/6*0.0/
```

An implied DO list may also be used as a list element. An implied DO list is a simple list followed by a comma and then a DO-control of the form:

$v = i_1, i_2, i_3$

where v is a DO-control integer variable name. This name is used as a local variable within the DATA statement and will have no correspondence with any actual variable with the same identifier.

i_1, i_2, i_3 are DO-parameters as described in Section 2-6. These parameters must be either constants or DO-control variables from outer implied DO's.

A list element in an implied DO list must be either an array element specification or another implied DO list. All implied DO lists must be enclosed within parentheses, however redundant parentheses may be used within implied DO's whenever desired. An array element must have at least one non-constant subscript. All variables used within subscript expressions must be DO-control variables of the DO-implied lists. A subscript expression may consist of any number of constants and DO-control variables combined with "+", "-" or "*". The expression must of course be a valid arithmetic expression and may not contain any other operations or parentheses.

2-8 DECLARATION STATEMENTS (CONT'D.)

DATA Variable List (Cont'd.)

Examples:

```
((A(I,J),J=1,10),I=1,10)
((A(3*I+5),I=9,7,-1)
(A(I), (B(I,J), J=1,I), C(I), I=8,10)
(((ABC(I+J,J+K), I=J,K), J=1,5), K=2,3)
```

DATA Constant List

A DATA constant list is of the form:

$C_1, C_2, C_3, \ldots, C_m$

where:

the C_i are of the following form:

c

r*c

where:

r is an unsigned integer repeat count, whose value (non-zero) indicates the number of times the constant is to be repeated, and

c is a signed or unsigned constant of an appropriate type as described in Section 2-3, DATA - Constants.

The constants may be any of the forms described in Section 2-3, including literal constants.

NOTE

Since both octal constants and literal strings may begin with a single quote, it is possible to generate an ambiguous data constant(s). For example: /'77, 1H'/ could be interpreted as an octal constant followed by a Hollerith literal or a 6 character literal (delimited by single quotes). Care should be taken to ensure that the desired constant string is correctly compiled. The compiler will assume any string delimited by single quotes to be a literal if the terminal quote is followed by other than a digit or a letter (ignoring blanks). Thus, the above example will be interpreted as a six character literal.

In general, the type of the constant must be the same as the type of the variable that it is initializing. However, the following rules apply in DATA statements.

1. Real and double-precision variables may be initialized with constants of either of those types.

2-8 DECLARATION STATEMENTS (CONT'D.)

2. Literal constants may be used with any type of variable except logical. A literal constant is broken up on a character by character basis, and depends on the number of words of storage occupied by the variable (see Storage Allocation Statements as described in this section). An integer variable may contain up to three characters, a real or double precision variable may contain up to six characters, and a complex variable may contain up to twelve characters. A literal constant may not have an effective length greater than the number of characters which will fit in the corresponding list variable.

3. If an array name is used without subscripts, then a literal constant may be specified which is longer than a single element, but may not exceed the entire length of the array. If the literal constant is not sufficient to initialize the entire array, then the element which is last initialized will be blank filled. Additional elements, within the array must then be defined with further DATA constants.

2-9 PROGRAMS AND SUBPROGRAMS

The complete set of program units that are executed together as a single job is called an executable program. An executable program consists of one main program and all required subprograms. Subprograms may be defined by the programmer, as described in this section, or may be preprogrammed and contained in the FORTRAN library.

Main Program

A main program is comprised of a set of FORTRAN statements, the first of which (other than comment lines) is not one of the following statements:

a FUNCTION statement

a SUBROUTINE statement, or

a BLOCK DATA statement,

and the last of which is an END statement. Even if a program unit contains internal subprograms, it is classified as a main program as long as the first statement is not one of the three listed above.

Main programs may contain any statement except a BLOCK DATA, FUNCTION, SUBROUTINE, ENTRY or RETURN statement. Once an executable program has been loaded, execution of the program begins with the first executable statement in the main program. Main programs may also be written in assembly language (see part IV, INTERFACING OF FORTRAN AND ASSEMBLER GENERATED MODULES).

Subprograms

Subprograms are programs that may be called by other programs. There are two broad classifications of subprograms as described in the following.

Functions

1. Statement functions
2. FUNCTION subprograms
3. Library functions (see Table 2-8)
4. Assembly Language functions (see Section IV)

Subroutines

1. SUBROUTINE subprograms
2. Assembly language subroutines (see Section IV)

Functions are referenced within expressions, and return a value; subroutines are referenced with CALL statements and do not necessarily return a value. A large number of library functions and subprograms are included in Harris FORTRAN (for further information refer to the DC 6024 FORTRAN Support Library, General Specification). Information concerning interfacing FORTRAN programs with assembly language subprograms is contained in Section IV of this specification.

Statement Functions

Statement functions are functions that can be defined in a single expression. A statement function definition has the form:

$f(d_1, d_2, d_3 \ldots, d_n) = e$

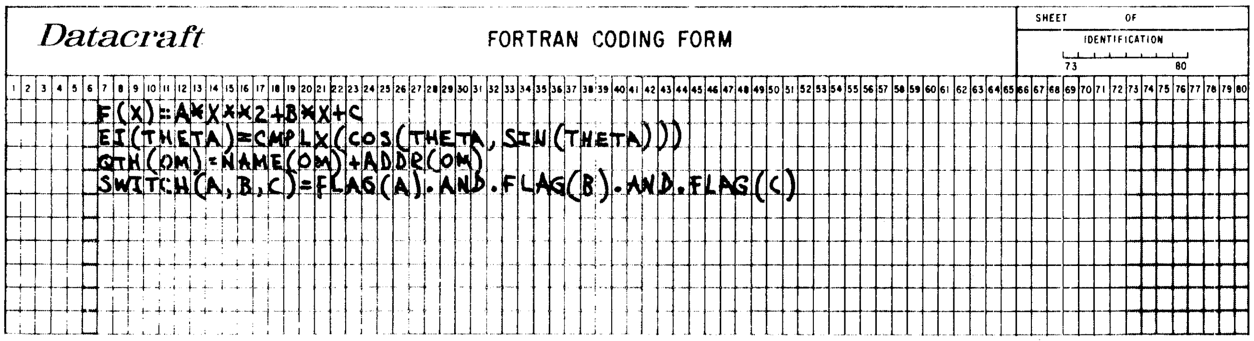
where:

f is the name of the function,

d_i is the identifier of a dummy scalar variable (see below), and

e is an expression of any mode that can legally be assigned to data of the type of f.

A statement function must have at least one dummy argument. Statement function dummies are treated only as scalars; they cannot be dummy arrays or subprograms (see "Arguments and Dummies" in this section). The expression e should contain at least one reference to each dummy. Other references in the expression are unrestricted except that the identifier f may not appear. In particular, subscripted variables may appear, as may references to other statement functions, which have been previously defined. As an example:



Datacraft FORTRAN CODING FORM

SHEET OF

IDENTIFICATION 73 80

```
      F(X)=A*X**2+B*X+C
      EI(THETA)=CMPLX(COS(THETA, SIN(THETA)))
      QTH(OM)=NAME(OM)+ADDR(OM)
      SWITCH(A,B,C)=FLAG(A).AND.FLAG(B).AND.FLAG(C)
```

Since each d_i is merely a dummy and does not actually exist, the names of statement function dummies may be the same as the names of any other entities in the program, except for the other dummies in the same statement function. Note, however, that if a statement function dummy is named X, and there is another variable in the program called X, then the appearance of X within the statement function expression refers to the dummy.

The statement function itself is typed like any other identifier: it may appear in an explicit type statement; if it does not, it will acquire an implicit type (see Section 2-8, DECLARATION STATEMENTS - Implicit Declarations).

FUNCTION Subprograms

Functions that cannot be defined in a single statement may be defined as FUNCTION subprograms. The subprograms are introduced by a FUNCTION statement of the form:

FUNCTION $f_1(d_1,d_2,\ldots,d_n),f_2(d_1,d_2,\ldots,d_n),\ldots f_n(d_1,d_2,\ldots,d_n)$

or

type FUNCTION $f_1(d_1,d_2,\ldots,dn),\ldots,f_n(d_1,d_2,\ldots,d_n)$

where:

f_i is the identifier and the name of the entrance point of the function. The identifier name f_1 is equated to the first executable statement of the function.

d_i is a dummy argument of any type of the forms described in "Arguments and Dummies". The set of different d_i's for all of the f_i's define the function's complete argument list. Only the dummy argument list associated with the identifier f_i is passed to the function in the order specified by the list of elements in f_i. Dummy arguments should be assigned values by a prior call to an associated ENTRY statement before being used in a meaningful manner.

Type is an optional type specification, which may be any of the following: INTEGER, REAL, DOUBLE PRECISION, COMPLEX or LOGICAL.

Every FUNCTION subprogram must have at least one dummy. Values may be assigned to dummies within the FUNCTION subprogram with certain restrictions (see "Arguments and Dummies").

A FUNCTION subprogram must contain at least one RETURN statement; a RETURN statement should be logically the last statement in a FUNCTION subprogram; that is, it should be the last statement executed for each execution of the FUNCTION. Control is returned to the calling program which last called an identifier f_i.

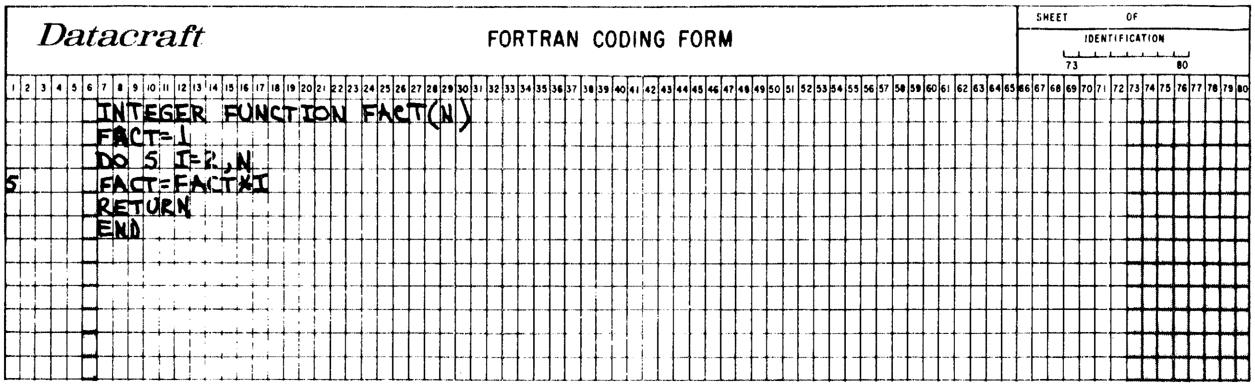
A RECUR statement must not be placed in a multiple entry FUNCTION subprogram.

Within the function f_i, the identifier of the f_1 FUNCTION subprogram is treated as though it were a scalar variable and must be assigned a value during each execution of the function. The value returned for a FUNCTION d_i is the last one assigned to the f_1 identifier prior to the execution of a RETURN statement.

FUNCTION statement examples:

```
DOUBLE PRECISION FUNCTION DIFFEQ(R,S,N)
REAL FUNCTION IWATT (W,X,Y,Z1,Z2)
FUNCTION EXTRA (N,A,B,C,V),EXTAR(N,A,B,C,V)
LOGICAL FUNCTION VERDAD(E,F,G,H,P)
FUNCTION FIRST (A,B,C,X,I),FAST(I,D,C),FASTER(R)
```

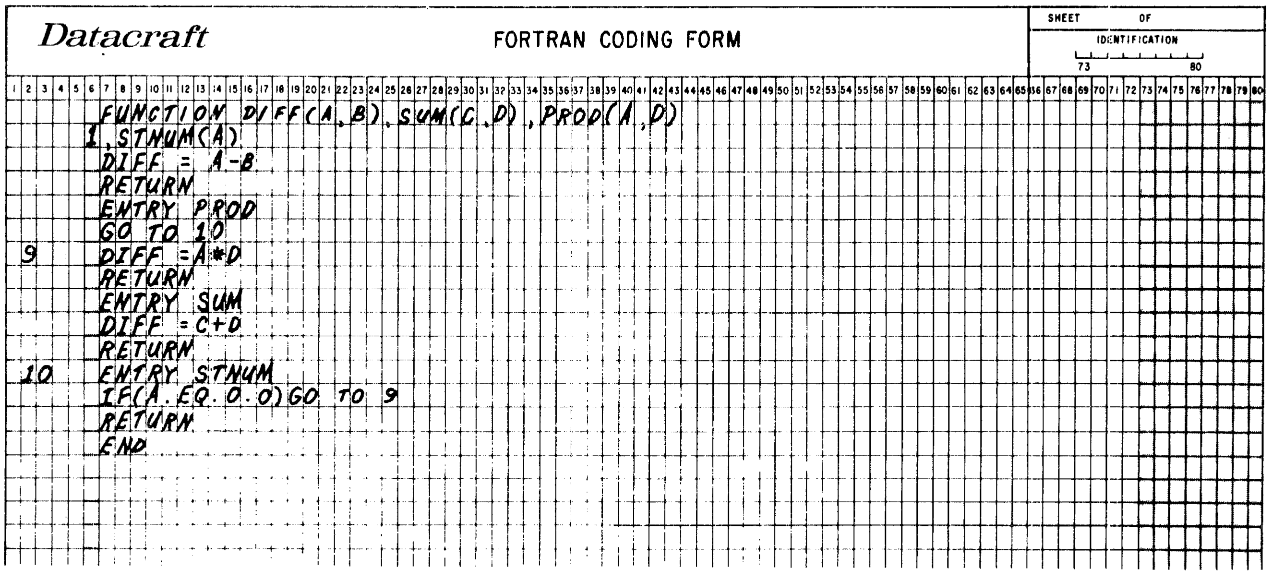
FUNCTION subprogram examples:


Datacraft FORTRAN CODING FORM

SHEET OF

IDENTIFICATION 73 80

```
      INTEGER FUNCTION FACT(N)
      FACT=1
      DO 5 I=2,N
5     FACT=FACT*I
      RETURN
      END
```


Datacraft FORTRAN CODING FORM

SHEET OF

IDENTIFICATION 73 80

```
      FUNCTION DIFF(A,B),SUM(C,D),PROD(A,D)
     1,STNUM(A)
      DIFF = A-B
      RETURN
      ENTRY PROD
      GO TO 10
 9    DIFF =A*D
      RETURN
      ENTRY SUM
      DIFF =C+D
      RETURN
 10   ENTRY STNUM
      IF(A.EQ.0.0)GO TO 9
      RETURN
      END
```

Library Functions

The DC 6024 FORTRAN library includes basic arithmetic functions, and support library subroutines. The calling sequence of these routines are automatically generated by the FORTRAN compiler. The library also contains the standard FORTRAN functions. Library subprograms are discussed in the FORTRAN Support Library General Specification. Table 2-8 gives a brief description of the standard FORTRAN library functions. The compiler recognizes the data type of the function, and makes the appropriate conversions where necessary and allowed (see Section 2-4, EXPRESSIONS - Mixed Expressions).

Table 2-8. Library Functions

Function	Definition	Number of Arguments	Name	Type	
				Argument	Function
Absolute Value	| arg |	1	ABS IABS DABS CABS	Real Integer Double Complex	Real Integer Double Real
Truncation	Sign of arg Times Largest integer $\leq$ |arg|	1	AINT INT IDINT	Real Real Double	Real Integer Integer
Choosing Largest Value	Max $(arg_1, arg_2 \ldots)$	≥ 2	AMAX0 AMAX1 MAX0 MAX1 DMAX1	Integer Real Integer Real Double	Real Real Integer Integer Double
Choosing Smallest Value	Min $(arg_1, arg_2 \ldots)$	≥ 2	AMIN0 AMIN1 MIN0 MIN1 DMIN1	Integer Real Integer Real Double	Real Real Integer Integer Double
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 Times $\| arg_1 \|$	2	SIGN ISIGN DSIGN	Real Integer Double	Real Integer Double
Positive Difference	arg_1 - Min $(arg_1$ $arg_2)$	2	DIM IDIM	Real Integer	Real Integer
Obtain Most Significant part of Double-Precision	arg.	1	SNGL	Double	Real
Obtain Real Part of Complex arg		1	REAL	Complex	Real
Obtain Imaginary Part of Complex arg		1	AIMAG	Complex	Real
Express Single-Precision arg. in Double-Precision Form		1	DBLE	Real	Double
Express Two Real args in Complex Form	$arg_1 + arg_2 \sqrt{-1}$	2	CMPLX	Real	Complex

Table 2-8. Library Functions (Cont'd.)

Function	Definition	Number of Arguments	Name	Type	
				Argument	Function
Obtain Conjugate of a Complex Arg	For arg = X+iY, C = X - iY	1	CONJG	Complex	Complex
Exponential	$e^{(arg)}$	1	EXP DEXP CEXP	Real Double Complex	Real Double Complex
Natural Logarithm	Log_e (arg)	1	ALOG DLOG CLOG	Real Double Complex	Real Double Complex
Common Log	Log_{10} (arg)	1	ALOG10 DLOG10	Real Double	Real Double
Trigonometric Sine	Sin (arg)	1	SIN DSIN CSIN	Real Double Complex	Real Double Complex
Trigonometric Cosine	Cos (arg)	1	COS DCOS CCOS	Real Double Complex	Real Double Complex
Hyperbolic Tangent	Tanh (arg)	1	TANH	Real	Real
Square Root	$(arg)^{1/2}$	1	SQRT DSQRT CSQRT	Real Double Complex	Real Double Complex
Arctangent	Arctan (arg) Arctan(arg_1/arg_2)	1 1 2 2	ATAN DATAN ATAN2 DATAN2	Real Double Real Double	Real Double Real Double
Remaindering*	arg_1 (mod arg_2)	2 2 2	DMOD AMOD MOD	Double Real Integer	Double Real Integer
Trigonometric Tangent	Tan (arg)	1	TAN DTAN	Real Double	Real Double
Arcsine	Arcsin (arg)	1	ASIN DASIN	Real Double	Real Double
Arcossine	Arccos (arg)	1	ACOS DACOS	Real Double	Real Double

*The function MOD (arg_1, arg_2) is defined as arg_1 - [arg_1/arg_2] *arg_2, where [x] is the integral part of x.

SUBROUTINE Subprograms

SUBROUTINE subprograms, like function subprograms, are self-contained programmed procedures. Unlike FUNCTIONS, however, SUBROUTINE subprograms do not have values associated with them, and may not be referenced in an expression. Instead, SUBROUTINE subprograms are accessed by CALL statements (see Section 2-6, CONTROL STATEMENTS - CALL Statement).

SUBROUTINE Subprograms (Cont'd.)

SUBROUTINE subprograms begin with a SUBROUTINE statement of the form

SUBROUTINE $p_1(d_1,d_2,\ldots,d_n),p_2(d_1,d_2,\ldots,d_n),\ldots,p_n(d_1,d_2,\ldots,d_n)$

or

SUBROUTINE $p_1, p_2, \ldots p_n$

where:

p_i is the identifier of the subroutine, and the name of the entrance point of the subroutine. The identifier name p_1 is equated to the first executable statement of the function.

d_i is a dummy argument, of any of the forms described in "Arguments and Dummies". The set of different d_i's for all of the p_i's define the subroutines complete argument list. Only the dummy argument list associated with the identifier p_i is passed to the subroutine in the order specified by the list of elements in p_i Dummy arguments should be assigned values by a prior call to an associated ENTRY statement before being used in a meaningful manner.

Note that while a FUNCTION must have at least one dummy, a SUBROUTINE may have none.

A SUBROUTINE program must contain at least one RETURN statement; a RETURN statement should be logically the last statement in a SUBROUTINE subprogram; that is, it should be the last statement executed for each execution of the SUBROUTINE. Control is returned to the calling program which last called an identifier p_i.

A RECUR statement must not be placed in a multiple entry SUBROUTINE.

A SUBROUTINE subprogram may return values to the calling program by assigning values to the d_i or to variables in common storage.

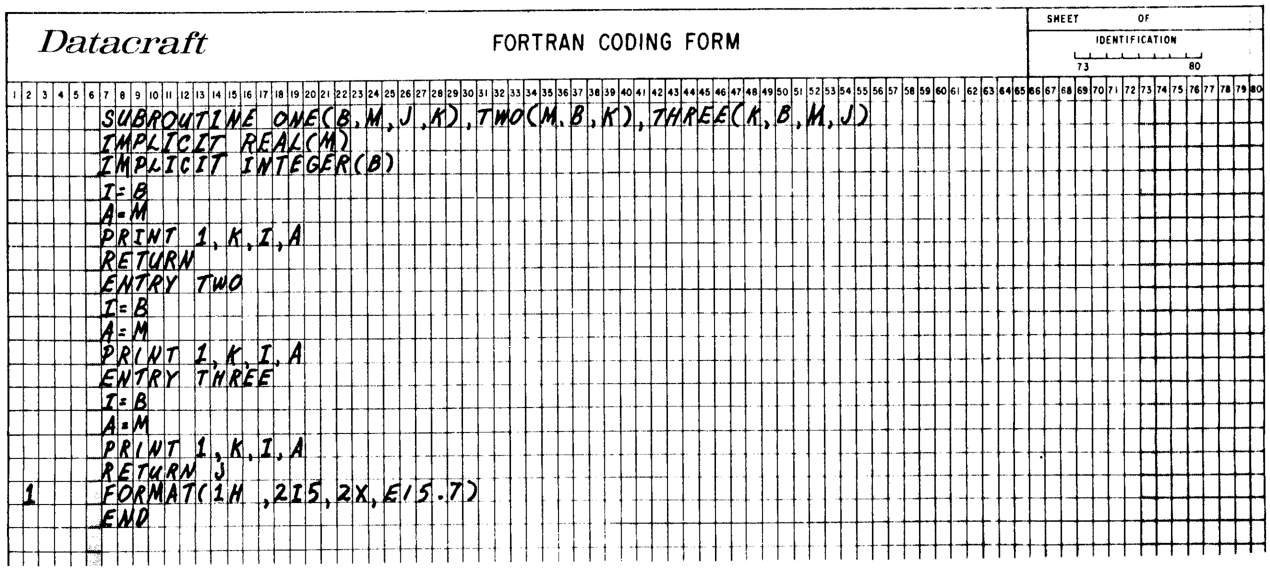
SUBROUTINE statement examples:

SUBROUTINE CHECK

SUBROUTINE ONE (B,M,J,K), TWO (M,B,K), THREE (K,B,M,J)

SUBROUTINE START (A,B,C),NOARG

SUBROUTINE subprogram example:



Datacraft FORTRAN CODING FORM

SHEET OF

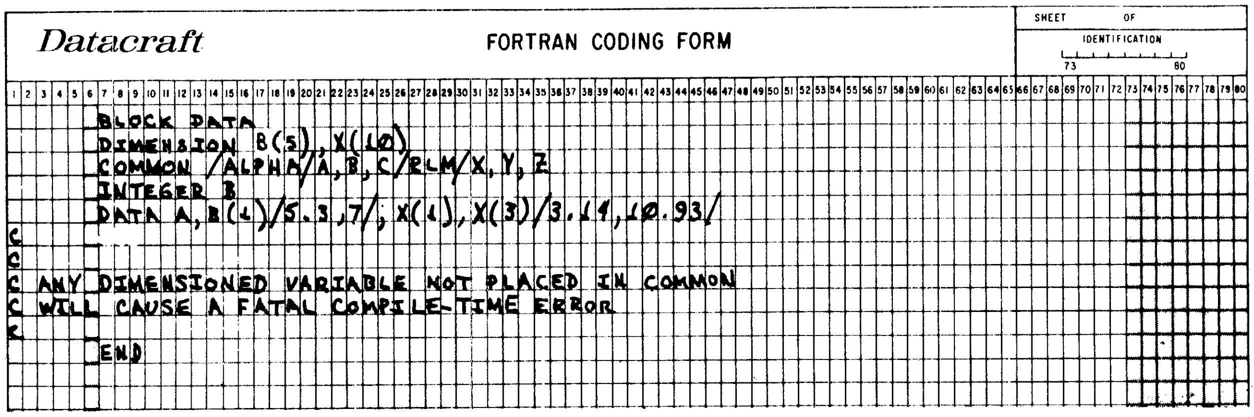
IDENTIFICATION

```
      SUBROUTINE ONE(B,M,J,K),TWO(M,B,K),THREE(K,B,M,J)
      IMPLICIT REAL(M)
      IMPLICIT INTEGER(B)
      I=B
      A=M
      PRINT 1,K,I,A
      RETURN
      ENTRY TWO
      I=B
      A=M
      PRINT 1,K,I,A
      ENTRY THREE
      I=B
      A=M
      PRINT 1,K,I,A
      RETURN J
 1    FORMAT(1H ,2I5,2X,E15.7)
      END
```

BLOCK DATA Subprograms

The BLOCK DATA subprogram must be used to initialize data in a COMMON block via the DATA statement. This subprogram contains only the DATA, COMMON, DIMENSION and type statements associated with the data being entered. The BLOCK DATA subprogram may not contain any executable statements. The first statement must be the BLOCK DATA statement.

Example:



Datacraft FORTRAN CODING FORM SHEET OF IDENTIFICATION 73 80

```
      BLOCK DATA
      DIMENSION B(5),X(10)
      COMMON /ALPHA/A,B,C/RLM/X,Y,Z
      INTEGER B
      DATA A,B(1)/5.3,7/,X(1),X(3)/3.14,10.93/
C
C
C ANY DIMENSIONED VARIABLE NOT PLACED IN COMMON
C WILL CAUSE A FATAL COMPILE-TIME ERROR
C
      END
```

ENTRY Statement (extended compiler only)

The ENTRY statement is of the form:

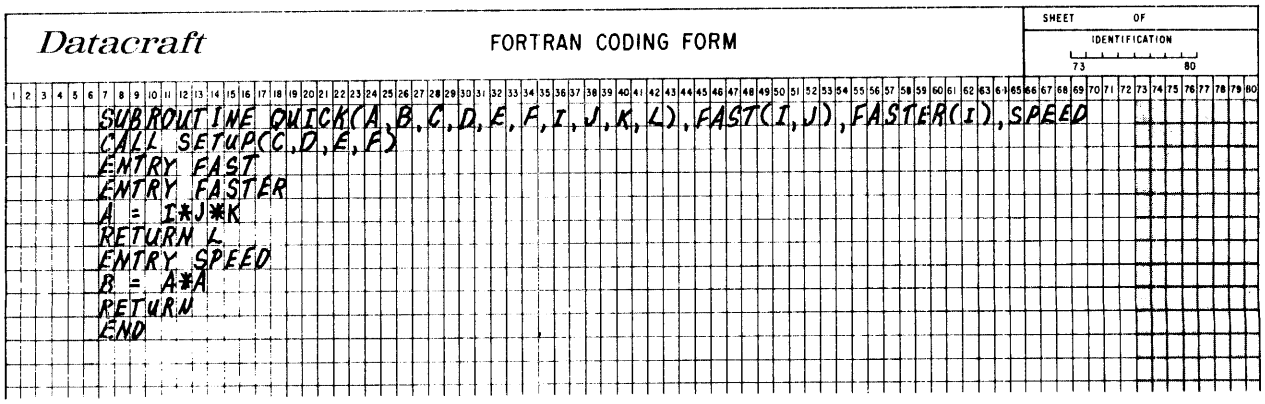
ENTRY f_i or ENTRY p_i

where:

f_i or p_i is the identifier and the name of the entrance point of the FUNCTION or SUBROUTINE subprogram. The identifiers f_1 or p_1 may not be used as alternate entry points. The dummy arguments if any specified by f_i or p_i are passed to or from the main program.

The ENTRY statement is nonexecutable and may be placed anywhere in the subprogram except as the terminal statement of a DO loop. A statement number on an ENTRY statement is optional even if an ENTRY statement immediately follows a RETURN statement. Passing control to an ENTRY statement from within a subprogram does not transfer additional arguments and control resumes with the first executable statement following the ENTRY statement.

ENTRY Examples:



Datacraft FORTRAN CODING FORM SHEET OF IDENTIFICATION 73 80

```
      SUBROUTINE QUICK(A,B,C,D,E,F,I,J,K,L),FAST(I,J),FASTER(I),SPEED
      CALL SETUP(C,D,E,F)
      ENTRY FAST
      ENTRY FASTER
      A = I*J*K
      RETURN L
      ENTRY SPEED
      B = A*A
      RETURN
      END
```

Arguments and Dummies

Dummy arguments provide a means of passing information between a subprogram and the program that called it. Both FUNCTION and SUBROUTINE subprograms may have dummy arguments, but a SUBROUTINE need not have any, while a FUNCTION must have at least one. Dummies are merely "formal" parameters, and are used to indicate the type, number, and sequence of subprogram arguments. A dummy does not actually exist, and no storage is reserved for it; it is only a name used to identify an argument in the calling program. An argument may be any of the following:

a scalar variable

an array element

an array name

an expression

a statement label

a literal constant

a subprogram name

A dummy itself may be classified within the subprogram as one of the following:

a scalar variable

an array

a subprogram

Table 2-9 indicates the permissible kinds of correspondence between an argument and a dummy.

Table 2-9. Permissible Correspondence Between Arguments and Dummies

Argument	Scalar	Dummy Array	Subprogram
scalar or array element	Yes	Yes	No
expression	Yes	No	No
statement label	Yes	No	No
array name	Yes	Yes	No
literal constant	Yes	Yes	No
subprogram name	No	No	Yes

A statement label argument is written as:

$k

where:

k is the actual statement label, and

$ distinguishes the item as a statement label (as opposed to an integer constant).

Arguments and Dummies (Cont'd.)

Within a subprogram, a dummy may be used in much the same way as any other scalar, array, or subprogram identifier with certain restrictions; namely, dummies may not appear in COMMON, EQUIVALENCE or DATA statements.

In general, dummies must agree in type with the arguments to which they correspond. For example, the following situation is an error because the types of the arguments and the dummies do not agree.

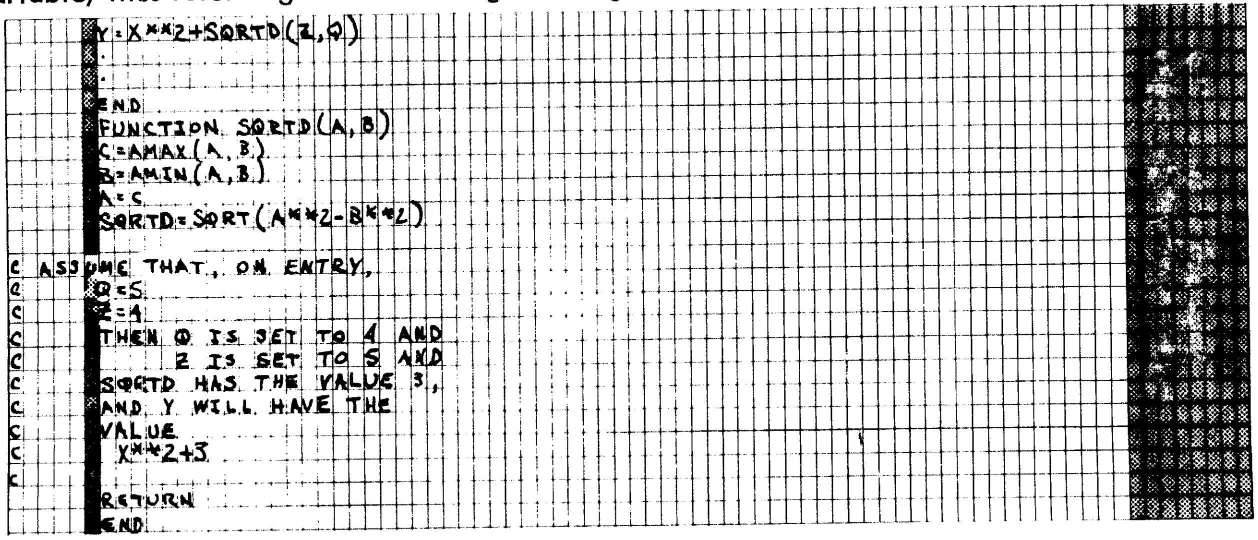
```
COMPLEX C             FUNCTION F(LL,CC)

LOGICAL L             LOGICAL LL

X = F (C,L)           COMPLEX CC
  .                     .
```

Reversing the order of either the arguments in the calling reference or the dummies in the FUNCTION statement would eliminate the error in this example.

There are three exceptions to the rule of type correspondence.

1. A statement number passed as an argument must correspond to a dummy variable of integer type.
2. A SUBROUTINE name (as opposed to a FUNCTION name) has no type.
3. A literal constant also has no type and may be received by any type of dummy array.

When a dummy corresponds to a variable in the calling argument list, a reference to the dummy is actually a reference to the argument variable. Thus, not only will the dummy initially have the value to which the argument variable was assigned at the time of the call, but any value subsequently assigned to the dummy will actually be assigned to the argument variable, thus returning a result through the argument list. For example:



```
      Y=X**2+SQRTD(Z,Q)
      .
      .
      END
      FUNCTION SQRTD(A,B)
      C=AMAX(A,B)
      B=AMIN(A,B)
      A=C
      SQRTD=SQRT(A**2-B**2)

C ASSUME THAT, ON ENTRY,
C     Q=5
C     Z=4
C     THEN Q IS SET TO 4 AND
C             Z IS SET TO 5 AND
C     SQRTD HAS THE VALUE 3,
C     AND Y WILL HAVE THE
C     VALUE
C       X**2+3
C
      RETURN
      END
```

Arguments and Dummies (Cont'd.)

where the values of Z and Q will be reversed whenever the initial value of Q is greater than that of Z.

On the other hand, when a dummy corresponds to an expression, the expression serves merely to initialize the value of the dummy, and consequently the value of the dummy should not be changed within the subprogram. Note that a single constant is a form of expression, as is a function reference (as opposed to a function name alone). For example, if X is a scalar variable and F is a function:

```
CALL ALPHA(X, 4.6, F, F(X))          SUBROUTINE ALPHA (A,B,C,D)
```

the dummy scalars B and D must not be assigned values within ALPHA, while the dummy A may be. The dummy C must be used within the subprogram as a subprogram identifier.

When a dummy corresponds to a statement label, the effect is the same as:

```
ASSIGN k TO d
```

where:

k is the statement label and d is the dummy.

A dummy, such as k, may be used in a GO TO statement.

Dummy Scalars

Dummy scalars are single valued entities that correspond to a single element in the calling program.

Dummies that are not declared to be arrays, or subprograms, are treated as scalars.

Dummy Arrays

A dummy argument may be defined to be an array by the presence of its identifier in an array declaration within the subprogram (the fact that a calling argument is an array does not in itself define the corresponding dummy to be an array). As with all dummies, a dummy array does not actually occupy any storage; it merely identifies an area in the calling program. The subprogram assumes that the argument supplied in the calling statement defines the first (or base) element of an actual array and calculates subscripts from that location.

Dummy Arrays (Cont'd.)

Normally, a dummy array is given the same dimensions as the argument array to which it corresponds. This is not necessary however, and sometimes useful operations can be performed by making them different. For example:

```
DIMENSION A(10,10)          SUBROUTINE OUT(B)
CALL OUT (A(1,6))           DIMENSION B(50)
  .                           .
  .                           .
  .                           .
```

In this case, the 1-dimensional dummy array B corresponds to the last half of the 2-dimensional array A i.e., elements A(1,6) through A(10,10). However, since an array name used without subscripts as an argument refers to the first element of the array, if the calling statement were

```
CALL OUT (A)
```

the dummy array B would correspond to the first half of the array A.

Arguments that are literal constants are normally received by dummy arrays (they may also be passed to assembly language subprograms). A literal constant is stored as a consecutive string of characters in memory, and its starting location is passed as the argument address. Thus, in the example:

```
CALL FOR (13HPHILIP MORRIS)          SUBROUTINE FOR(M)
                                     INTEGER M(5)
  .
  .
  .
```

the following correspondences hold:

M(1) = 3HPHI

M(2) = 3HLIP

M(3) = 3Hb̸MO

M(4) = 3HRRI

M(5) = 3HSb̸b̸

where:

b̸ represents the character blank.

Dummy Arrays (Cont'd.)

Literal Constants are filled out with trailing blanks to the nearest double word boundary (multiple of six characters). Therefore, passing such a constant to a dummy of a type that occupies more than one word per element (e.g., double precision) will normally result in a valid definition of the entire dummy. Thus, 1HX when passed to real variable has the same effect as 6H Xb̸b̸b̸b̸b̸. Note this does not always apply to a complex dummy, therefore it may result in an un-defined imaginary part to the dummy.

If an array corresponds to something that is not an array or a literal constant, the latter will correspond to the first element of the array. This is true whether the calling argument is an array and the dummy is not, or vice versa. Thus, for example, if the calling argument is a scalar and the dummy is an array, references in the subprogram to elements of the array other than the first element will correspond to whatever happens to be stored near the scalar. Obviously, care must be taken in creating correspondences of this nature, as they may depend upon a particular implementation.

Adjustable Dimensions

Since a dummy array does not actually occupy any storage, its dimensions are used only to locate its elements, not to allocate storage for them. Therefore, the dimensions of a dummy array do not have to be defined within the subprogram in the normal manner. Instead, any or all the dimensions of a dummy array may be specified by scalar variables rather than by constants. This permits the calling program to supply the dimensions of the dummy array each time the subprogram is called. The following statements demonstrate adjustable dimensions:

```
DIMENSION P(10,5), Q(9,3)        FUNCTION SUM (R, N, M)
X = SUM(P, 10,5)                 DIMENSION R(N,M)
Y = SUM(Q, 9,3)                  .
.                                .
.                                .
.
```

Only a dummy array may be given adjustable dimensions, and the dimensions must be specified by integer scalars. These scalars must be dummies or COMMON variables only.

The variable used as adjustable dimensions may be referenced elsewhere in the subprogram but should not be changed.

Dummy Subprograms

A dummy subprogram must correspond to an argument that is a subprogram name, and it is the only kind of dummy that may do so. The dummy name merely serves to identify a subprogram whose actual location is defined by the calling program. Therefore, a call on a dummy

Dummy Subprograms (Cont'd.)

subprogram is actually a call on the subprogram whose name is specified as the argument. A dummy subprogram is classified in the same manner as any other subprogram (see Section 2-3, DATA - Identifiers).

Examples of dummy subprograms are:

```
EXTERNAL SIN, DSIN, SQRT, DSQRT        FUNCTION DIFF(F,DF,Z)

A = DIFF (SIN, DSIN, X)                DOUBLE PRECISION DF

B = DIFF(SQRT, DSQRT, Y)               DIFF = DABS(F(Z) - DF(Z))

                                       RETURN
.
.
.                                      END
```

A subprogram identifier to be passed as an argument must be identified as a subprogram. That is, it must be used as a subprogram and/or appear in an EXTERNAL statement. Otherwise it will be classified as a scalar variable.

<u>RECUR Statement</u> (Extended Compiler Only)

The RECUR statement simplifies writing of recursive subprograms in FORTRAN.

The format is:

```
RECUR n
```

Where n is an integer constant which specifies the maximum number of levels of recursion. The code generated by the compiler saves the return addresses in a stack. If the subprogram has arguments, the argument pointers are updated each time the subprogram is re-entered. Any stacking of arguments or intermediate values must therefore be programmed by the user.

<u>NAME Statement</u> (Extended Compiler Only)

The format for the NAME statement is as follows:

```
NAME n
```

Where n is an identifier. (See Identifiers, Section 2-3). The purpose of this capability is to allow FORTRAN main programs to be cataloged by the DOS EDITPF processor, or be cataloged in the NAME file by the EDITNF processor.

SECTION III
FORTRAN DIAGNOSTIC MESSAGES

3-1 GENERAL DESCRIPTION

The Harris FORTRAN system includes an extensive set of compile-time and run-time diagnostic messages. Compile-time diagnostics include in the message a complete phrase which normally will indicate to the programmer the exact nature of the error as well as the point in the source statement where the error was discovered. The run-time diagnostics do not produce a verbose message, but attempt to pin point the location in the source program where the error occurred. This section explains the format of both compile-time and run-time diagnostics, and lists all possible reasons which might cause the error conditions to occur.

3-2 COMPILE-TIME DIAGNOSTICS

Diagnostic messages will appear on the list output file in-line with the source statements. In most cases, the diagnostic message will appear on the line immediately following the questionable statement. Although this is not always possible due to the one-pass nature of the compiler, the "snapshot" feature should pinpoint the location of the questionable text.

The format of a compile-time diagnostic message is:

```
ERROR     XX     YYYYYY     message
```

or

```
NOTE      XX     YYYYYY     message
```

where:

XX is an octal number corresponding to the type of error, and

YYYYYY is the last six characters encountered within the statement when the discrepancy was discovered.

The message may be up to 30 characters in length and generally indicates the exact nature of the error. Table 3-1 is a list of all compile-time diagnostics and a description of all conditions which might cause the error.

An ERROR outputs a code to the link loader to prevent loading. A NOTE allows linking to proceed.

Table 3-1. Compile-Time Diagnostic Messages

Error Number	Message	Cause
0	DATA POOL OVERFLOW	The entire data storage area available to the compiler is filled. This includes symbol table and all other table storage combined, since table storage is allocated dynamically. This is the only error condition which prevents compilation from continuing.
1	INVALID OPERATOR	a) 2 or more consecutive operators b) A special character not recognized as an operator c) Using .NOT. as a leading operator following an arithmetic or relational operator. d) an attempt to shift or rotate by a value that is not an integer constant whose absolute value is less than 24.
2	INVALID CONSTANT	a) Too many digits in a constant. b) Magnitude of a real or double precision constant is out of range. c) an octal constant contains an "8" or "9" d) a constant appears where an identifier is expected e) A non-zero constant does not appear where expected.
3	INVALID SYNTAX	The compiler has accepted the statement as a legitimate FORTRAN statement, but the construction of some particular element does not conform to the rules of the language. In particular, this message is caused by: a) Incorrectly formed exponent in a real or double precision constant. b) Unrecognizable logical operator. c) Identifier with more than 6 characters.* d) Invalid construction of a statement function e) Invalid use of "=" (equal sign). f) Invalid construction of a DO or ASSIGN statement. g) Two consecutive relational operators not separated by a logical operator. h) Invalid construction of an argument list.
4	MISSING OPERAND	An identifier or constant does not exist in the text at a point where expected.
5	RETURN STATEMENT	a) A RETURN statement has been recognized in a main program b) A subprogram contains no RETURN statement.

* WARNING

Table 3-1. Compile-Time Diagnostic Messages (Cont'd.)

Error Number	Message	Cause
6	INVALID STATEMENT NUMBER	a) Incorrectly constructed statement number. b) A statement number contains more than 5 numeric characters. c) A reference exists to an undefined statement number. These errors are discovered immediately following the output of the statement number map. d) More than one statement has the same statement number.
7	DATA IN COMMON	The variable list of a DATA statement contains an item which has been allocated common storage and the program is not a BLOCK DATA subprogram.
10	SUBSCRIPT USAGE	a) An array is declared using variable subscripts which are not dummies (i.e., do not appear as arguments of a subroutine or FUNCTION statement. b) An array element is accessed whose number of subscripts do not match the declared number of subscripts for that array.
11	INVALID STATEMENT	The statement is not recognized as a legitimate FORTRAN statement or the source program was not properly terminated with an END$ statement.
12	PARENTHESIS	a) A left or right parenthesis does not occur where expected. b) The number of left and right parenthesis within an expression do not agree in number.
13	MIXED MODES	Invalid mixing of item modes (data types) within a) an expression b) an assignment statement c) a DATA statement
14	INVALID DELIMITER	a) An invalid special character has caused the processing of a statement to terminate. b) The "/" character does not appear where expected in a DATA or COMMON statement. c) In an expression, an invalid character follows an array or SUBPROGRAM identifier. (Only "," and "(" are acceptable).
15	INVALID DATA SPECIFICATION	a) A DATA statement variable list does not contain the same number of items as its corresponding constant list. b) An H-specification is too large to fit into its corresponding variable. c) A DATA statement in a BLOCK DATA subprogram is attempting to initialize data into a non-COMMON variable.

Table 3-1. Compile-Time Diagnostic Messages (Cont'd.)

Error Number	Message	Cause
15	INVALID DATA SPECIFICATION (Cont'd)	d) An H-specification that is preceded by a repeat-count is contained on more than one card by use of a continuation line.
16	UNDEFINED VARIABLE	a) An identifier has been used but not defined. b) An ENTRY identifier is not defined in a subprogram statement. c) A subprogram identifier has not been referenced by an ENTRY statement.
17	INVALID ITEM USAGE	a) An identifier previously defined as a variable subroutine or array is used in a context that requires a different item usage. b) A constant is used where an identifier is expected. c) An ENTRY point has multiple definition.
20	INVALID LOGICAL IF	A logical IF statement is used as the executable statement of a logical IF statement.
21	MISSING STATEMENT NO.	A FORMAT statement is unlabeled.
22	INVALID H SPECIFICATION	An H-specification is longer than the remaining characters in the statement.
23	INVALID BLOCK DATA PROGRAM	An executable statement has been detected in a BLOCK DATA subprogram.
24	INVALID COMMON USAGE	Multiply defined common variables.
25	INVALID MODE	a) Logical operation attempted using other than logical or integer typed variables. b) Arithmetic operation attempted on logical variables.
26	FUNCTION NEEDS ARGUMENTS	A FUNCTION statement has been encountered that does not specify any arguments.
27	EQUIVALENCE	a) Two items are equivalenced, both of which have been allocated to a COMMON block. b) An equivalence group demands the extension of a COMMON block in a negative direction. c) Invalid construction of an EQUIVALENCE statement. d) Contradicting equivalence groups.
30	STATEMENT ORDER	a) Specification statement follows DATA, statement function, or executable statement. b) DATA statement follows statement function or executable statement.

Table 3-1. Compile-Time Diagnostic Messages (Cont'd.)

Error Number	Message	Cause
30	STATEMENT ORDER (Cont'd.)	c) Statement function follows an executable statement. d) FUNCTION, SUBROUTINE, or BLOCK DATA statement is not the first sta tement of a program.
31	NO PATH TO HERE	An executable statement which is not labeled with a statement number immediately follows either an arithmetic IF or a GO TO statement. *
32	INVALID DO	a) The terminal statement of a range of a DO is a GO TO, arithmetic IF, RETURN, PAUSE, STOP, or DO statements. b) Illegally nested DO statements. c) An error in an implied DO within an I/O list.
33	MODULE OVERFLOW	This segment has exceeded 32,767 decimal locations.

* WARNING

Scalar variables that appear only to the right of the equal sign in assignment statements cause a WARNING message to be generated. Subsequent execution of the compiled program is not inhibited. The last line of compiler list output is formated as follows:

```
XXX ERRORS          SIZE   YYYYY   ZZZZZ
```

Where XXX is the number of errors discovered during compilation. (This output does not appear if no errors have been discovered.) YYYYY is the module size expressed as a decimal integer. ZZZZZ is the module size expressed as an octal integer.

3-3 RUN-TIME DIAGNOSTICS

If an error condition occurs during the execution of a FORTRAN program or any program that calls functions from the FORTRAN library, an error procedure is initiated. Depending upon the operating system and the status of the execution options, an error message will be output and/or an abort will occur (see appropriate reference manual). The format of the error message (if present) is one of the following:

```
SAU  xx  @  yyyyyy

FER  xx  @  yyyyyy
```

where:

xx is a two digit error number

yyyyyy is an octal address which is the return address of the routine in which the error occurred.

Table 3-2 is a list of all run-time diagnostics. A description of the result of the operation if not aborted is also given.

3-3 RUN-TIME DIAGNOSTICS (CONT'D.)

An additional debugging aid is also provided for execution errors that are not readily diagnosed. Upon the occurrence of a run-time error, in addition to the normal error message, a source level error location may be produced. In order for this to occur, a compile time option must be set during compilation of the program which is generating the error. If this option was set during compilation (see option "W" or B11 in Section 3-4), and an error occurs, the following message will be output:

```
--> ERROR IN xxxxxx AT STATEMENT NUMBER yyyyyy + zzzzzz LINES.
```

where:

xxxxxx is the name of the FORTRAN routine in which the error occurred. *MAIN* indicates that the error occurred in the main program. Note: If the error occurred in a library routine or in a FORTRAN subprogram that was not compiled with the "W" or B11 option, then this message will indicate the last statement executed in a module which <u>was</u> compiled with the "W" or B11 option, prior to the error.

yyyyyy is the last statement number encountered prior to the statement in which the error occurred. If the error occurred in a statement with a statement number, then yyyyyy is that statement number. Note, that in this context, prior means the statement physically preceding and does not necessarily indicate execution sequence.

zzzzzz is the number of the statement relative to yyyyyy in which the error occurred. If the error occurred in statement yyyyyy, then zzzzzz is 0. If it occurred after yyyyyy, then zzzzzz is one more than the number of physically intervening statements between yyyyyy and the statement in error. For the purpose of this error message, the first statement of a program or subprogram is considered as "   0 +   1 LINES. "

Note: zzzzzz etc., does <u>not</u> include continuation statements, comment cards or the object of a logical IF as additional statements.

If the error occurred in a subprogram which was called thru one or more levels of subprograms from the main program, then each of these calls will be indicated by the following message (if the calling routine was compiled with option "W" or B11 set):

```
--> CALLED BY xxxxxx AT STATEMENT NUMBER yyyyyy + zzzzzz LINES.
```

where xxxxxx, yyyyyy and zzzzzz are as defined above.

Table 3-2. Run-Time Diagnostics

Error Number	Explanation	Action if Not Aborted
01	Square root or a negative number.	result = 0
02	An FXA instruction was executed such that the integer result could not be contained in register A.	result = FSP or FSN
03	Division by zero.	result = FSP
04	Arithmetic underflow (generated by add, subtract, multiply, divide or square).	result = 0
05	Arithmetic overflow (generated by add, subtract, multiply, divide or square).	result = FSP or FSN
06	SIN or COS of a number so large that all significance is lost.	result = 0
07	ATAN2 or DATAN2 called with both arguments zero.	result = 0
08	Logarithm of zero or a negative number.	result = 0
09	Underflow during EXP or DEXP.	result = 0
10	Overflow during EXP or DEXP.	result = FSP
11	Underflow during exponentiation (X**Y).	result = 0
12	Overflow during exponentiation (X**Y).	result = FSP or FSN
13	Exponentiation error (00**0 or neg**0).	result = 1
14	Exponentiation error (0**neg).	result = FSP
15	Exponentiation error (neg**neg or neg**pos).	result = 0
16	ASIN or ACOS with |argument| >1.0.	result = 0
41	Invalid output FORMAT specification.	Output terminated
42	Invalid input FORMAT specification.	Input terminated
43	Illegal character during input.	Character ignored
44	Underflow during numeric input conversion.	result = 0

Table 3-2. Run-Time Diagnostics (Cont'd.)

Error Number	Explanation	Action if Not Aborted
45	Overflow during numeric input conversion.	result = FSP or FSN
46	More than 10 BUFFER IN or OUT files.	operation not performed
47	More than 10 files were defined via the DEFINE FILE statement.	operation not performed
48	Attempted READ, WRITE, or FIND a record on an undefined random access file.	operation not performed
49	Attempted READ, WRITE, or FIND a record number larger than the defined number of records in the random access file.	operation not performed

NOTE:	FSP	=	'37777777 for integer results
		=	'37777777, '37777577 for all SAU and Double precision non-SAU results
		=	'37777777, '00000177 for Real non-SAU results
	FSN	=	'40000001 for integer results
		=	'40000000, '00000577 for all SAU and Double precision non-SAU results
		=	'40000001, '00000177 for Real non-SAU results

3-4 IN-LINE CONTROL STATEMENTS

The Extended Harris FORTRAN compiler processes in-line control statements. This extension is not available in the Basic Harris FORTRAN compiler. An in-line control statement is identified by a colon (:) in column 1 of a source statement. Like other FORTRAN statements, columns 73-80 of in-line control statements are not processed. In-line control statements are divided into two groups:

1. Conditional-compile control statements, which provide means for skipping (not compiling) a set of source statements if a specified condition is true.

2. Option control statements, which provide means for turning various compile-time options "on" or "off" at any point in the program.

A program may contain any number of in-line control statements. Any error detected in an in-line control statement terminates the processing of that in-line control statement. The error is considered to be a non-fatal compilation error and a warning message is issued.

3-4. 1 Conditional-compile Control Statements

Conditional-compile control statements provide means for conditional compilation of source statements. They are divided into two groups: Skip-statements and an ESKP-statement.

A conditional-compile block consists of a set of statements, the range of the conditional-compile block, enclosed between a skip-statement and an ESKP-statement. A skip-statement opens a conditional-compile block, while an ESKP statement closes it. A conditional-compile block may contain any number of conditional-compile blocks in its range, i. e, conditional-compile blocks may be nested to any level. The range of a conditional-compile block may be empty. When conditional-compile blocks are nested, the first encountered ESKP-statement--closes the last opened conditional-compile block, the second ESKP-statement closes next-to-last opened conditional-compile block, etc.. Each conditional-compile block opened in the program must be closed.

3-4. 1. 1 Skip-Statements

The skip-statement has the form:

:SKFS $f_1, f_2, \ldots, f_i, \ldots,$ fn

or

:SKFZ $f_1, f_2, \ldots, f_i, \ldots,$ fn

Where f_i is a flag number, an unsigned integer constant, in the range of 0 and 23.

Examples:

```
:SKFS   1, 10, 2, 3
:SKFZ   20, 1
:SKFZ   2
```

The list of flag numbers specified on a skip-statement constitute the skip condition for the associated conditional-compile block. Compilation of a skip-statement causes evaluation of the specified skip condition. If the skip condition is true, statements in the range of the associated conditional-compile block are skipped (not compiled or ignored) by the compiler. Compilation is resumed at the ESKP statement of the skipped conditional-compile block. If the skip condition is false, normal compilation sequence continues.

In case of an SKFS statement, the skip condition is true only if all flags specified by it are "on". In case of an SKFZ statement, the skip condition is true only if all flags specified by it are "off". In all other cases the skip condition is considered false.

If the skip condition is true for a specific conditional-compile block, any conditional-compile blocks in its range will be skipped regardless of their skip condition. If an error is detected in a skip-statement, a warning message is issued and the statement is ignored by the compiler.

Under DMS/DOS/TOS/ROS, flags for compilation are specified using $FLAGS statement of the Job Control.

Example:

```
$FLAGS     .0,1
$ASSIGN    7, INPUT
$FORTRAN
```

Under VULCAN, these flags are specified using the FLAGS specification on the Job Control statement or processor call statement. It should be the last parameter on the statement, thus it should follow the AREANAME if an area name is specified. The form of this specification is:

```
FLAGS=     n, n, ..., n
```

where the n's are the flag numbers to be set during compilation. All other flags are reset.

Example:

```
$FO. EM, INPUT, FLAGS = 0,1
$$*FORTRAN. E, F=2
```

3-4.1.2 ESKP-Statement

The ESKP-Statement has the form:

```
:ESKP
```

Compilation of an ESKP-statement does not have any effect. Normal compilation sequence continues. An ESKP-statement must always close a conditional-compile block, i.e., each ESKP-statement must be preceded by a matching skip-statement. If an error is detected in an ESKP-statement, a warning message is issued and the statement is ignored by the compiler.

3-4.1.3 Use of Conditional-Compilation

The conditional-compilation technique may be used in producing different versions of a program, each having different capabilities and characteristics. It provides an important tool in producing diagnostic versions of a program. Using this technique, debug or diagnostic statements are made a permanent part of the program source. These statements are placed in the range of one or more conditional-compile blocks. Each of these blocks is controlled by one or more compilation flags. These blocks may be set up in such a way as to provide different levels of debugging information for different combinations of compilation flags. For normal execution purposes, all debugging information is excluded from the program during compilation, by selecting an appropriate combination of compilation flags. A diagnostic version of the same program may also be generated without making any changes in the program source. This is achieved by recompiling the same program with appropriate compilation flags set to compile the desired debugging information.

The following example illustrates the use of in-line conditional-compile control statements:

Example:

```
          DIMENSION  ID  (6)
          .
          .
          DO  1  I=1,6
          CALL GETCHR  (ICHAR,MODE)
:SKFZ     2
          WRITE (-, 100) I,ICHAR,MODE
100       FORMAT (I6, A6, I6)
:ESKP
          IF (MODE. LT. 0) GO TO 2
          ID (I) = ICHAR
1         CONTINUE
          CALL GETCHR (ICHAR,MODE)
          IF (MODE. GE. 0) CALL ERROR
2         CONTINUE
:SKFZ     1
          WRITE (-,101) ID, ICHAR
101       FORMAT (6A1, A6)
:ESKP
```

The above example obtains a FORTRAN identifier in the singly dimensioned array ID and its delimiter in ICHAR. Subroutine GETCHR returns next character from input stream and its MODE. The example contains two conditional-compile blocks, set up to produce two levels of debugging information. The first conditional-compile block is controlled by the compilation flag number 2. If this flag is "on" at compile time, the first conditional-compile block will be compiled. This will cause I, ICHAR, and MODE to be printed during each iteration of the DO-loop. If the flag number 2 is "off" at compile time, this output is not produced during execution. The second conditional-compile block is controlled by the compilation flag number 1. If this flag is "on" at compile time, the second conditional-compile block will be compiled. This will cause the identifier ID and the delimiter ICHAR to be printed. If the flag number 1 is "off" at compile time, this output will not be produced during execution.

3-4.2 Option Control Statements

The Harris FORTRAN Compiler provides various options for compilation of source programs. These compile-time options may be turned "on" or "off" using in-line option control statements.

The option control statement has the form:

:n_1, n_2, . . . , n_i, . . . , n_n

or

:NO n_1,n_2, . . . , n_i, . . . , n_n

where n_i is a compile-time option name.

A compile-time option name consists of two or more alphanumeric characters, the first of which is a letter. Only the first two characters of a compile-time option name are used to identify the compile-time option.

The first form of the option control statement defines an option-on control statement, compilation of which causes the specified compile-time options to be turned "on". The second form of the option control statement-defines an option-off control statement, compilation of which causes the specified options to be turned "off". The ASsembly mode option or the ENd assembly mode option may not be used in an option-off control statement.

Examples:

```
:LIST, OBJECT LISTING, TR
:NO OBJ,MAP OUTPUT, WALKBACK
```

Each compile-time option, excluding the ASsembly mode and ENd assembly mode options, is assigned a bit in the Job Control option word. Under DMS/DOS/TOS/ROS, each option-bit (0-23) is assigned a system option number (0-23), and it may be turned "on" or "off" using the $OPTION command of the Job Control. Under VULCAN, each option-bit (0-23) is assigned a single character Job Control option (A-X), and it may be turned "on" or "off" using the SO (Set Options) command of the Job Control. In this case, option-bits may also be turned "on" using the options-specification field of Job Control commands or Processor Calls.

The value of a compile-time options remains unchanged during compilation until changed using in-line option control statements. Most of the compile-time options may be changed (turned "on" or "off") at any point in the program. However, a group of compile-time options may be changed for a new program unit (main program or subprogram) only before any FORTRAN statement, excluding comments, is compiled for the new program unit. That is, these compile-time options may not be changed within a program unit. Compile-time options included in this group are:

1. DOuble precision option
2. LOng address generation option
3. WALK-back code generation option
4. FATAL compilation errors only option

During compilation, option names specified on an option control statement are processed from the left to right. After processing each option name, the associated compile time option is turned "on" or "off". If an error is detected in an option control statement, processing of the option control statement is terminated. However, any compile-time option changes performed prior to the occurrence of the error remain effective.

3-4.3 Initial Values of Compile-Time options

At the start of compilation of a program, the FORTRAN Compiler obtains the Job Control option word from the operating system. The value of each option-bit in the option word determines the initial value of the associated compile-time option. In most cases, the compile-time option is set to the value of the associated option-bit, i.e., if the option-bit is "on", the associated compile-time option is turned "on", etc. In other cases, the compile-time option is set to the opposite value of the associated option-bit, i.e., if the option-bit is "on", the associated compile-time option is turned "off", etc. The ASsembly mode option and the ENd assembly mode option are initially turned "off".

Table 3-3 lists all compile-time options with their two character option names, associated system option under DMS/DOS/TOS/ROS, associated Job Control option under VULCAN, and the value of the compile-time option when the associated option is "on".

3-5 COMPILE-TIME OPTIONS

This section describes various compile-time options processed by the FORTRAN compiler.

1. DOuble precision option - (Extended Compiler Only) - If this option is "on" at compile time, all implicit real variables and constants are typed as double precision, and calls to FORTRAN real functions are replaced by calls to their double precision counterparts.

   Example: If the Double precision option is "on",

   ```
   A = SIN (B) +1.3
   ```

   is equivalent to:

   ```
   DOUBLE PRECISION

   A = DSIN (B) + 1.3D0
   ```

   This option may not be changed within a program unit using in-line option control statements.

Table 3-3 Compile-Time Options

OPTION	Option name (O)	Associated Option (A) under		Relationship between O and A
		VULCAN	DMS/DOS/TOS/ROS	
*Double Precision	DO	D	4	O=A
Object Listing	OB	O	5	O=A
Extended Object Listing	EX	X	6	O=A
Map Output	MA	M	8	O=. NOT. A
*Long Address Generation	LO	L	9	O=A
Triad Output	TR	T	10	O=A
*Walk-Back Code Generation	WA	W	11	O=A
Abnormal Subscript Redefinition	AB	A	12	O=A
Additional Subscript Optionization	SU	S	13	O=A
Computed GOTO Checking Code Generation	GO	I	14	O=. NOT. A
Debug Statement Compilation	DE	G	15	O=A
Ignore Blank Statements	IG	B	16	O=. NOT. A
Skipped Statement Listing	SK	K	17	O=. NOT. A
Indented Listing	IN	P	18	O=A
Optional Control Statement Listing	OP	R	19	O=. NOT. A
Source Statement Listing	LI	E	20	O=. NOT. A
Sequence Number on the Left	SE	N	21	O=A
Character Code Conversion	CH	C	22	O=A
*Fatal Compilation Errors Only	FA	F	None	O=A
Assembly Mode	AS	None	Nome	None
End Assembly Mode	EN	None	None	None

*These options may not be changed within a program unit using in-line control statement.

2. OBject listing option - If this option is "on" at compile time, the code generated by the compiler is output to the list out device along with the normal FORTRAN output. If the source statement LIsting option is "off", this output is suppressed. The object listing format is:

```
AAAAA   xxx    yyyyy    B    cccccccc
```

where:

AAAAA is the relative address of the generated instruction,
xxx (Extended Compiler Only) is the instruction mnemonic,
yyyyy (Extended Compiler Only) is the link loader code (refer to link loader documentation), and
cccccccc is the link loader load word.

3. EXtended Object Listing Option - If this option is "on" at compile time, the code generated by the compiler during the map output is output to the list out device along with the normal map output. This code includes string back definitions and locations of various scalars and constants used in the program unit. This output is in the same format as for the object listing output. This option is normally useful only to a system programmer. If the OBject listing option or the MAp ouput option is "off", this output is suppressed.

4. MAp output option - If this option is "on" at compile time, a map is output when an END statement is detected. The map output contains all scalars, arrays, statement numbers, etc., and their relative addresses. If this option is "off" at compile time, the FORTRAN map output is suppressed.

5. LOng address generation option - If this option is "on" at compile time, the code generated by the compiler correctly accesses data in common in the upper memory map (over 32K). Programs or subprograms will also correctly access internal arrays. This option may not be changed within a program unit using in-line option control statements.

6. TRiad output option - If this option is set at compile time, several tables generated by the compiler during an expression scan are output in a partially symbolic format. This option is normally useful only to a system programmer.

7. WAlk-back code generation option - If this option is "on" at compile-time, additional code is output within each program module such that a source level error message is output during any execution contingency. See Section 3-3 for a more complete discussion. This option may not be changed within a program unit using in-line option control statements.

8. ABnormal subscript redefinition option - If this option is "on" at compile time, the automatic inter statement optionization performed by the compiler is disabled, i.e., subscripts are recalcuated at each statement. This option is useful only if redefinition of subscripts during execution occurs in an abnormal fashion. See Section 5 for a more complete discussion.

9. Additional SUbscript optimization option - If this option is "on" at compile time, saved subscript calculations using variables in COMMON are not determined to be invalid after a subprogram call. This option should not be used if variables in COMMON are used as subscripts and modified during subprogram calls. See Section 5 for a more complete discussion.

10. Computed GOTO checking code generation option - If this option is "on" at compile time, additional code is generated by the compiler to check the range of the computed GOTO variable. If the value of the variable is out of range, the additional code will transfer control to the statement following the computed GOTO statement.

11. DEbug statement compilation option - If this option is "on" at compile time, the "D" in column 1 of a debug statement is replaced with a blank and the statement is processed as a normal FORTRAN statement. If this option is "off" at compile time, debug statements are treated as comments and ignored by the compiler. If the ASsembly mode option is "on", this option does not have any effect.

12. IGnore blank statements option - If this option is "on" at compile time, blank statements (columns 1-72 blank) are treated as comments and ignored by the compiler. If this option is "off" at compile time, blank statements are treated as normal FORTRAN statements.

13. SKipped statement listing option - If this option is "on" at compile time, conditional-compile control statements, skipped statements of a conditional-compile block, and skipped debug statements are listed by the compiler. If this option is "off" at compile time, these statements are not listed by the compiler. If the source statement LIsting option is "off", this option does not have any effect.

14. INdented listing option -- If this option is "on" at compile time, automatic indentation is performed by the compiler while listing source statement. See Section 2-5 for a more complete discussion on indented listing. If the source statement LIsting option is "off", this option does not have any effect.

15. Listing Option of OPtion Control statements - If this option is "on" at compile time, option control statements are listed by the compiler. If this option is "off" at compile time, option control statements are processed but not listed by the compiler. If the source statement LIsting option is "off", this option does not have any effect.

16. Source statement LIsting option - If this option is "on" at compile time, FORTRAN source statements are listed by the compiler. If this option is "off" at compile time, listing of valid statements (FORTRAN and in-line control statements) is suppressed and only errors are listed.

17. SEquence numbers on left option - If this option is "on" at compile time, the source line sequence number will be output to the left of the line, separated from the rest of the line by a colon (:). This is useful if the output list device is a teletype or a display terminal.

18. CHaracter code conversion option - If this option is "on" at compile time, the 026 characters ?@% & # are converted to "'() += on input by the compiler. This is for compatibility between various keypunch character sets.

19. FAtal compilation errors only (VULCAN only) - If this option is "on" at compile time, only fatal compilation errors result in a system error message. If this option if "off" at compile time, any compilation error results in a system error message. Fatal errors always cause a system error message. This option may not be changed within a program unit using in-line option control statements.

20. ASsembly mode option - If this option is "on" at compile time, all ensuing source statements (excluding control statements) are treated as assembly language statements. The compilation mode is returned to normal FORTRAN compilation mode upon encountering an ENd assembly mode option on an option control statement. This option may not be used in an option-off (the NO option prefix) control statement. This option does not have any option-bit assigned to it.

21. ENd assembly mode option - If this option is "on" at compile time, Assembly mode is turned "off" and normal FORTRAN compilation is carried out. This option may not be used in an option-off (the NO option prefix) control statement. This option does not have any option-bit assigned to it.

3-6 DEBUG STATEMENTS

Debug statements, as the name suggests, provide an important tool in producing diagnostic version of a program. This extension is available only in the Extended FORTRAN Compiler. A source input line having a "D" in column is defined as a debug statement. Compilation of debug statements depends on the value of the DEbug statement compilation or B15 (DMS/DOS/TOS/ROS) or "G" option. If the option is "on" at compile time, the "D" in column 1 of a debug statement is replaced with a blank and the statement is processed as a normal FORTRAN statement. If the option is "off" at compile time, debug statements are treated as comments and they are ignored by the compiler.

A program may contain any number of debug statements. Debug statements may be used as continuation lines of any statement. It must be noted that the statement number field of debug statements is limited to only four columns. As compared to in-line conditional-compile control statements, the debugging information produced by debug statements is limited to only one level. This is due to the fact that either all debug statements are compiled or all are skipped.

The following example illustrates the use of debug statements:

Example:

```
        DO     100    I=1,N
        TIME = TIME + DELTA
        FORCE = FUNC1 (TIME)
        DISPL = FUNC2 (FORCE)
D       PRINT, FORCE, DISPL
100     CONTINUE
```

The above example contains a debug statement in the range of a DO-loop. During each iteration of the DO-loop, the FORCE and DISPL are computed using functions FUNC1 and FUNC2 respectively. If the DEbug statement compilation option is "on" at compile time, the values of FORCE and DISPL are printed at the end of each iteration of the DO-loop. If the option is "off" at compile time, the values are not printed.

SECTION IV
INTERFACING FORTRAN AND ASSEMBLER GENERATED MODULES

4-1 CALLING FORTRAN GENERATED SUBPROGRAMS

The FORTRAN compiler generates entrances to all subprograms with a mechanism to pass the arguments if any, whether subroutines, functions or statement functions. Any FORTRAN subroutine expects to find the address of the calling instruction plus one in the (J) register, therefore it should be called with a BLL instruction. An argument list should follow the BLL instruction containing the addresses of the exact number of arguments that the subprogram is expecting.

For example, the FORTRAN subroutine that begins with the statement:

SUBROUTINE ALPHA (arg_1, arg_2, . . ., arg_n)

should be called by an assembly language program in the following manner:

```
L       BLL      $ALPHA
L+1     DAC      arg1              address of first argument
L+2     DAC      arg2              address of second argument
.       .
.       .
.       .
.       .
L+n     DAC      argn              address of nth argument
L+n+1   .        control is returned to this location
        .
        .
```

4-2 CALLING OF FORTRAN FUNCTIONS

In addition to the subprograms generated by both the SUBROUTINE and FUNCTION statements, FORTRAN functions (subprograms that begin with a FUNCTION statement) return a function value via the registers. The particular register depends upon the data type of the function. Table 4-1 lists the registers used for each data type.

4-2 CALLING OF FORTRAN FUNCTIONS (CONT'D.)

Table 4-1. Register Location of Function Values

	Result Register(s)**	
Data Type	SAU	non-SAU
Integer	(A)	(A)
Logical	(A)	(A)
Real	(X)	(D)
Double Precision	(X)	(D)
Complex (Real Part)	(X)	(D)
Complex (Imaginary Part)	(D)	(F$IMAG)*

*F$IMAG is a double word which resides in the FORTRAN library and is linked into FORTRAN programs whenever complex operations are specified. Hence, this pseudo-register may be accessed by an external reference to FIMAG.

**The appropriate condition register will also be set to reflect the result of the function call.

The user of FORTRAN library functions or FORTRAN compiler generated functions must take this into account when calling these functions from assembly language programs.

4-3 FORTRAN-CALLABLE ASSEMBLER LANGUAGE PROGRAMS

The FORTRAN statement: CALL ALPHA (arg_1, arg_2, . . . , arg_n) generates the assembly language equivalent of:

```
L         BLL        $ALPHA

L+1       DAC        arg1         address of 1st argument

L+2       DAC        arg2         address of 2nd argument

 .         .

 .         .

 .         .

 .         .

L+n       DAC        argn         address of nth argument
L+n+1                control must return to here.
```

The addresses in the argument list may be indirect, therefore the subprogram ALPHA should access these addresses by use of the GAP instructions.

4-3 FORTRAN-CALLABLE ASSEMBLER LANGUAGE PROGRAMS (CONT'D.)

To illustrate a method of accessing arguments supplied by a call from a FORTRAN compiler generated program, the following is the assembly language equivalent of what the compiler generates at the entry to a SUBROUTINE or FUNCTION with no multiple entry points and having at least one argument:

The statement: SUBROUTINE OR FUNCTION ALPHA ($arg_1, arg_2, \ldots, arg_n$) generates:

```
        XDEF        ALPHA, ENTRY
        RORG        0 (See Note 1)
        BLOK        n + 1
ENTRY   TNK         n
        GAP         1
        TIM         ENTRY, K     transfer argument list
        BWK         *-2
        TJM         ENTRY -(n + 1)  save return
        USER'S First Instruction
```

To illustrate a method of accessing a FORTRAN compiler generated program with no arguments, the following is the assembly language equivalent of what the compiler generates at the entry to a SUBROUTINE:

The statement: SUBROUTINE ALPHA generates:

```
        XDEF        ALPHA,ENTRY
        RORG        0 (See Note 1)
        BLOK        1
ENTRY   TJM         ENTRY -1  save return
        USER'S First Instruction
```

To illustrate a method of accessing arguments supplied by a call from a FORTRAN compiler generated program with alternate entry points to the first named identifier:

The statement: SUBROUTINE OR FUNCTION ONE (arg_1, arg_2, arg_3),
TWO (arg_4, arg_2, arg_1), THREE ($arg_1, arg_5, arg_2, arg_6$)

```
                XDEF          ONE,0
                XDEF          TWO, 1
                XDEF          THREE,2
                RORG          0 (See Note 1)
        0       BUC           ONE
        1       BUC           TWO
        2       BUC           THREE
                BLOK          n + 1 where n = number of different arguments (6)
      ONE       TNK           m     where m = number of arguments in first identifier (3)
                GAP           1
                TIM           ONE - (n-m),K  transfer argument list
                BWK           *-2
                TJM           ONE - (n + 1)  save return as exit address
                USER'S First Instruction
```

To illustrate a method of accessing arguments supplied by a call from a FORTRAN compiler generated program to an alternate entry point:

The statement SUBROUTINE or FUNCTION ONE (arg_1,arg_2,arg_3), TWO (arg_4,arg_2, arg_1), THREE (arg_1,arg_5,arg_2,arg_6)

assume the statement:

ENTRY THREE

then the following code would be generated:

```
                BUC           * + 2 (j) + 2 where j = number of arguments in the ENTRY
                                            identifier's dummy list (4)
     THREE      GAP           1
                TIM           Exit address + arg1 relative dummy number (1)
                GAP           1
                TIM           Exit address + arg5 relative dummy number (5)
                GAP           1
                TIM           Exit address + arg2 relative dummy number (2)
```

```
          GAP         1

          TIM         Exit address + arg6 relative dummy number (6)

          TJM         Exit address

          USER's First Instruction
```

To illustrate a method of accessing a FORTRAN compiler generated ENTRY point to a program with no arguments:

assume the statement SUBROUTINE ...,NOARG,...

assume the statement

ENTRY NOARG

The the following code would be generated:

```
          BUC         * + 2

NOARG     TJM         Exit address

          USER's First Instruction
```

NOTE 1: The RORG is not necessarily at zero if the subprogram contains DIMENSION or DATA statements.

From this point, any access to a dummy argument can be made safely by using an indirect reference to the point in the newly generated argument list, since the GAP instruction has eliminated all indirect levels.

The previous examples suggest methods of accessing arguments, argument addresses and alternate ENTRY points to SUBROUTINES or FUNCTION subprograms; the programmer may use any method he wishes.

It must be remembered that if the calling FORTRAN program assumes that the subprogram is a function (i.e., calls it within an expression as opposed to using a CALL statement), the result of the function must appear in the proper register when control is returned to the calling program (see Table 4-1). In addition the appropriate condition register must be set to reflect the result of the function.

SECTION V
OPTIMIZATION

5-1 GENERAL

This section describes the object code optimization automatically performed by the FORTRAN compiler.

5-2 COMPILER OPTIMIZATION

The FORTRAN compiler automatically produces optimized object code in several areas. This optimization is all local, that is, each statement is optimized independently of other statements. These optimizations take several forms.

5-2.1 Sub-Program Entries

For a sub-program with no arguments, the subprogram entry consists of a single store instruction to save the linkage address. When there are three or more arguments, a four instruction loop is set up to obtain argument addresses. If only one or two arguments are present, then an expanded form of this loop is used which takes two instruction for one argument and four instructions for two arguments. Thus with one or two arguments, no extra storage is used, and execution time is decreased. An argument passing subroutine is not used.

5-2.2 IF Statements

The program flow branching performed after an arithmetic IF is optimized. If two of the result branches to be used are the same, then the appropriate branch instruction is used to branch on either result condition.

5-2.3 Immediate Instructions

The Harris Series 6000 computers contain an extensive set of immediate or operand instructions. These instructions are automatically used by the compiler whenever possible. Their use causes a decrease in memory access time since the constant is part of the instruction and does not need to be independently fetched. A decrease in storage is also possible, since the constants do not need to be stored.

5-2.4 Subscripts

When an array variable is subscripted with a constant, the element address referenced is calculated at compile time. This enables an array references to be as efficient as a scalar reference, if the subscript is a constant. If the subscripts of an array is a variable, but contains a constant (e.g., A(I,3) or A(I+5)) then the constant is incorporated into the effective array base address. Thus the subscript calculation is made more efficient.

5-2.5 Sub-Expression Optimization

During the processing of a statement involving arithmetic computations, the FORTRAN compiler recognizes calculation sequences that are similar. When such common sub-expressions are recognized, the object code is optimized so that the sub-expression is calculated only once. When it is calculated, the value is saved in a temporary location until it is needed in the second expression. This optimization extends to all sub-expressions within the statement, including subscript calculations.

5-2.6 Logical IF Optimization

If a logical IF is a simple logical IF, that is, if it consists of a single relational operator and no .AND., .OR., .XOR., or .NOT. operators, then the normal creation of a logical value is bypassed and a conditional branch is used immediately.

For example, the following logical IF will generate the following code:

```
IF (I .LT. J) GO TO 55

    TMA     I
    SMA     J
    BON     $55
```

5-2.7 Intrinsic Functions

Certain intrinsic functions are converted to in-line instructions. The resultant code will save execution time and/or storage. In no case will either be sacrificed. The following intrinsic functions are optimized:

ABS, DABS, IABS, FLOAT, IFIX, INT, IDINT, SNGL, DBLE and REAL

5-2.8 Simple Integer Expressions

Certain simple integer expressions will make use of some specialized machine opcodes. A "TFM" or a "TZM" will be used when setting an integer to -1 or zero (or a logical variable to .TRUE. or .FALSE.).

An "AAM" will be used when the variable being assigned is also added in to the expression. As a special extension to this, "AUM" will be used if the increment is 1.

Examples:

```
FORTRAN              Generated Code

I = 0                   TZM   I

J = 3-4                 TFM   J

I = K-J+I               TMA   K
                        SMA   J
                        AAM   I
```

5-2.8 Simple Integer Expressions (Continued)

FORTRAN	Generated Code
L = L+1	AUM L
L = L - 1	TNA 1 AAM L

5-2.9 Index Registers

Whenever possible, array references are done thru index registers. This results in storage and time savings for several reasons.

An indirect reference does not need to be made to the calculated element address, thus saving the indirection time and the storage of the calculated value.

When calculating the subscript value, the array base need not be added in since it is incorporated within the indexed reference.

Different arrays accessed with the same subscript may use the same index register. This is also due to the incorporation of the array base address in the array referencing instruction.

Example:

```
IA(I)  =  IA (I)  +  IB(I) *IC(I)

          TMK     I
          TMA     IC (0), K
          MYM     IB (0), K
          AAM     IA (0), K
```

5-2.10 Inter Statement Optimization

Whenever possible, subscript calculations are <u>not</u> re-calculated from statement to statement. This occurs with subscripts in index registers as well as complete array references stored in temp locations.

At the termination of one statement and the initiation of the next, several tests are made to determine the validity of saved subscripts. Depending upon the results of these tests; none, or more, or all of the saved subscript calculations are determined invalid. Subscripts are determined to be invalid if:

1. A variable which is defined by appearing on the left side of an equal sign, is used in the subscript calculation.

2. A variable which appears in a calling sequence to a function or subroutine, is used in the subscript calculation.

5-2.10 Inter Statement Optimization (Continued)

3. They were calculated in the object of a logical IF.

All saved subscript calculations are determined to be invalid if:

1. The new statement has a statement number.

2. The previous statement was:

   a) a statement function definition
   b) an ENTRY statement
   c) a DO statement
   d) a DECODE statement
   e) a READ statement
   f) an I/O statement with an implied DO list

3. In-line assembly code has been entered or just exitted.

It is possible, however, for a subscript calculation to become invalid in spite of the above tests. If a variable used in a subscript calculation is in COMMON it may be modified in a subroutine call. A subscript variable may also be modified by being equivalenced to a variable which is modified.

In this case, the inter statement optimization may generate erroneous results. If this occurs, there are two methods to correct the problem.

1. Place a statement number on all statements in which subscripts need to be recalculated. This will cause all subscripts to be recalculated as they are needed as explained above.

2. If method 1 is not desired because of program complexity, option "A" or B12 (see Section 3-4) should be set during compilation. This will cause subscripts to be recalculated at each statement. Note: It should be carefully determined that the problem as described above is actually occurring since the use of this option may significantly increase run-time and core storage. Note: The "A" or B12 option only affects inter statement optimization, it has no effect upon the optimization as described in preceding paragraphs.

Example:

```
IA (I)  =         14
          TMK          I
          TOA          14
          TAM          IA(0), K

B(J)    =         13.5
          TMJ          J
          AJJ
          TMX          =13.5
          TXM          B(0), J
```

5-2.10 Inter Statement Optimization (Continued)

Example (Continued)

```
C (K)  =        IA(I+4)*  B(J + 1)
          TMI             K
          AII
          TMX             B(1) , J
          TMA             IA (4) , K
          MAX
          TXM             C (0) , I
```

SECTION VI
IN-LINE CODE

6-1 INTRODUCTION

6-1.1 Scope of Extension

This extension processes in-line assembly language mnemonics, symbolic labels, statement numbers and operands embedded within a FORTRAN IV main program, subroutine or function subprogram.

6-1.2 General Description

The in-line extension to the Series 6000 Extended FORTRAN Compiler is a one pass processor system. All source language statements provide a one-to-one map into machine language instructions and single or multiple data word configurations.

The in-line assembly mode of the compiler is turned "on" using an in-line option control statement containing "AS" (Assembly) as an option name. The in-line assembly mode is turned "off" using an in-line option control statement containing "EN" (End Assembly) as an option-name. Any section of in-line assembly code may contain any number of in-line control statements (See Section 3).

Source statements may contain a statement label, a mnemonic instruction or pseudo-operation, an operand field, and a comments field. Target language is Series 6000 Machine Language which is listed as a binary file whose record format is identical to the input record format of the Link Loader.

6-2 SOURCE LANGUAGE FORMAT

6-2.1 Scope

This section describes the format for source input statements to the Series 6000 In-Line Extension to the FORTRAN IV compiler.

6-2.2 Statement Number or Label Field

The statement number field may contain an asterisk (*) in column 1, a statement number, a symbolic label or may be left entirely blank.

An asterisk in column 1 causes all succeeding columns in the line to be treated as comments. No binary output is generated.

Statement numbers conform to the rules of FORTRAN IV. The statement number is assigned the address of the instruction or pseudo-operation being labeled. Note that imbedded blanks may not be present within the statement label.

A symbolic label must begin with an alphabetic character, contain not more than 6 characters and contain no delimiting characters as defined by the section "SYMBOLIC LABELS". The label is typed either implicitly or explicitly and may be used in a FORTRAN statement provided the label does not contain special characters.

<u>Examples (Valid)</u>

```
1          ***
2          RORG      2000
RATE2      DATA      3.0E6
F$GO       TOA       1
```

<u>Examples (Invalid)</u>

```
1.3        NOP                 Special Character
A+B        BLOK      4         Delimiter
1ABC       QSS       '2        Not Numeric
=WXYZ      TAM       D3(20)    First Char not Alpha
```

Note that a statement label (if present) must begin in column 1.

6-2.3 Operation Field

The operation field begins with the first non-blank column after the statement label or the first non-blank column if no statement label is present (column 1 is blank). The operation field may contain a three-character computer instruction mnemonic, or a three-or-four character pseudo-operation code.

An asterisk (*) following a three character computer instruction mnemonic is used to indicate an indirect memory reference except in the case of the computer input/output instructions, where the asterisk is used as an override or merge specification. These special cases are defined in the Macro Assembler General Specification AA61649, Table A-14.

6-2.4 Operand and Comments Fields

The operand field begins with the first non-blank column after the operation field. This field may not contain any imbedded blanks except within literal constants (e.g., " " or 3HA A). The comments field begins with the first non-blank column after the operand field. If the instruction mnemonic requires no operand, then the comments field begins immediately after the operation field.

This field may contain an operand constant, address, etc., as indicated by the particular instruction. (Reference the Macro Assembler General Specification AA61649, Tables A-4 to A-19 for a detailed listing.)

6-2.5 Sequence Field (Columns 73-80)

The sequence field may contain any identification (e.g., a card sequence number) the user desires. This field is ignored by the in-line portion of the compiler.

6-3 OPERAND FORMATS

6-3.1 Scope

This section describes the operand formats used with the in-line extension of the FORTRAN IV compiler. A list of the Series 6000 instruction mnemonics and their permissible operand formats is contained in the Appendix of the Macro Assembler General Specification AA61649 manual. Exceptions are noted as encountered in this document.

6-3.2 Current Location (*)

An asterisk (*) in the operand field indicates that the relative location of the current instruction is used as the value of the asterisk (*).

Examples (Valid)

```
BUC          *+2       SKIP THE NEXT INSTRUCTION
TOA          LABEL-*   ABSOLUTE DIFFERENCE
```

6-3.3 Symbolic Labels

A symbolic label in the operand field indicates that the address associated with the label is to be used as the operand address. A symbolic label may be declared as common, an array, data reference, statement number, dummy, variable, function results or external. Labels assume the type (REAL, INTEGER, etc.) as declared by the associated FORTRAN program. A symbolic label must begin with an alphabetic character and is delimited by a plus (+), minus (-), quote ("), blank ( ), comma (,), apostrophe ('), left parenthesis "(", or right parenthesis ")". Only the first six characters are used to determine the uniqueness of the symbolic label.

Examples (Valid)

```
TMA          RATE12          alphanumeric label
TME          F/LOAD          Note: Only in-line code allows
TLO          A.*:=           special characters in the symbolic
BUC          B$ADD           label
```

Examples (Invalid)

```
TMA          12AB            numeric
TME          A'B             delimiter
TLO          A+B             both must be defined first
BUC          AR(AY           subscript
```

6-3.3.1 Common

Labeled and blank COMMON are used as defined by the associated FORTRAN program. Should the user reorder or redefine the COMMON areas, then the generated addresses or values are subject to change.

Examples (Valid)

```
COMMON ITOP,VOLT,WATT,COMARY(2,3,4),BOTTOM
COMMON/LABEL/AMP,POWER,POLE,POST
COMMON/DIFLAB/HOT,COLD,SOUTH,EAST,WEST,NORTH
.
.
.
```

Examples (Valid) Cont'd.

```
:ASSE
      ZZZ*      WATT,K                    position 3 indirect indexed
      BSL       2+AMP                     loc 2 of block label
      TOA       BOTTOM-ITOP-1             length of blank common
*THE OFFSET OF DIFFERENT LABEL COMMONS ADJUSTED BY BLANK COMMON
      TDM       NORTH-POWER+WATT          position '13
      TAM       COMARY(3)+1               word 6 of COMARY
      TOA       HOT-COLD+3                positive 1
```

Examples (Invalid)

```
      TLO       LABEL                     new scalar variable
      TOA       HOT-COLD                  negative value
```

6-3.3.2 Array

Arrays are used as defined by the DIMENSION statement or other FORTRAN area declaration statements of the associated FORTRAN program. An array is addressable as

1) NAME without subscripts refers to word 1 of the array.

2) NAME (C) with one level of subscripting for any n dimensional array. The constant C references the first word of item C automatically adjusted by the declared array type.

3) NAME (i,j,k) where i,j and k are constants and correspond to the correct number of subscripts declared for the array NAME.

A dummy array name is not subscripted because the address of the array is not known until the execution of the related subprogram. The user should use indexing and/or indirecting when referencing dummy array values.

Examples (Valid)

```
      FUNCTION SUB (IDUM, ADUM)
      DIMENSION DRPL (12), D3 (3,3,3), ARAY (10), ADUM (100)
:ASSE
      TLO       D3                        address of first word
      TMI       D3+5                      contents of word 6
      BLOK      D3(2)-D3(1)               words per item
      BLL       D3(8)                     item 8
      BRL       D3(2,3,1)                 item 8 also
      TNA       ARAY(10)-DRPL+2           size of all dimensions
```

Examples (Invalid)

```
      TOA       D3(1)-D3(2)               negative value
      TMD       D3(1,2)                   incorrect number of subscripts
      TAM       DRPL (-1)                 neg subscript
      TME       ADUM(2)                   dummy
```

6-3.3.3 Data Reference

Constants as defined by the FORTRAN DATA statement are referenced via the symbolic identifier of the associated program. The symbolic label references the address of the first word of the respective data item regardless of the assigned type.

Examples (Valid)

```
          DATA INT, REL, RONG/2,3.0,5.0/
:ASSE
          TMA      INT                       integer 2 to A register
          TOA      REL-INT                   difference of 1 word apart
          TMX      RONG                      real 5.0 to SAU X register
          TMB      REL+1                     exponent of 3.0
```

Examples (Invalid)

```
          TOA      INT-REL                   negative value
          TME      REL(2)                    not a dimension
          TMD      REL/RONG                  no label REL/RO
```

6-3.3.4 Statement Number

The statement number is used to reference relative locations within the in-line code or to reference the address of FORTRAN statements. If a statement number appears as a label within a set of in-line code, then a path error is not issued following a decision type FORTRAN statement. Operand statement numbers are preceeded by a dollar sign "$" and may not contain embedded blanks. Statement number labels may not contain leading or embedded blanks.

Examples (Valid)

```
          BUC     $10                        go to statement 10
3 4 5     TOA     $11-$10                    words between 11 and 10
LABEL     TMA*    $345+5,I                   5 words plus loc $345
20        BLOK    10                         10 words
          .
          .
          .
10        TMD     $20+2                      word 3 of blok 10
          TND     $10-LABEL+1                size of space
          TLO     $30                        address of statement 30
          .
          .
          .
          RETURN
:ASSE
          TLO     D3
14        BUC     *+2                        no path error
:END
          I = I + 1
```

Examples (Invalid)

```
        TME      -$4                                   negative address
        TMA      $5+3                                  statement number 5 undefined
        TLO      $1AB                                  delimiter
        BUC      $1 2 3                                space terminates operand field
        RETURN
:ASSE
        TOA      5
        TMA      REL
        TME      D3 (1,2,3)-4
        BUC      INT
:END
        I = I + 1                                      path error
```

6-3.3.5 Dummy

Dummy arguments provide a means of passing information between a subprogram and the program that called it. Dummies point to the relative address which contains the address of the information desired. For details rererence the FORTRAN Compiler General Specification AA61516 "FORTRAN - Callable Assembler Language Programs" section 4-3. Dummy operands may be address adjusted but not subscripted.

Examples (Valid)

```
        SUBROUTINE TEST (A,I,J,K,B,C)
        DIMENSION B(100)
:ASSE
        TLO      A+4                                   address of address of B
        TMK      C                                     address of C in main program
        TMA*     I                                     contents of I from main prog
*SIMULATE SUBSCRIPTING
        TMK      J                                     address of J
        TMA      4,K                                   same as J(5)
        TAM      3,K                                   same as J(4) = J(5)
*NUMBER OF DUMMIES
        TOA      C-A+1                                 count of dummies
```

Examples (Invalid)

```
        TMA      A-C+1                                 negative value
        TOE      B(3)                                  subscript
        TME      -I                                    negative value
```

6-3.3.6 Variable

Variables are data whose values may vary during program execution and which are referenced with an identifier. Variables may be any of the FORTRAN data types. Symbolic labels or statement numbers may be used as variables.

Examples (Valid)

```
        TEM          ADDR              single word
        TAM          ADDR+1
        TMX          I                 double word
```

Examples (Invalid)

```
        TMA          A'4               symbol syntax
        TME          B+2               if B not defined
        TMI          C(1)              if not dimensioned
```

Note that under VULCAN, new variable names are immediately allocated. Thus if it is desired to create some special data structure within a variable, then it should be set up within the in-line code before its first reference. For example:

```
        :ASSEMB
        ABC          DATA    1,2,3,4,5
        55           TMR     ABC          is valid, but
        :ASSEMB
                     TMR     ABC
                     BUC     $56
        ABC          DATA    1,2,3,4,5    is not valid
```

6-3.3.7 Function Results

Within a function the identifier of the function subprogram is treated as though it were a scalar variable and must be assigned a value. The value returned for a function is the last one assigned to its identifier prior to the execution of a RETURN statement.

Examples (Valid)

```
        FUNCTION DIFF (A,B,C)
:ASSE
        TDM          DIFF              double word
        TLO          DIFF              address of result
        TMA          DIFF              first word of result
```

Examples (Invalid)

```
        TAM          DIFF+1            not defined
        TME          DIFF(3)           not dimensioned
```

6-3.3.8 External Requests

An external request instructs the Link Loader to search a library for the desired name and load the named module. An unconditional external request is indicated by a dollar sign ($) followed by a symbolic label. External labels may not be address adjusted.

Examples (Valid)

```
        DAC          $F$ADD            address of F$ADD
        TMA*         $A*. B,I          contents indexed
        BAC          $EXIT,2           byte external
```

Examples (Invalid)

```
        TLO     5+$A                  address adjusted
        TIM     $B+3                  delimiter
        TMA     $A-$B                 delimiter
        DAC     $$ABC                 not true symbolic
```

6-3.4 Absolute Constants

Absolute constants in the operand field indicate that the integer constant (octal or decimal) is to be used as the operand address. The resultant value must be positive.

Examples (Valid)

```
        TMA     5                     absolute location 5
        TME     '10                   absolute location '10
```

Examples (Invalid)

```
        TAM     -5                    neg address
        TMI     3.2                   not integer
        DAC     5-6                   neg address
        TME     2B1                   not numeric
        TMA     12345678962573418     too big
        TME     '18                   not all octal
```

6-3.5 Address Arithmetic

Any combination of the current location (*), symbolic labels, statement numbers, or absolute constants may be joined by the plus (+) or minus (-) operations to define and address. Symbolic labels and statement numbers must have been defined before address adjustment is allowed with the desired items. Such a combination is referred to as an "operand expression".

The mode of an operand expression is relative if the number of plus and minus signs preceding the relative labels (current location "*" included) are not equal. Otherwise, the mode of the operand expression is absolute.

Examples (Valid)

```
A       TMA     A+5                   relative
B       TOA     B-A                   absolute
C       TMA     D3(1,2,3)+6           relative
        TMA     B-A + C               relative
```

Examples (Invalid)

```
A       TMA     -A-5                  negative
B       TOA     A-B                   negative
        TOA     D3(1) - D3(2)         negative
        TME     A-100                 if negative
```

6-3.6 Indexed Address Reference

An operand address may be appended with an index reference by placing a comma (,) in the column to the immediate right of the operand address and following the comma with the index specification (I, J, K). For the details of the hardware indexing scheme refer to Series 6000 REFERENCE MANUAL DC 6024 COMPUTER SYSTEM.

Examples (Valid)

```
TMA     0,K                 true address in K
TME     5,I                 I + 5 locations
TMI     D3(1,2,3),J         array indexed
```

Examples (Invalid)

```
TMA     A,H                 H not an index
TME     B,-J                Syntax
TMI     C,1                 1 not an index
```

6-3.7 Text

Alphanumeric text may be used as an operand by the use of leading and following quotation marks (".. "). The text is right-justified in the instruction operand and any unused portion of the operand is zero filled. Text operands may not be used with address arithmetic. Up to two characters of text may be entered. The first text character may be any valid character. The second optional text character may be any character except a quote symbol.

Examples (Valid)

```
TOA     "AB"                AB
COB     """"                quote symbol
TOE     " A"                blank A
TOA     "","                quote comma
```

Examples (Invalid)

```
TOA     "ABC"               too many characters
TOA     "A""                delimiter
TOE     "B                  delimiter
COB     "C" + 5             Syntax
```

6-3.8 Literal

A literal is indicated by an equal sign (=) as the first character of the operand field, followed by a data or logical constant (reference DATA Section 2-3) to be assigned a nonprogram location and permits the address of that assignment to be used as the operand address.

The expression mode of a literal is relative.

If the machine representation of a constant is more than one word, the assigned address of the first word is used as the operand address. Any identical constants of the same type of either FORTRAN or in-line origin which require at least one full computer word are assigned the same location.

Any memory reference instruction using a literal may not be appended with an indirect, index or address adjustment.

<u>Examples (Valid)</u>

TMD	=12.5D-10	double word
TLO	=(3.0, 5.0)	complex 4 words
TMA	=1	one word
TME	=.TRUE.	logical

<u>Examples (Invalid)</u>

TAM	=5	to memory
TMA	=3 + 6	address adjusted
TLO	=LABEL	symbol
TMA*	=2	indirect
TME	=3,K	indexed

6-3.9 FORTRAN External

Symbols defined by a FORTRAN EXTERNAL statement are permitted as in-line operands as outlined in this document under the topic of "Symbolic Labels - External Request".

6-3.10 Memory Referencing Boolean Instructions (Bit - Processor)

Some of the bit-processor instructions require two operand fields separated by a comma. The second operand field must be numeric and comply with the requirements of each operation code. The operand requirements of the field sizes modes are listed in the Appendix of the Macro Assembler General Specification AA61649.

<u>Examples (Valid)</u>

DMH	1,1	and memory with H
QBM	19,2	query bit of memory

<u>Examples (Invalid)</u>

QBM	LABEL	relative

6-4 PSEUDO-OPERATIONS

6-4.1 Scope

This section describes the pseudo-operations that are processed by the in-line extention of FORTRAN IV extended compiler.

6-4.2 RORG (Relative Origin)

The RORG pseudo-operation sets the next instruction's program location to the contents of the absolute or relative expression in the operand field. If a symbolic label or a statement number is present, it is assigned the value of the operand expression. If symbolic labels or statement numbers are present in the operand expression, they must have been previously defined. If an operand error is encountered a no operation instruction replaces the RORG pseudo-operation.

Examples (Valid)

RORG	5	relative five
RORG	ARRAY (1,1,1)	
RORG	LABEL	
RORG	*-5	if not negative
RORG	ARRAY(101)-ARRAY(1)	rel 200_{10}

Examples (Invalid)

RORG	-5	negative
RORG	40000	too big
RORG	$EXTRN	external

6-4.3 BLOK (Reserve Memory)

This pseudo-operation reserves a block of storage (n locations). (n = the value of the absolute expression in the operand field.) If a symbolic label or statement number is present in the tag field, it is assigned the first location in the block. If symbolic labels or statement numbers are present in the operation expression, they must have been previously defined. If an operand error is encountered, the BLOK pseudo-operation code is replaced with a no operation instruction.

Examples (Valid)

BLOK	10	octal 12 words
BLOK	$11 - $10	if defined
BLOK	ARRAY 2 - ARRAY 1	size * type

Examples (Invalid)

BLOK	-10	negative
BLOK	LABEL	relative
BLOK	$SIN	external
BLOK	40000	too big

6-4.4 DATA

The DATA pseudo-operation operand field may contain any number and combination of the data constants defined in the following paragraphs.

If more than one item is present (items are separated by commas), they are assigned sequential memory locations. If a symbolic label or statement number is present, it is assigned the location of the first data item. There is no relationship between the FORTRAN typing of a symbolic label assigned to a DATA pseudo-operation and the operand field supplied by the user. In-line constants are coded in the same format as FORTRAN constants. Identical constants yield identical internal representation.

6-4.4.1 Single Integer

A single integer data constant consists of an optional sign (+ or -) and 1-7 decimal digits. Single integer constants are contained in one computer word. The magnitude of an integer constant must not exceed 8,388,607.

Examples (Valid)

```
INT     DATA     +428        typed integer
REL     DATA     1           the label REL is typed real
```

6-4.4.2 Single-Precision Real

A single-precision real data constant consists of an optional sign (+ or -), 1 or more decimal digits of which only 6+ most significant digits are retained, a decimal point, and followed by an optional decimal exponent. The exponent follows the numeric value and consists of the letter E followed by a signed or unsigned integer that represents the power of ten by which the numeric value is to be multiplied. The magnitude of the range of real numbers is about $2.94X10^{-39}$ to $1.7X10^{38}$. A single-precision real constant is contained in two computer words.

Examples (Valid)

```
INT     DATA     3.14        the label INT is typed integer
REL     DATA     17E5        typed real
        DATA     -3.2E-10    not labeled
```

6-4.4.3 Double-Precision Real

A double-precision real data constant consists of an optional sign (+ or -), 1 or more decimal digits of which only the 11+ most significant digits are retained, a decimal point, and followed by an optional decimal exponent. The exponent follows the numeric value and consists of the letter D followed by a signed or unsigned integer that represents the power of ten by which the numeric value is to be multiplied. The magnitude of the range of real numbers is about $2.94X10^{-39}$ to $1.7X10^{38}$. A double-precision real constant is contained in two computer words.

Examples (Valid)

```
INT     DATA     3.14DØ      the label is typed integer
REL     DATA     17D5        typed real
*THE FOLLOWING EXAMPLE CONSISTS OF SIX WORDS UNLABELED
        DATA     -3.2D-10, 3.7D5, 4.6D-6
```

6-4.4.4 Octal Constants

An octal constant is designated by an apostrophe (') followed by 1 to 8 octal digits (0-7). Octal constants which are less than 8 digits are right justified in the 24-bit computer word with leading zeros.

Examples (Valid)

```
INT       DATA      '77           typed integer
REL       DATA      '12345670     typed real
*THE FOLLOWING EXAMPLE CONSISTS OF FOUR WORDS UNLABELED
          DATA      '1, '3, '47, '256
```

Examples (Invalid)

```
          DATA      '-3           negative
          DATA      '384          the digit 8 is not octal
          DATA      '123456712345 too many digits
```

6-4.4.5 Logical Constants

Logical constants may assume either of two forms:

.TRUE. which is equivalent to a DATA -1

or

.FALSE. which is equivalent to a DATA Ø

These forms have the logical values "true" and "false", respectively.

Examples (Valid)

```
          DATA      .TRUE.            -1
          DATA      .FALSE.           0
          DATA      .TRUE., .FALSE.   -1,0
```

6-4.4.6 Complex

Complex constants are expressed as an ordered pair of constants enclosed in parenthesis in the form (a, b), where a and b are signed or unsigned real constants. The complex constant (a, b) is interpreted as meaning a+bi, where i is equal to the square root of -1. A complex constant is contained in four computer words.

Examples (Valid)

```
     COMPLEX REL
:ASSE
INT            DATA      (1.34,52.01)        typed integer
REL            DATA      (98344.,0.34E+02)   typed complex
*THE FOLLOWING EXAMPLE CONSISTS OF 12 WORDS UNLABELED
               DATA      (1.0,2.0),(-1.0,-1000.),(3E7,4E-6)
:END
```

Neither part of a complex constant may exceed the value limits established for real data.

6-4.4.7 Hollerith

A Hollerith constant takes the form:

nHs or 's' or "s"

where:

s is a Hollerith string. Note that blanks are significant in Hollerith strings.

n is an unsigned integer specifying the number of characters in the string. The maximum number of characters per statement is 55.

The data are packed as three 8-bit ASCII characters per word, left justified and blank filled. Each DATA Hollerith string begins with a new word. See Section 2-3, Literal Constants for a more complete description.

6-4.4.8 Truncated Text

Truncated text may be used as a data constant. Truncated text is packed 4 characters per word, left-justified, blank filled. A truncated text constant may generate one or more words depending on the text string length. The format of a truncated text constant is:

T "string"

Examples (Valid)

```
DATA      T"A"
DATA      T"THIS IS A LONG TEXT STRING"
```

6-4.4.9 Binary

A binary data constant is designated by one or more bit specifications of the form: Bn, where "n" is the unitary bit position (0 thru 23) that is to be set on.

Examples (Valid)

```
DATA      B0
DATA      B0B1B2, B23B22B21
```

6-4.5 DAC

The DAC pseudo-operation is considered as a 16-bit memory reference instruction, however, any indirect or index specifications are placed in bit positions 21 through 23 to conform to the indirect memory reference format as defined in the Series 6000 Computer System Reference Manual.

Examples (Valid)

```
        INT         DAC         $10         the address of statement number 10
        REL         DAC         TABLE,I
                    DAC         *           the address of this word
                                ARRAY(1,2,3)
                    DAC         $ANGLE      external address
```

6-4.6 BAC

Refer to the Macro Assembler General Specification AA61649 Section 4-25, Pseudo-Operations BAC.

6-4.7 ***

This pseudo-operation reserves one word of storage, which is set to zero.

6-4.8 ZZZ

The pseudo-operation is considered as a 15-bit memory reference instruction containing an operation code of '00, which may be appended with indirect and index references.

Examples (Valid)

```
        ZZZ         TABLE
        ZZZ*        TABLE+5,I
```

6-4.9 Octal Operation Code

An octal operation code is indicated by using an apostrophe (') as the first character of the opcode and following it by a two-digit octal constant. An octal operation code is processed as 15-bit memory reference instruction and must include an operand. The operation code may be appended with indirect or indexed references.

Examples (Valid)

```
        '24         ALPHA+10
        '24*        BETA,I
```

Examples (Invalid)

```
        '38         ALPHA       8 is not an octal digit
        '4          BETA        operation is not two digits
        '123        BETA+1      operation is more than two digits
```

6-4.10 REEN (VULCAN compiler only)

The REEN pseudo-operation is used if in-line code is used, but the re-entrancy of the FORTRAN program has not been violated. A REEN load code is normally output by the compiler if no in-line code is present, or if a REEN pseudo-op is encountered someplace within "in-line" code. For a further discussion of "REEN", see the MACRO Assembler General Specification AA61649.

6-4.11 PORG (VULCAN compiler only)

The PORG pseudo-operation is used to switch the location counter mode to the PORG or data area mode. This mode is used for instructions or data which may be modified by the program during execution. The format of the pseudo-op is the same as for the RORG pseudo-op. To return to the instruction location counter mode, the RORG pseudo-op is used. For a further discussion of "PORT", see the MACRO Assembler General Specification.

6-4.12 PDATA (VULCAN compiler only)

The PDATA pseudo-operation is used to temporarily switch the location counter mode to the PORG mode. If the current location mode is already PORG, then this pseudo-op is identical to the DATA pseudo-operation. Otherwise, the PDATA pseudo-op is equivalent to the following sequence of operations:

```
PORT     *
DATA     . . .
RORG     *
```

6-4.13 PBLOK (VULCAN compiler only)

The PBLOK pseudo-operation is used to temporarily switch the location counter mode to the PORG mode. If the current location mode is already PORG, then this pseudo-op is identical to the BLOK pseudo-operation. Otherwise, the PBLOK pseudo-op is equivalent to the following sequence of operations:

```
PORG     *
BLOK     . . .
RORG     *
```

6-4.14 RDAT

The RDAT pseudo-operation is used to generate a string of data constants a specified number of times. The format of this pseudo-op is:

RDAT $\quad X (C_1, C_2, C_3 . . . C_n)$

where X is a positive integer constant indicating how many times the following constant list is to be repeated. Each C_i is a constant which may be of any form that is allowed with the DATA pseudo-operation.

<u>Examples (Valid)</u>

```
RDAT     5(0)
RDAT     10(.TRUE.,'A',B0B5)
```

6-4.15 PRDAT (VULCAN compiler only)

The PRDAT pseudo-operation is used to temporarily switch the location counter mode to the PORG mode. If the current location mode is already PORG, then this pseudo-op is identical to the RDAT pseudo-operation. Otherwise, the PRDAT pseudo-op is equivalent to the following sequence of operations.

```
        PORG    *
        RDAT    . . .
        RORG    *
```

6-5 COMPILE-TIME AND RUN-TIME OPTIONS

6-5.1 Scope

This section describes the in-line facility for listing control, messages and diagnostics.

6-5.2 List Control

See Section III, FORTRAN Diagnostic Messages - Compile-Time and Run-Time Options.

Exception to option B5 set.

At compile time, the code generated by the in-line extension is output to the list out device along with the in-line statements.

The in-line generated output format is:

```
AAAAA     B     cccccccc
```

where AAAAA is the relative address of the generated operation or pseudo-operation code, B is the link loader code (refer to Link Loader documentation), and cccccccc is the link loader load word.

6-5.3 Compile-Time Options

See Section III, FORTRAN Diagnostic Messages - Compile-Time and Run-Time Options.

NOTE

External requests are not converted to the double precision mode when OPTION B4 is set. However, DATA constants are converted to the double precision format.

<u>Examples</u>

```
        BLL     $SIN          not converted to DSIN
        BLL     COS           new scalar name COS
        DATA    3.60E0        converted to double type
```

6-5.4 Run-Time Options

Reference the desired operating system.

6-5.5 Error Codes and Messages

All diagnostics associated with the rules of in-line coding produce warning messages. Even though the message may begin with the word "ERROR" an abort code is not issued to the loader. The stringing of the load address of undefined identifiers is maintained where possible by the use of a relative no operation instruction and the undefined identifier as the operand.

The following supplemental diagnostics apply to specific in-line code violations. Other diagnostics may appear within a set of in-line coding due to system violations or being superseded by a FORTRAN IV violation.

Error Number	Message	Cause
1	INVALID OPERATOR	a) Invalid computer instruction mnemonic or pseudo-operation code. b) Octal operation code has incorrect number of digits, should be two. c) Indirecting or Indexing a literal constant.
2	INVALID CONSTANT	An arithmetic address adjustment is not an integer.
3	INVALID SYNTAX	a) Invalid mode of operation expression. b) Incorrect number of operand fields. c) Identifier construction error. d) Literal constant as a variable. e) Negative resultant operand. f) No operand or indirecting or indexing expected. g) Magnitude of Operand to large. h) Text as relative expression. i) Text construction error.
4	MISSING OPERAND	a) Operand missing or not beginning in column 15. b) Index not I, J, or K.
6	INVALID STATE-MENT NUMBER	a) Duplicate statement numbers. b) Multiply defined identifiers.
10	SUBSCRIPT USAGE	a) Incorrect number of subscripts. b) Not positive integer constants.
11	INVALID STATEMENT	Incorrect ordering of ":" control statements.
12	PARENTHESIS	a) Literal constant not complex. b) Incorrect subscript construction. c) Data constant not complex.
13	MIXED MODES	a) Address adjusted undefined identifier. b) Address adjusted external request. c) Multiple undefined identifiers in an expression.

Error Number	Message	Cause
14	INVALID DELIMITER	a) Consecutive delimiter. b) Undefined identifier, literal constant, external, text, or undefined statement number not followed by a blank or comma. c) Embedded blank in an operand expression. d) Operand exceeded column 72 and incomplete at column 72. e) Dummy array subscripted. f) Missing comma in two operand expression.
17	INVALID ITEM USAGE	a) An identifier is used where a constant is expected. b) FORTRAN EXTERNAL identifier as a variable. c) Byte external second operand greater than two.
30	STATEMENT ORDER	FORTRAN specification statement following an in-line statement.
31	NO PATH TO HERE	No label or statement number following (not necessarily immediately) a FORTRAN logical path for execution. Reference this document Miscellaneous - Path diagnostic.

6-5.6 Run-Time Diagnostics

Reference the desired operating system.

6-6 MISCELLANEOUS

6-6.1 Scope

This section describes several items which are unique to the operation of the in-line coding.

6-6.2 DO Termination

FORTRAN DO statements may terminate within an in-line section. However, extra code is generated after the processing of the in-line statement. If the in-line statement is a pseudo-operation instruction such as a BLOK, then the user must provide for a logical path of execution. Neither the contents of the condition register nor the A Register are preserved.

Example (Valid)

```
                DO 1 I=1,10
:ASSE
                TMI    I                   1 to index I
1               AME    IARAY(1)-1,I        sum contents of IARAY to E
                TEM    IANSWR              answer
:END
```

Example (with caution)

```
              DO 2 I=1,10,2
:ASSE
              TMK     I                 I to index K
              AMD     RATE(50)-2,K      sum five values of rate (X)
*NOTE USE OF A REGISTER
*NOTE DISCONTINUOUS INSTRUCTIONS IN MEMORY
2             BLOK    20                skip 20 memory locations
*THE USER MUST PROVIDE FOR A LOGICAL PATH
*AROUND THE BLOK 20 TO THE DO LOGIC.
```

6-6.3 Automatic Symbol Assignment

Any symbolic label encountered within the operand expression is declared as having been properly used by the user and is assigned storage according to its FORTRAN type. These labels do not appear in the memory map as undefined variables.

Example (Valid)

```
:ASSE
*DEFINE AND USE TEMPORARY STORAGE
              TDM     T$SAVE            unique to in-line
              .                         not necessary to
              .                         allocate by data or
              .                         any other method.
              TMD     T$SAVE
:END
```

Example (Invalid)

```
:ASSE
*DEFINE AND USE INCORRECTLY
              TDM     T$SAVE            correct
              .
              .
              .
              TMB     T$SAVE+1          undefined address adjusted
*THE ABOVE STATEMENT IS IN ERROR "DELIMITER"
:END
```

6-6.4 Second Operand Field

The second operand field must be either an index (I, J, or K) or a positive integer. Reference the individual instruction for the specific format.

6-6.5 Forward Reference Symbols

Symbolic labels which have not been assigned a memory location are termed as forward reference symbols. Statements which assign memory locations are the DIMENSION, COMMON, EQUIVALENCE, DATA, FUNCTION dummies, and SUBROUTINE dummies. Statement labels which define a memory location are statement numbers in columns 1-5 of both FORTRAN and in-line and symbolic labels in columns 1-6 of an in-line statement. Since the relative address of a forward referenced label is not known at compile time the label may not have address arithmetic.

Example

```
            DATA A/1.0/
10          B = C
:ASSE
            TMD      A        valid
            TMB      A+1      valid
            TMD      B        valid
            TMB      B+1      invalid
            BUC      $10+2    valid
            BUC      $11+3    invalid
:END
            STOP
            END$
```

6-6.6 Path Diagnostic

The FORTRAN diagnostic message "NOTE 31 xxxxxx NO PATH TO HERE" is issued after the first FORTRAN statement following a set of in-line code which contains no statement labels and are not in the logical path for execution by the previous FORTRAN statement.

Example (Valid)

```
            GO TO 10
:ASSE
            TMA      I                  statement label may
            TNK      5                  appear anywhere within
11          TAM      J(6),K             an in-line set of code
            BWK      *-1
:END
```

Example (Invalid)

```
            GO TO 10
:ASSE
            TMD      D                  this section of code
            TDM      E                  is not in a FORTRAN logical
:END                                    path for execution
```

6-6.7 Statement Function Dummies

Statement subprogram dummies are self contained and are therefore not addressable as dummy variables.

SECTION VII
HARRIS STRUCTURED - FORTRAN LANGUAGE

7-1 GENERAL

The Harris Structured - FORTRAN language is a superset of the Harris FORTRAN language. It adds structured programming capabilities to FORTRAN language.

7-2 STRUCTURED PROGRAMS

7-2.1 Scope

This section describes various terms used in this chapter in describing the structure and execution of structured programs.

7-2.2 Structured - FORTRAN Compiler

The Structured - FORTRAN Compiler processes the Harris Structured - FORTRAN language. This structured programming extension can be incorporated in any Harris FORTRAN compiler by setting the FLAG 6 "on" during the assembly of the compiler.

7-2.3 Block-Statements, Boundary-Statements and Exit-Statements

The Harris Structured - FORTRAN language provides additional statements for constructing blocks of a structured program with minimum or no use of the GOTO statements and statement numbers. These additional statements are called block-statements. Block statements are classified into five groups:

1. Block-IF, WHILE, block-DO, FOR, and LOOP statements.

2. END IF, END WHILE, UNTIL, END FOR, and END LOOP statements.

*3. OR IF and ELSE statements.

**4. EXIT IF, EXIT WHILE, EXIT DO, EXIT FOR, and EXIT LOOP statements.

**5. EXIT IF IF, EXIT WHILE IF, EXIT DO IF, EXIT FOR IF, and EXIT LOOP IF statements.

Block-statements of groups 1 through 3 are called boundary-statements. Block-statements of groups 4 and 5 are called exit-statements.

* The keyword ELSE IF may be used in place of the keyword OR IF.

** The keyword ESCAPE maybe used in place of the keyword EXIT.

A program unit may contain any number of block-statements, intermixed with other FORTRAN statements, satisfying the structural and nesting restrictions. Like any other FORTRAN statement, a block-statement may be assigned a statement label. All block-statements are executable statements. During execution of a program, control may be transferred to any block-statement.

7-2.4 Blocks of a Program Unit

A program unit may be viewed as consisting of one or more blocks of statements, placed one after another and/or one within another. Blocks are one of two general classes:

1. A FORTRAN-block or F-block consists of one or more statements, none of which is a block-statement.

2. A Structural-block or S-block consists of two or more boundary-statements enclosing other statements.

7-2.5 Range(s) of a S-block

A range of a S-block consists of the executable statements following a boundary-statement of the S-block up to but not including, the next boundary-statement of the same S-block.

A S-block having N boundary-statements has (N-1) ranges. If a S-block S_1 appears within a range R of another S-block S_2, the entire S-block S_1 must be within the range R of the outer S-block S_2. A range of a S-block may be empty. A S-block may contain any number of S-blocks within its range(s).

7-2.6 Initial-Statements

The first statement of a S-block is always a boundary-statement. A boundary-statement which is valid as the first statement of a S-block is called an initial-statement. An initial-statement opens a S-block and identifies the block-type of the S-block opened by it. The Structured - FORTRAN language permits five different initial-statements and correspondingly there are five block-types available. Boundary-statements of group 1 described earlier in this section, represent the available initial-statements.

7-2.7 Terminal-Statements

The last statement of a S-block is always a boundary-statement. A Boundary-statement which is valid as the last statement of a S-block is called a terminal-statement. A terminal-statement closes a S-block. Associated with each block-type there is a different terminal-statement. A terminal-statement may be used to close a S-block of associated block-type only. Boundary-statements of group 2 described earlier in this section represent the available terminal-statements.

7-2.8 Alternative-Initial-Statements

A boundary-statement which is neither an initial-statement nor a terminal-statement is called an alternative-initial-statement. Associated with each block-type there are zero or more alternative-initial statements. An alternative-initial-statement may be used as a boundary-statement for a S-block of associated block-type only. A S-block may have any number of associated alternative-initial-statements as its intermediate boundary-statements. Boundary-statements of group 3 described earlier in this section represent the available alternative-initial-statements.

7-2.9 Exit-Statements

An exit-statement provides means for exiting a S-block or terminating execution of a S-block. That is, it provides means for transferring control to the statement following the terminal-statement of the S-block. An exit-statement always specifies the block-type of the S-block to be exited. Associated with each block-type there are two exit-statements:

1. An unconditional-exit-statement, which provides unconditional exit from the S-block. Block-statement of group 4 described earlier in this section represent available unconditional-exit-statements.

2. A condition-exit-statement, which specifies a condition for exit. Block-statements of group 5 described earlier in this section represent one available conditional-exit-statements.

An exit-statement, unconditional or conditional, must be within a range of a S-block of specified block type. A S-block may contain any number of associated exit-statements. When an exit-statement occurs in a range of more than one S-block, only the innermost S-block of specified block-type is exited.

7-2.10 S-Blocks and Associated Block Statements

Table 7-1 lists available block-types with associated block-statements.

Table 7-1. S-Blocks and Associated Block-Statements

Block-type	Initial-Statement	Terminal Statement	Alternative-Initial-Statements	Exit Statement
IF	Block-IF	END IF	OR IF, ELSE	EXIT IF EXIT IF IF
WHILE	WHILE	END WHILE	None	EXIT WHILE EXIT WHILE IF
DO-UNTIL	Block-DO	UNTIL	None	EXIT DO EXIT DO IF
FOR	FOR	END FOR	None	EXIT FOR EXIT FOR IF
LOOP	LOOP	END LOOP	None	EXIT LOOP EXIT LOOP IF

Two or more S-blocks may not use the same statement as their boundary-statement. This is an important difference between S-blocks and FORTRAN DO-loops, where nested DO-loops may use the same terminal-statement.

Table 7-2 illustrates various terms defined in this section. It shows a portion of a structured program consisting of two nested S-blocks: a FOR-block enclosing an IF-block. The type and block-level of each statement in the program is indicated on the right of the statement.

The IF-block constitutes the range of the FOR-block. The IF-block has two ranges. The first range consists of the CALL SUB1(DONE) statement and the conditional-exit-statement. The second range consists of the CALL SUB2 statement and the unconditional-exit-statement. Note that each of the exit-statements specifies a FOR-block for exit and is in the range of the FOR-block.

Table 7-2. A Structured Program

Program	Statement-Type	Block-Level
DIMENSION IDATA(10)	FORTRAN-Statement	0
LOGICAL DONE	FORTRAN-Statement	0
.	.	.
.	.	.
FOR I=1, 10	Initial-Statement	1
IF(IDATA(I). GT. 0)	Initital-Statement	2
CALL SUB 1 (DONE)	FORTRAN-Statement	2
EXIT FIR IF (DONE)	Conditional-Exit-Statement	2
ELSE	Alternative-Initial-Statement	2
CALL SUB2	FORTRAN-Statement	2
EXIT FOR	Unconditional-Exit-Statement	2
END IF	Terminal-Statement	2
END FOR	Terminal-Statement	1

7-2.11 Block-Level

The block-level of a statement S is defined as the difference between the number of initial-statements from the beginning of the program unit up to and including S, and the number of terminal-statements in the program unit preceding S.

For a correctly structured and nested program unit, the following restrictions must be observed:

1. The block-level of every statement must be non-negative.
2. The block-level of the END statement must be zero.
3. The block-level of each block-statement must be positive.
4. A block-statement must not be used as the terminal-statement of a FORTRAN DO-loop or as the object of a FORTRAN logical IF statement.
5. An alternative-inititial-statement S_1 and the nearest preceding boundary-statement S_2, having the same block-level as S_1 must be valid boundary-statements for the same block-type. The statements, if any enclosed between S_1 and S_2 constitute a range of the S-block.
6. An exit-statement must be within a range of a S-block of specified block-type.
7. The block-level of a FORTRAN DO statement and its terminal-statement must be the same.

It must be clear from the definitions of S-block and block-level, that all boundary-statements of a S-block must have the same block-level.

7-3 BOUNDARY-STATEMENTS

7-3.1 Scope

This section describes the form and execution of boundary-statements associated with block-types available in the Structured-FORTRAN Compiler.

7-3.2 IF-Block

An IF-block has the general structure:

```
        BLOCK-IF Statement                (Initial-Statement)
        .
        .                                 (First-range of the IF-block)
        .
        OR IF Statement                   (Alternative-initial-statement)
        .
        .                                 (Alternative-range of the IF-block)
        .
```

ELSE Statement	(Alternative-initial-statement)
.	(Last-range of the IF-block)
.	
.	
END IF Statement	(Terminal-statement)

The IF-block provides means for executing one range out of its one or more ranges. The range to be executed is selected by making alternative tests in the specified order. The OR IF and ELSE statements are optional in an IF-block.

7-3.2.1 Block-IF Statement

The block-IF statement has the form:

```
IF (e)
```

or

```
IF (e)  THEN
```

where e is a logical expression.

Execution of a block-IF statement causes evaluation of the logical expression e. If the value of e is true, normal execution sequence continues and the execution of the first range of the IF-block begins. If the value of e is false, control is transferred to the next OR IF, ELSE, or END IF statement that has the same block-level; that is, control is transferred to next boundary-statement of the IF-block.

Examples:

```
IF (MORE) THEN
IF (T. LT. TMAX)
IF (I. AND. J  . NE. 0)  THEN
```

7-3.2.2 OR IF Statement

The OR IF statement has the form:

```
OR IF (e)
```

or

```
OR IF (e)  THEN
```

where e is a logical expression.

Execution of an OR IF statement causes evaluation of the logical expression e. If the value of e is true, normal execution sequence continues and execution of the following alternative range of the IF-block begins. If the value of e is false, control is transferred to the next OR IF, ELSE, or END IF statement that has the same block-level. That is, control is transferred to the next boundary-statement of the IF-block.

Examples:

```
OR IF (I. EQ. 2)
ELSE IF (T. GT. TMAX)THEN
OR IF (NEXT)
```

7-3.2.3 ELSE Statement

ELSE

Execution of an ELSE statement has no effect. Normal execution sequence continues and the execution of the last range of the IF-block begins.

7-3.2.4 Termination of Execution of a Range of an IF-block

If execution of the last statement of a range of an IF-block does not result in a transfer of control, control is transferred to next END IF statement having the same block-level as the first statement following the range. That is, control is transferred to the terminal-statement of the IF-block.

7-3.2.5 END IF Statement

The END IF statement has the form:

END IF

Executions of an END IF statement does not have any effect. Normal execution sequence continues.

7-3.2.6 Order of OR IF and ELSE Statements

An IF-block may contain any number of OR IF statements and/or only one ELSE statement having the same block-level as the initial-statement of the IF-block. After an ELSE statement, an END IF statement having the same block-level must appear before the appearance of next OR IF statement having the same block level.

IF-block Example:

```
   Structure-FORTRAN                          FORTRAN

IF(ICOM. EQ. 'ADD. ) THEN                  IF(ICOM. NE. 'ADD') GOTO2
   OP1 = OP1+OP2                              OP1=OP1+OP2
OR IF (ICOM. EQ. 'SUB')                    GO TO 4
   OP1 = OP1-OP2                     2     IF (ICOM. NE. 'SUB') GO TO 3
ELSE                                       OP1 = OP1-OP2
   CALL ERROR                              GO TO 4
END IF                               3     CALL ERROR
                                     4     CONTINUE
```

In the above example, statements on the left illustrate a use of the IF-block and statements on the right represent its equivalent in the standard FORTRAN. The example computes the sum ('ADD' command) or difference ('SUB' command) of two operands OP1 and OP2. If the command ICOM is neither an 'ADD' command nor a 'SUB' command, an error condition is detected and the subroutine ERROR is called.

7-3.3 WHILE-BLOCK

The WHILE-block has the general structure:

```
WHILE Statement              (Initial-Statement)
.
.                            (Range of the WHILE-block)
.
END WHILE Statement          (Terminal-Statement)
```

The WHILE-block provides means for executing its range repetitively as long as (While) the specified condition is true. The test for the condition is made before each execution of the WHILE-block range. Thus, it is possible that the range of a WHILE-block may not be executed at all.

7-3.3.1 WHILE Statement

The WHILE statement has the form:

```
WHILE (e)
```

where e is a logical expression.

Execution of a WHILE statement causes evaluation of the logical expression e. If the value of e is true, normal execution sequence continues and execution of the range of the WHILE-block begins. If the value of e is false, control is transferred to the statement immediately following the END WHILE statement having the same block-level as the WHILE statement. That is, control is transferred to the statement following the WHILE-block.

Examples:

```
WHILE (ICOUNT. NE. 0)
WHILE (. NOT. EOF)
WHILE (T. LE. TMAX)
```

7-3.3.2 END WHILE Statement

The END WHILE statement has the form:

```
END WHILE
```

Execution of an END WHILE statement results in the transfer of control to the nearest preceding WHILE statement that has the same block-level. That is, control is transferred to the initial-statement of the WHILE-block.

WHILE-block Example:

```
     Structured-FORTRAN                      FORTRAN

DIMENSION A(100),LINK(100)           DIMENSION A(100),LINK (100)
.                                    .
.                                    .
SUM = 0.                             SUM = 0.
NEXT = LSTART                        NEXT = LSTART
WHILE(NEXT.GT.0)                  1  IF (NEXT.LE.0) GO TO 2
   SUM=SUM+A(NEXT)                   SUM=SUM+A(NEXT)
   NEXT=LINK(NEXT)                   NEXT=LINK(NEXT)
END WHILE                         2  GO TO 1
                                     CONTINUE
```

In above example, statements on the left illustrate a use of the WHILE-block and statements on the right represent its equivalent in the standard FORTRAN. The example computes the SUM of elements of a linked-list A. Array LINK is used to store the link to the next element in the linked-list. A zero or negative value for the link indicates the end of the linked-list. Note that a test for the end of the linked-list is made before each iteration of the WHILE-block.

7-3.4 DO-UNTIL-BLOCK

The DO-UNTIL-block has the structure:

```
Block-DO Statement                   (Initial-statement)
.
.                                    (Range of the DO-UNTIL-block)
.
UNTIL Statement                      (Terminal-Statement)
```

The DO-UNTIL-block provides means for executing its range repetitively until the specified condition is true. The test for the condition is made after each execution of the DO-UNTIL-block range. Thus, the range of a DO-UNTIL-block will always be executed at least once.

7-3.4.1 Block-DO Statement

The Block-Do statement has the form:

DO

Execution of a block-DO statement does not have any effect. Normal execution sequence continues and execution of the range of the DO-UNTIL-block begins.

7-3.4.2 UNTIL Statement

The UNTIL statement has the form:

```
UNTIL (e)
```

where e is a logical expression.

Execution of an UNTIL statement causes evaluation of the logical expression e. If the value of e is true, normal execution sequence continues and the DO-UNTIL-block is exited. If the value of e is false, control is transferred to the nearest preceding block-DO statement that has the same block-level. That is control is transferred to the initial-statement of the DO-UNTIL-block.

Examples:

```
UNTIL (ICOUNT.EQ.0)
UNTIL (EOF)
UNTIL (T.GT.TMAS)
```

DO-UNTIL-block Example:

```
Structured-FORTRAN                    FORTRAN

DATA IBLANK/3H   /                    DATA IBLANK/3H   /
.                                     .
.                                     .
DO                                  1 CALL GETCHR(ICHAR)
   CALL GETCHR (ICHAR)                IF (ICHAR.EQ.IBLANK) GO TO 1
UNTIL(CHAR.NE.IBLANK)
```

In the above example, statements on the left illustrate a use of the DO-UNTIL-block and statements on right represent its equivalent in the standard FORTRAN. The example skips over leading blanks and gets the next non-blank character from an input buffer. During each iteration of the DO-UNTIL-block, the next character from the buffer is obtained in ICHAR by calling the subroutine GETCHR. Next, ICHAR is checked for a blank. If ICHAR is a blank, the DO-UNTIL block is executed one more time, i.e., the blank is ignored. If ICHAR is not a blank, the DO-UNTIL-block is exited, i.e., execution continues with the statement following the UNTIL statement.

7-3.5 FOR-Block

The FOR-block has the structure:

FOR Statement	(Initial Statement)
•	
•	(Range of the FOR-block)
•	
END FOR Statement	(Terminal Statement)

The FOR-block is functionally equivalent to the FORTRAN DO-loop.

7-3.5.1 FOR Statement

The FOR Statement has the form:

FOR $v=i_1,i_2,i_3$

or

FOR $v=i_1,i_2$

where v is a scalar variable of integer type, i_1, i_2 and i_3 are scalar variable or constants of integer type. Integers i_1 and i_2 must appear. If i_3 is not present, it is assumed to have the value 1.

The FOR statement is similar to the DO-statement of a DO-loop. The DO statement specifies a statement label to identify the terminal statement of the DO-loop; however, the FOR statement does not specify any statement label. The terminal-statement of a FOR-block is always the next END FOR statement that has the same block-level.

Execution of a FOR statement activates the FOR-block and defines the FOR-variable v with the value of the initial parameter i_1. Next, loop-control processing begins.

Examples:

```
FOR I=1, 10

FOR IVAR = I, J, K

FOR  J = I, K, -1
```

7-3.5.2 LOOP-Control Processing of a FOR-block

Loop-control processing determines if further execution of the range of the FOR-block is required. If the incremental value i_3 is an integer scalar variable or positive integer constant, the variable v is compared to the terminal value i_2. If v is less than or equal to i_2, normal execution sequence continues and execution of the range of the FOR-block begins. If v is greater than i_2, control is transferred to the statement following the terminal-statement of the FOR-block and the FOR-block becomes inactive.

If the incremental value i_3 is a negative constant, the variable v is compared with the terminal value i_2. If v is greater than or equal to i_2, normal execution sequence continues. If v is less than i_2, control is transferred to the statement following the terminal-statement of the FOR-block and the FOR-block becomes inactive.

7-3.5.3 END FOR Statement

The END FOR statement has the form

```
END FOR
```

Execution of an END FOR statement results in incrementing the value of v by the value of incremental value i_3. Next, execution continues with the loop-control processing of the FOR-block.

The value of the variable v may not be modified within the range of the FOR-block.

7-3.5.4 Transfer Into the Range of a FOR-block

The control may be transferred into the range of or to the terminal statement of an active FOR-block only.

FOR-block example:

```
 Structures-FORTRAN                        FORTRAN

DIMENSION A(10, 10)                    DIMENSION  A(10,10)
.                                      .
.                                      .
FOR I=1,9                              DO  1  I=1,9
  I1=I+1                               I1 = I+1
  FOR J=I1, 10                         DO  1  J=I1, 10
    TEMP = A(I,J)                      TEMP = A(I,J)
    A(I,J) = A(J,I)                    A (I,J) = A(J,I)
    A(J,I) = TEMP                      A(J,I) = TEMP
  END FOR                         1    CONTINUE
END FOR
```

In above example, statements on the left illustrate a use of the FOR-block and statements on the right represent its equivalent in the standard FORTRAN. The example computes the transpose of a 10X10 by matrix A. Note that two separate END FOR statements are required to close two FOR-blocks: the first END FOR statement closes the inner FOR-block with J as its loop-variable and the second END FOR statement closes the outer FOR-block with I as its loop-variable.

7-3.6 LOOP-BLOCK

The LOOP-block has the structure:

LOOP Statement	(Initial-Statement)
.	
.	(Range of the LOOP-block)
.	
END LOOP Statement	(Terminal-Statement)

There are two types of LOOP-blocks:

1. A Finite-LOOP-block specifies the number of times its range is to be executed.
2. An Infinite-LOOP-block does not specify the number of times its range is to be executed. The range of an infinite LOOP-block is repetitively executed until the LOOP-block is exited.

7-3.6.1 LOOP Statement

The LOOP statement has the form:

```
LOOP (i)

or

LOOP
```

where i is a scalar variable or constant of integer type. It is called the iteration count specification.

The first form of the LOOP statement identifies a Finite-LOOP-block; the second form identifies an Infinite-LOOP-block.

Execution of the LOOP statement of a Finite-LOOP-block initializes the iteration count for the LOOP-block to the value of the iteration count specification. Next, loop-control or processing begins.

Execution of the LOOP statement of Infinite-LOOP-block does not have any effect. Normal execution sequence continues and execution of the range of the LOOP-block begins.

Examples:

```
LOOP

LOOP (100)

LOOP (I)
```

7-3.6.2 Loop-Control Processing of a Finite-LOOP-block

Loop-Control processing determines if further execution of the range of the Finite-LOOP-block is required. The iteration count is tested. If the iteration count is positive, execution of the statement immediately following the LOOP statement begins. If the iteration count is not positive, the LOOP-block becomes inactive and control is transferred to the statement immediately following the next END LOOP statement that has the same block-level. That is, control is transferred to the statement following the Finite-LOOP-block.

7-3.6.3 END LOOP Statement

The END LOOP statement has the form:

```
END LOOP
```

Execution of an END LOOP statement of a Finite-LOOP-block results in the decrementing of its iteration count by one. Next, execution continues with the transfer of control to the loop-control processing of the same loop-block.

Execution of an END LOOP statement of an Infinite-LOOP-block results in the transfer of control to the statement immediately following the nearest preceding LOOP statement. That is, control is transferred to the initial-statement of the Infinite-LOOP-block.

7-3.6.4 Iteration Count Specification of a Finite-LOOP-block

The iteration count specification i of a Finite-LOOP-block is used to establish the iteration count of the LOOP-block. If i is an integer scalar variable, a change in its value does not affect the remaining iteration count of the associated active Finite-LOOP-block. Similarly, the decrementing of the iteration count during execution of the END LOOP statement does not affect the value of the associated i.

7-3.6.5 Transfer into the Range of a LOOP-block

Control may be transferred into the range of or to the terminal statement of an active Finite-LOOP-block only. There are no restrictions on the transfer of control to any executable statement in an Infinite-LOOP-block.

Finite-LOOP-block Example:

```
 Structured-FORTRAN                    FORTRAN

NONES = 0                            NONES = 0
LOOP (24)                            DO 1 I=1,24
   LSBIT= IVAL. AND. 1               LSBIT = IVAL. AND. 1
   NONES = NONES+LSBIT               NONES = NONES+LSBIT
   IVAL= IVAL. SHIFT. -1             IVAL = IVAL. SHIFT. -1
END LOOP                        1    CONTINUE
```

In above example, statements on the left illustrate a use of the Finite-LOOP-block and statements on the right represent its equivalent in the Harris standard FORTRAN. The example computes the number of 1's in the binary representation of a number IVAL. The LOOP-block is iterated 24 (the word-length in a series 6000 machine) times. During each iteration of the LOOP-block, the following sequence takes place:

1. The 1s count NONES is incremented by an amount equal to the least significant bit (0 or 1) of the current representation of IVAL.

2. The number IVAL is shifted to the right by one bit position.

7-4 EXIT STATEMENTS

7-4.1 Scope

This section describes the form and execution of exit-statements.

7-4.2 Unconditional-exit-Statements

An unconditional-exit-statement has the form:

EXIT type

where type is one of five block types:

IF, WHILE, DO, FOR, or LOOP.

Execution of an unconditional-exit-statement results in the transfer of control to the statement following the next terminal-statement of specified block-type. That is, the innermost S-block of specified block-type enclosing the unconditional-exit-statement is unconditionally exited.

A S-block may contain any number of associated unconditional-exit-statements. Each unconditional-exit-statement must be in a range of a S-block of specified block-type.

Examples:

```
EXIT IF

EXIT FOR

EXIT DO
```

7-4.3 Conditional-exit-Statement

The conditional-exit-statement has the form:

EXIT type IF (e)

where: type is one of five block types: IF, WHILE, DO, FOR, or LOOP, e is a logical expression.

Execution of a conditional-exit-statement causes evaluation of the logical expression e. If the value of e is true, control is transferred to the statement following the next terminal-statement of specified block-type. That is, the innermost S-block of specified block-type is conditionally exited. If the value of e is false, normal execution sequence continues.

An S-block may contain any number of associated conditional-exit-statements. Each conditional-exit-statement must be in a range of a S-block of specified block-type.

Examples:

```
EXIT WHILE IF (I. LE. 0)

EXIT FOR IF (T. GT. TMAX)

EXIT LOOP IF (DONE)
```

Exit-Statement Example:

```
 Structured-FORTRAN                      FORTRAN

LOOP
   READ,NUM                      1    READ, NUM
   EXIT LOOP IF (NUM. LE. 0)          IF(NUM. LE. 0) GO TO 3
   CALL SEARCH(NUM,MODE)              CALL SEARCH(NUM,MODE)
   IF (MODE. LE. 0)                   IF(MODE. GT. 0) GO TO 2
     CALL ERROR                       CALL ERROR
     EXIT LOOP                        GO TO 3
   ELSE                          2    CALL PROCES
     CALL PROCES                      GO TO 1
   END IF                        3    CONTINUE
END LOOP
```

In above example, statements on the left illustrate a use of exit-statements and statements on the right represent its equivalent in the standard FORTRAN. The example contains two S-blocks: a LOOP block enclosing an IF-block. The LOOP-block contains two exit-statements in its range: a conditional-exit-statement and an unconditional-exit-statement. The unconditional-exit-statement also occurs in a range of the inner IF-block. During each iteration of the LOOP-block, a number NUM is read from the input device. A non-positive value for NUM signals the end of input data and the LOOP-block is exited. This is achieved using a conditional-exit-statement. If NUM is positive, the subroutine SEARCH is called to locate NUM in the data-base. The result of the search is returned in MODE. A non-positive value for NUM indicates that NUM is absent in the data-base. This causes an error condition and the subroutine ERROR is called to process the error. After return from ERROR, the LOOP-block is exited using an unconditional-exit-statement. If NUM is present in the data-base, the subroutines PROCES is called to process the data base. Next, the LOOP-block is iterated for a new value of NUM.

7-5 INDENTED LISTING

The Structured-FORTRAN Compiler provides a compile-time option to produce indented listing of a structured program. If the INdented listing or B18 (DMS/DOS/TOS/ROS) or "P" (VULCAN) option is "on" at compile time, source statements are automatically indented in the source listing produced by the compiler. The indentation of a source statement is defined as the number of columns by which columns 7-72 of the statement are shifted to the right in the source listing. The indentation is determined as follows:

1. The indentation of a comment line, an in-line control statement, an in-line assembly statement, or a skipped statement is zero.

2. The indentation of a boundary-statement is proportional to its block-level minus 1, i.e.,

   Indentation = 3* (block-level -1)

   Columns 73-80 of such a statement are omitted in the indented listing.

3. The indentation of an exit-statement or a normal FORTRAN statement is proportional to its block-level, i.e.,

   Indentation = 3* block-level

   Columns 73-80 of such a statement are omitted in the indented listing.

4. The indentation of a continuation line is same as the indentation of its initial line. Columns 73-80 of a continuation line are omitted in the indented listing.

If the INdented listing option is "on" at compile time, the following conditions prevail:

1. The source line sequence number is placed on the left of the line regardless of the value of the SEquence numbers on the left or B21 (DMS/DOS/TOS/ROS) or "N" (VULCAN) option.

2. The continuation character (column 6) of a continuation line is replaced with a plus sign (+).

3. A dot (.) is placed as an indentation-indicator in every third column, starting from the first column, of the indentation space of an indented line.

4. The maximum indentation of any source line is limited to 39 columns. A source line having greater indentation is listed with an indentation of 39 columns. It must be noted that, however, there is no upper limit on the nesting depth of S-blocks.

5. Blank statements are processed as normal FORTRAN statements regardless of the value of the IGnore blank statements or B16 (DMS/DOS/TOS/ROS) or "B" (VULCAN) option.

6. Comment lines may not be inserted between an initial line and its first continuation line or between two continuation lines. That is, comment lines are treated as statement separators.

7. If the DEbug statement compilation or B15 (DMS/DOS/TOS/ROS) or "D" (VULCAN) option is "off" at compile time, debug statements are treated as statement separators.

8. If the OBject listing or B5 (DMS/DOS/TOS/ROS) or "O" (VULCAN) option is "on" at compile time, the listing of the code generated for a statement will appear before the listing of the statement.

Listings of a program with the INdented listing option "on" and "off" are shown in Figure 7-1.

7-6 COMPILE-TIME DIAGNOSTICS

The Structured-FORTRAN compiler performs an extensive error checking of structured programs. In addition to normal diagnostic messages issued by the compiler, the following two compile-time diagnostic messages apply only to programs using structured programming extensions:

1. The FORTRAN Diagnostic Message

   ERROR 40 xxxxxx STRUCTURALLY INVALID STATEMENT

   is issued by the compiler whenever a block-statement violates any of the nesting restrictions discussed earlier in the section. A structurally invalid statement produces a fatal error condition; however, it does not affect the structure of the program.

2. The FORTRAN Diagnostic Message

   ERROR 41 bbbbb: BLOCK OPENED AT LINE nnn IS NOT CLOSED

   is issued by the compiler whenever a S-block is not closed before an END statement is encountered for the program unit. In the message:

bbbbb	specifies the type of the S-block, i.e., IF, WHILE, DO, FOR, or LOOP, and
nnn	specifies the source line number for the initial-statement of the S-block.

   A separate diagnostic message is issued for every S-block, starting from the outermost S-block and moving down to the innermost S-block, remaining open at the end.

```
C                                                                   MATMUL
C      AN EXAMPLE OF "INDENTED LISTING" OPTION                      MATMUL
C                                                                   MATMUL
       SUBROUTINE MATMUL(A,B,C,M,N,P)
       INTEGER P
       DIMENSION A(M,N),B(N,P),C(M,P)
       FOR I=1,M
       .  FOR K=1,P
       .  .  SUM=0.
       .  .  FOR J=1,N
       .  .  .  SUM=SUM+A(I,J)*B(J,K)
       .  .  END FOR
       .  .  C(I,K)=SUM
       .  END FOR
       END FOR
       RETURN
       END
```

LISTING WITH "INDENTED LISTING" OPTION "ON"

```
C                                                                   MATMUL         1
C      AN EXAMPLE OF "INDENTED LISTING" OPTION                      MATMUL         2
C                                                                   MATMUL         3
       SUBROUTINE MATMUL(A,B,C,M,N,P)                               MATMUL         4
       INTEGER P                                                    MATMUL         5
       DIMENSION A(M,N),B(N,P),C(M,P)                               MATMUL         6
       FOR I=1,M                                                    MATMUL         7
       FOR K=1,P                                                    MATMUL         8
       SUM=0.                                                       MATMUL         9
       FOR J=1,N                                                    MATMUL        10
       SUM=SUM+A(I,J)*B(J,K)                                        MATMUL        11
       END FOR                                                      MATMUL        12
       C(I,K)=SUM                                                   MATMUL        13
       END FOR                                                      MATMUL        14
       END FOR                                                      MATMUL        15
       RETURN                                                       MATMUL        16
       END                                                          MATMUL        17
```

LISTING WITH "INDENTED LISTING" OPTION "OFF"

Figure 7-1. Indented Listing