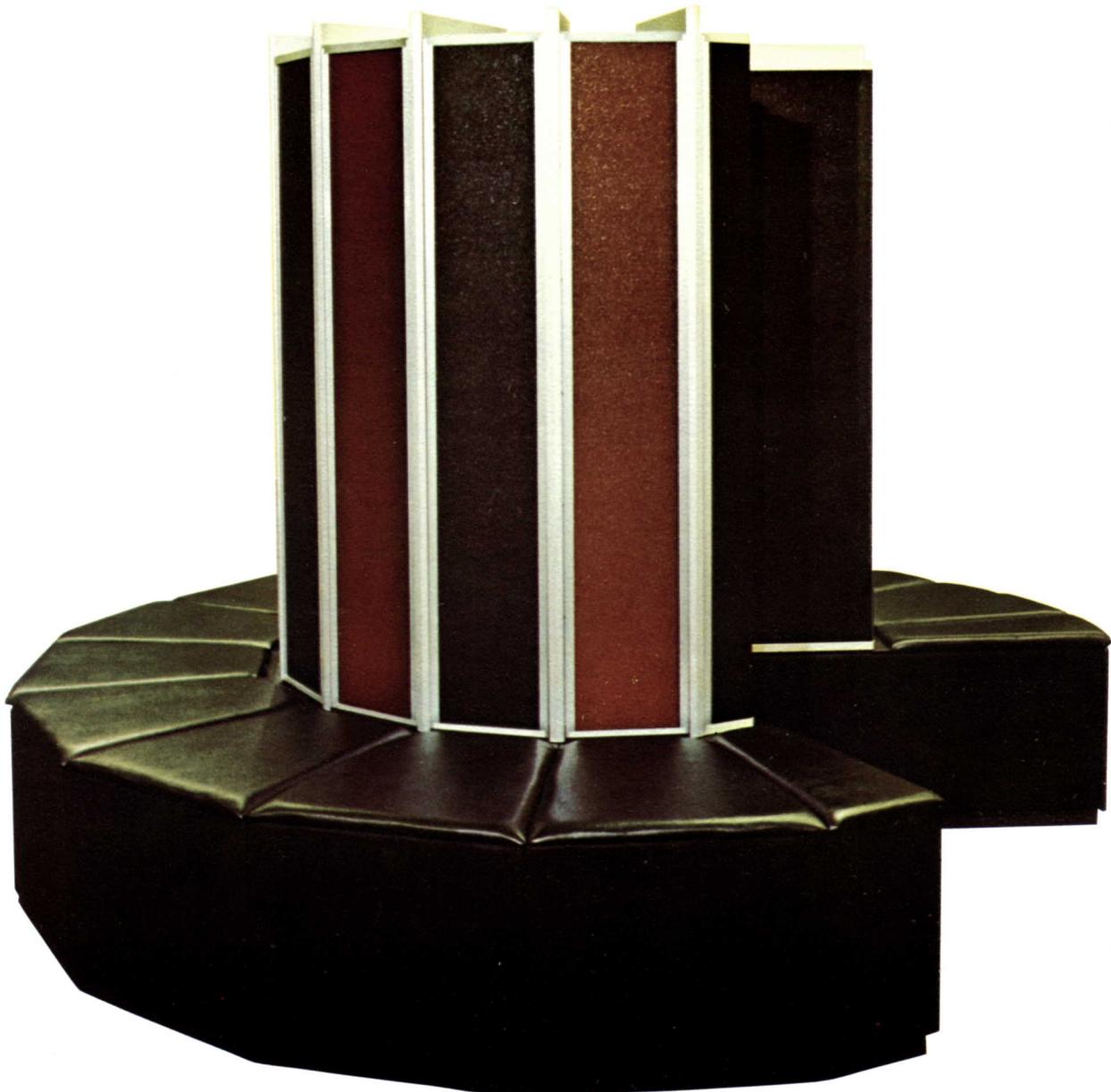


CRAY
RESEARCH, INC.

CRAY-1®
COMPUTER SYSTEM

**CRAY-1 FORTRAN (CFT)
REFERENCE MANUAL**

2240009



CRAY-1 FORTRAN LANGUAGE STATEMENT INDEX

EXECUTABLE

ASSIGNMENT

- Arithmetic assignment 3- 70
- Logical assignment 3- 71
- ASSIGN (statement label) 3- 71
- Formatted data assignment statements
 - ENCODE 3-115
 - DECODE 3-115

CONTROL

- GO TO statements
 - Unconditional GO TO 3- 72
 - Assigned GO TO 3- 74
 - Computed GO TO 3- 73
- IF statements
 - Arithmetic IF statements
 - Three-branch arithmetic IF . . . 3- 75
 - Two-branch arithmetic IF . . . 3- 75
 - Logical IF statements
 - Direct logical IF 3- 76
 - Indirect logical IF 3- 77
- Recursion statement
 - DO 3- 77
- Program unit control statements
 - CALL 3-126
 - RETURN 3-129
 - PAUSE 3- 82
 - STOP 3- 81
 - END 3- 82

INPUT/OUTPUT

- Direct input/output statements
 - Formatted READ 3- 85
 - Unformatted READ 3- 85
 - Formatted WRITE 3- 85
 - Unformatted WRITE 3- 85
 - PRINT 3- 85
 - PUNCH 3- 85
- Buffered input/output statements
 - BUFFER IN 3-112
 - BUFFER OUT 3-112
- Media control statements
 - ENDFILE 3- 91
 - BACKSPACE 3- 91
 - REWIND 3- 91

OTHER

- CONTINUE 3- 81

NON-EXECUTABLE

PROGRAM UNIT SPECIFICATION

- PROGRAM 3-116
- FUNCTION statements
 - FUNCTION 3-123
 - INTEGER FUNCTION 3-123
 - REAL FUNCTION 3-123
 - DOUBLE PRECISION FUNCTION . . 3-123
 - DOUBLE FUNCTION 3-123
 - COMPLEX FUNCTION 3-123
 - LOGICAL FUNCTION 3-123
- SUBROUTINE 3-126
- ENTRY 3-128
- EXTERNAL 3- 67
- BLOCK DATA 3-135

DATA SPECIFICATION

- Declaration statements
 - PARAMETER 3- 66
 - DIMENSION 3- 58
- Typing statements
 - IMPLICIT 3- 65
 - INTEGER 3- 64
 - REAL 3- 64
 - DOUBLE PRECISION 3- 64
 - DOUBLE 3- 64
 - COMPLEX 3- 64
 - LOGICAL 3- 64
- Association statements
 - EQUIVALENCE 3- 58
 - COMMON 3- 60
- Initialization statement
 - DATA 3- 68

OTHER SPECIFICATION

- FORMAT 3- 92
- Statement function definition . . 3-120

CRAY-1®
COMPUTER SYSTEM

**CRAY-1 FORTRAN (CFT)
REFERENCE MANUAL**

2240009

Copyright© 1976, 1977, 1978 by CRAY RESEARCH, INC.
This manual or parts thereof may not be reproduced in any
form without permission of CRAY RESEARCH, INC.





RECORD OF REVISION

PUBLICATION NUMBER 2240009

<u>Revision</u>	<u>Print Date</u>	<u>Description</u>
	July, 1976	Preliminary distribution, Xerox copies.
A	February, 1977	First printing. Since changes are very extensive, they are not noted by change bars.
B	November, 1977	Second printing. Since this represents a complete rewrite, changes are not noted by change bars.
C	April, 1978	Updates the manual to be in full agreement with the April, 1978 release of the CRAY-1 FORTRAN Compiler (CFT) Version 1.01.

Each time this manual is revised and reprinted, all changes issued against the previous version in the form of change packets are incorporated into the new version and the new version is assigned an alphabetic level. Between reprints, changes may be issued against the current version in the form of change packets. Each change packet is assigned a numeric designator starting with 01 for each new revision level. Every page changed by a reprint or by a change packet has the revision level and change packet number in the lower right-hand corner. All changes are noted by a change bar along the margin of the page.

Requests for copies of CRAY RESEARCH, INC. publications should be directed to: CRAY RESEARCH, INC.
7850 Metro Parkway
Suite 213
Bloomington, MN 55420

PREFACE

The CRAY-1 FORTRAN Compiler (CFT) transforms CRAY-1 FORTRAN Language statements into the highly efficient computer programs required for effective use of the CRAY-1 Computer System. This manual describes the CRAY-1 FORTRAN Language in its entirety, the use of the CRAY-1 FORTRAN Compiler, and related CRAY-1 Operating System characteristics.

This publication results from a high degree of cooperation and dedication on the part of many at Cray Research. Notable among these are Richard Nelson, Head FORTRAN Programmer, and Irene Mallgrave and Richard Hendrickson of FORTRAN Compiler Development. Editorial and graphics support were provided by Janet Robidoux, Head Publications Editor, and Chad Jewett of Technical Communications. The manuscript was typed by Mary Huber and Arlene LaBounty.

Cray Research is especially grateful for the invaluable general and technical critiques of this material by:

- Mostyn Lewis of the European Centre for Medium Range Weather Forecasts at Bracknell, Berkshire, England;
- Robert Cave of Princeton, New Jersey; and
- Russell Rew of the National Center for Atmospheric Research, Boulder, Colorado.

The final measure of this manual's worth lies in its usefulness to you, the reader. Cray Research invites your comments and criticisms as essential to the continued improvement of this publication.

Neill T. Ward
November, 1977

CONTENTS

CONTENTS

<u>PREFACE</u>	iii
1. <u>INTRODUCTION</u>	1-1
1.1 GENERAL	1-1
1.2 THE CRAY-1 COMPUTER SYSTEM	1-1
1.3 THE CRAY-1 FORTRAN LANGUAGE	1-1
1.4 CRAY-1 FORTRAN PROGRAMMING	1-1
1.5 THE CRAY-1 FORTRAN COMPILER (CFT)	1-2
1.6 THE CRAY-1 OPERATING SYSTEM (COS)	1-2
2. <u>THE CRAY-1 COMPUTER SYSTEM</u>	2-1
2.1 GENERAL	2-1
2.2 SYSTEM OVERVIEW	2-1
2.3 THE CENTRAL PROCESSING UNIT	2-3
3. <u>THE CRAY-1 FORTRAN LANGUAGE</u>	3-1
3.1 GENERAL DESCRIPTION	3-1
3.1.1 The CRAY-1 FORTRAN compiler	3-1
3.1.2 Conformance to the ANSI Standard	3-1
3.1.3 Syntax description	3-2
3.2 FUNDAMENTALS	3-3
3.2.1 Notation	3-3
Character sets	3-3
Sequences	3-3
Syntactic items	3-4
Lists and list items	3-6
3.2.2 Statements	3-7
3.2.3 The FORTRAN coding form	3-7
FORTRAN coding form format	3-7
Lines	3-9
3.2.4 The executable program	3-10
Program units	3-10
Procedures	3-12
3.2.5 Data representations	3-14
Data types	3-14
Data specification	3-14
Variables, arrays, array elements, and functions	3-14
Definition	3-15
Reference	3-15
Storage	3-15
Association	3-15

3.2.6	Order of statements and lines	3-16
3.2.7	Normal execution sequence	3-17
3.3	DATA	3-19
3.3.1	Types of data	3-19
3.3.2	Data identifiers	3-19
3.3.3	Constants	3-20
	Integer constants	3-21
	Real constants	3-21
	Double-precision constants	3-23
	Complex constants	3-24
	Logical constants	3-25
	Boolean constants	3-25
	Hollerith constants	3-26
3.4	ARRAYS	3-28
3.4.1	Array declarators	3-28
	Form of an array declarator	3-28
	Kinds of array declarators	3-28
3.4.2	Properties of an array	3-29
	Data type of an array and an array element	3-29
	Dimensionality of an array	3-29
	Size of a dimension	3-29
	Size of an array	3-29
	Array element order	3-30
	Array storage sequence	3-30
3.4.3	Array element names	3-31
	Subscript expression	3-31
	Subscript values	3-31
3.4.4	Dummy and actual arrays	3-31
3.4.5	Adjustable arrays and adjustable dimensions . . .	3-34
3.4.6	Use of array names	3-34
3.5	EXPRESSIONS	3-35
3.5.1	Arithmetic expressions	3-35
	Arithmetic Operators	3-36
	Form and interpretation of arithmetic expressions	3-36
	Types and interpretation of arithmetic expressions	3-41
	Integer quotients	3-43
	Type conversion	3-43
3.5.2	Hollerith expressions	3-44
3.5.3	Relational expressions	3-45
	Relational operators	3-45
	Arithmetic relational expressions	3-45
	Hollerith relational expressions	3-46
	Values of logical factors, terms, disjuncts, and expressions	3-49

3.5.4	Logical expressions	3-47
	Logical operators	3-47
	Form and interpretation of logical expressions	3-48
3.5.5	Boolean expressions	3-51
3.5.6	Precedence of all operators	3-51
3.5.7	Summary of rules of interpretation	3-52
3.5.8	Evaluation of expressions	3-52
	Order of evaluation of functions.	3-53
	Parentheses and expressions	3-54
	Evaluation of arithmetic expressions.	3-54
	Evaluation of Hollerith expressions	3-55
	Evaluation of relational expressions.	3-55
	Evaluation of logical expressions	3-55
	Evaluation of Boolean expressions	3-55
3.6	STATEMENTS	3-56
3.6.1	Statement classification.	3-56
3.6.2	Data specification	3-56
	DIMENSION statements	3-58
	EQUIVALENCE statements	3-58
	COMMON statements	3-60
	Type statements	3-64
	IMPLICIT statements	3-65
	PARAMETER statements.	3-66
	EXTERNAL statements	3-67
3.6.3	Data initialization	3-68
	Form of a data statement	3-68
	DATA statement restrictions	3-69
3.6.4	Data assignment	3-70
	Arithmetic assignment statements.	3-70
	Logical assignment statements	3-71
	ASSIGN statements	3-71
3.6.5	Program control	3-72
	Unconditional GO TO statements	3-72
	Computed GO TO statements	3-73
	Assigned GO TO statements	3-74
	Three-branch arithmetic IF statements	3-75
	Two-branch arithmetic IF statements	3-75
	Direct logical IF statements.	3-76
	Indirect logical IF statements.	3-77
	DO statements	3-77
	CONTINUE statements	3-81
	STOP statements	3-81
	PAUSE statements	3-82
	END statements	3-82
3.6.6	Input/output operations	3-82
	Input/output records, files, units, and formats	3-83
	READ, WRITE, PRINT, and PUNCH statements	3-85

Execution of data transfer input/output statements	3-88
BACKSPACE, ENDFILE, andREWIND statements	3-91
Restrictions on input/output statements	3-91
3.6.7 Format specification	3-92
FORMAT statements	3-92
Form of a format specification	3-92
Edit descriptors	3-93
Interaction between input/output lists and format specifications	3-95
Positioning by format control	3-96
Editing	3-97
3.6.8 Buffered input/output operations	3-111
The BUFFER IN and BUFFER OUT statements	3-112
The UNIT function	3-113
The LENGTH function	3-114
3.6.9 Formatted data assignment	3-115
ENCODE and DECODE statements	3-115
The ENCODE statement	3-116
The DECODE statement	3-116
3.6.10 The main program	3-116
The PROGRAM statement	3-116
Main program restrictions	3-117
3.6.11 Functions and subroutines	3-117
Categories of functions and subroutines	3-117
Referencing functions	3-118
Intrinsic functions	3-119
Statement functions	3-120
External functions	3-123
Subroutines	3-126
ENTRY statements	3-128
RETURN statements	3-129
Arguments and common blocks	3-130
Common blocks	3-134
3.6.12 Utility procedures	3-135
3.6.13 Block data subprograms	3-135
Block data statement form	3-135
Block data subprogram restrictions	3-135
3.7 ASSOCIATION AND DEFINITION	3-137
3.7.1 Storage and association	3-137
Storage sequences	3-137
Association of storage sequences	3-137
Association of entities	3-137
3.7.2 Defining entities	3-138
3.7.3 Undefining entities	3-139

3.8	SCOPE AND CLASSES OF SYMBOLIC NAMES	3-140
3.8.1	Scope of symbolic names	3-140
	Global entities	3-140
	Local entities	3-141
3.8.2	Classes of symbolic names	3-141
	Common blocks	3-141
	External functions	3-142
	Subroutines	3-142
	The main program	3-142
	Block data subprograms	3-142
	Arrays	3-143
	Variables	3-143
	Constants	3-143
	Statement functions	3-144
	Intrinsic functions	3-144
4.	<u>CRAY-1 FORTRAN PROGRAMMING</u>	4-1
4.1	GENERAL	4-1
4.2	FORTRAN PROGRAMMING FOR CRAY-1 VECTOR OPERATIONS	4-1
4.3	FORTRAN PROGRAMMING FOR CRAY-1 MEMORY OPERATIONS	4-9
5.	<u>THE CRAY-1 FORTRAN COMPILER</u>	5-1
5.1	GENERAL	5-1
5.2	INPUT TO CFT	5-1
5.3	OUTPUT FROM CFT	5-1
5.3.1	Binary output	5-2
5.3.2	Listable output	5-2
	Source statement listings	5-2
	Symbol table and common block name list	5-2
	Statement label list	5-4
	Warning messages	5-4
	Fatal error messages	5-5
5.4	COMPILER DIRECTIVES	5-5
5.4.1	Compiler directive lines	5-6
5.4.2	Listable output control directives	5-6
	EJECT directives	5-6
	LIST directives	5-7
	NOLIST directives	5-7
	CODE directives	5-7
	NOCODE directives	5-7
5.4.3	Vectorization control directives	5-7
	VECTOR directives	5-7
	NOVECTOR directives	5-7
	IVDEP directives	5-8
5.4.4	Integer control directive	5-8

6.	<u>THE CRAY-1 OPERATING SYSTEM</u>	6-1
6.1	GENERAL	6-1
6.2	JOB PREPARATION	6-1
6.3	REQUIRED JOB CONTROL STATEMENTS	6-2
6.3.1	Job identification	6-2
6.3.2	Assign dataset characteristics	6-3
6.3.3	CRAY-1 FORTRAN compiler execution.	6-3
6.3.4	Set or clear sense switch.	6-6
6.3.5	Relocatable loader	6-6
6.3.6	The job control statement file	6-7
6.4	ERROR MESSAGES DURING PROGRAM EXECUTION	6-7

APPENDIXES

A	CHARACTER SET	A-1
B	CRAY-1 FORTRAN INTRINSIC FUNCTIONS	B-1
C	CRAY-1 FORTRAN UTILITY PROCEDURES	C-1
D	CFT ERROR MESSAGES	D-1
E	COS ERROR MESSAGES	E-1
F	CREATING NON-FORTRAN PROCEDURES	F-1
G	CRAY-1 CENTRAL PROCESSING UNIT	G-1
H	RANDOM INPUT/OUTPUT OPERATIONS	H-1

FIGURES

2-1	The CRAY-1 Computer System.	2-2
2-2	The CRAY-1 central processing unit.	2-3
3-1	FORTRAN coding form	3-8
3-2	The executable program	3-11
G-1	The computation section	G-4
G-2	Numeric data representation	G-10

TABLES

3-1	Special characters	3-4
3-2	Required order of lines and statements.	3-17
3-3	Subscript values	3-32
3-4	Arithmetic operators.	3-36
3-5	Interpretation of operators in expressions.	3-36
3-6	Precedence of arithmetic operators.	3-37
3-7	Arithmetic operand, expression, and result typing relationships.	3-42
3-8	Relational operators	3-45
3-9	Logical operators	3-47
3-10	Precedence among all operators.	3-52
3-11	Statement classification	3-57
3-12	Print control characters.	3-90
6-1	ON and OFF options.	6-5
G-1	Functional unit relationships	G-9
G-2	Significant exponent value ranges	G-12

1 INTRODUCTION

INTRODUCTION

1

1.1 GENERAL

CRAY-1 FORTRAN provides the CRAY-1 Computer System programmer with a means of specifying computer programs in a high-level language while retaining control of the performance and capabilities of the CRAY-1 Computer. This is made possible by the CRAY-1 FORTRAN Compiler, CFT, which transforms CRAY-1 FORTRAN Language statements into highly efficient machine-language instruction sequences, or programs. Programs created by CFT exploit CRAY-1 features without special consideration by the FORTRAN programmer.

The CRAY-1 Operating System (COS) supports CFT and the programs created by CFT by initiating and monitoring their execution.

A working knowledge of CRAY-1 FORTRAN demands an understanding of certain aspects of:

- The CRAY-1 Computer System,
- The CRAY-1 FORTRAN Compiler (CFT), and
- The CRAY-1 Operating System (COS).

1.2 THE CRAY-1 COMPUTER SYSTEM

Section 2 describes the fundamentals of CRAY-1 Computer System organization, capabilities, and use. It emphasizes those areas of greater significance to the CRAY-1 FORTRAN programmer such as memory size, data formats and vector processing capabilities.

1.3 THE CRAY-1 FORTRAN LANGUAGE

Section 3 fully describes the CRAY-1 FORTRAN language. The presentation is progressive, proceeding from the identities and forms of basic syntactic elements through the full range of CRAY-1 FORTRAN statement types and applications. Examples reinforce more formal explanations and confirm the reader's correct understanding of the material.

1.4 CRAY-1 FORTRAN PROGRAMMING

Section 4 identifies characteristics of the CRAY-1 Computer System, the consideration of which would improve the performance of computer programs written in the CRAY-1 FORTRAN Language. Additionally, this section defines CRAY-1 Operating System (COS) methods for organizing and referencing these programs and related data.

1.5 THE CRAY-1 FORTRAN COMPILER (CFT)

During the transformation of CRAY-1 FORTRAN Language statements into executable machine language programs, the CRAY-1 FORTRAN Compiler (CFT) produces symbolic listings and other information for the user. Additionally, the user may control these and certain other compiler actions. Section 5 describes CFT operations supporting the CRAY-1 FORTRAN programmer.

1.6 THE CRAY-1 OPERATING SYSTEM (COS)

Section 6 presents those features of the CRAY-1 Operating System (COS) provided for use by the CRAY-1 FORTRAN programmer. It describes the creation, loading and execution of computer programs compiled by CFT.

2 THE CRAY-1 COMPUTER SYSTEM

THE CRAY-1 COMPUTER SYSTEM

2

2.1 GENERAL

This section provides an overview of the CRAY-1 Computer System. Emphasis is on those features of the central processing unit of significance to the CRAY-1 FORTRAN programmer. A complete description of the CRAY-1 is documented in the "CRAY-1 Reference Manual" (2240004).

The CRAY-1 Computer System is a large-scale, general-purpose, scientific digital computer designed for high-speed computation on large volumes of data. It is capable of storing over a million binary computer words of instructions and data and of providing computational results in excess of 100-million per second. The CRAY-1 has twelve high-speed input/output channels for data transfers to and from its mass storage subsystem, one or more computer systems, and its Maintenance Control Unit.

The CRAY-1 FORTRAN Compiler (CFT), while closely adhering to the "ANSI FORTRAN X3.9-1966" FORTRAN Language standards, draws upon capabilities unique to the CRAY-1 Computer System. Machine language programs produced by CFT are capable of highly efficient use of the CRAY-1's resources without the user specifying more than standard FORTRAN constructs. There are, however, certain fundamental characteristics of the CRAY-1, as with any computer programmed in FORTRAN, that may enhance the effectiveness of FORTRAN program specification. These include memory size and speed and data word length and format.

2.2 SYSTEM OVERVIEW

The CRAY-1 Computer System serves as an extremely powerful computational resource for one or more "host" processors. Although capable of operating in a stand-alone mode, the CRAY-1 is normally dedicated to computation with a host processor controlling low-speed peripheral equipments.

The configuration of the CRAY-1 Computer System consists of:

- The central processing unit (CPU),
- The mass storage subsystem,
- The external interface unit(s),
- The maintenance control unit (MCU),
- Local input/output stations (optional), and
- The CRAY-1 operating system (COS).

As illustrated in figure 2-1, the Central Processing Unit (CPU) is the nucleus of CRAY-1 Computer System operations. The CPU not only receives, transmits, and processes all information entering or leaving the CRAY-1 but also controls the operation of all other system components supporting these CPU activities.

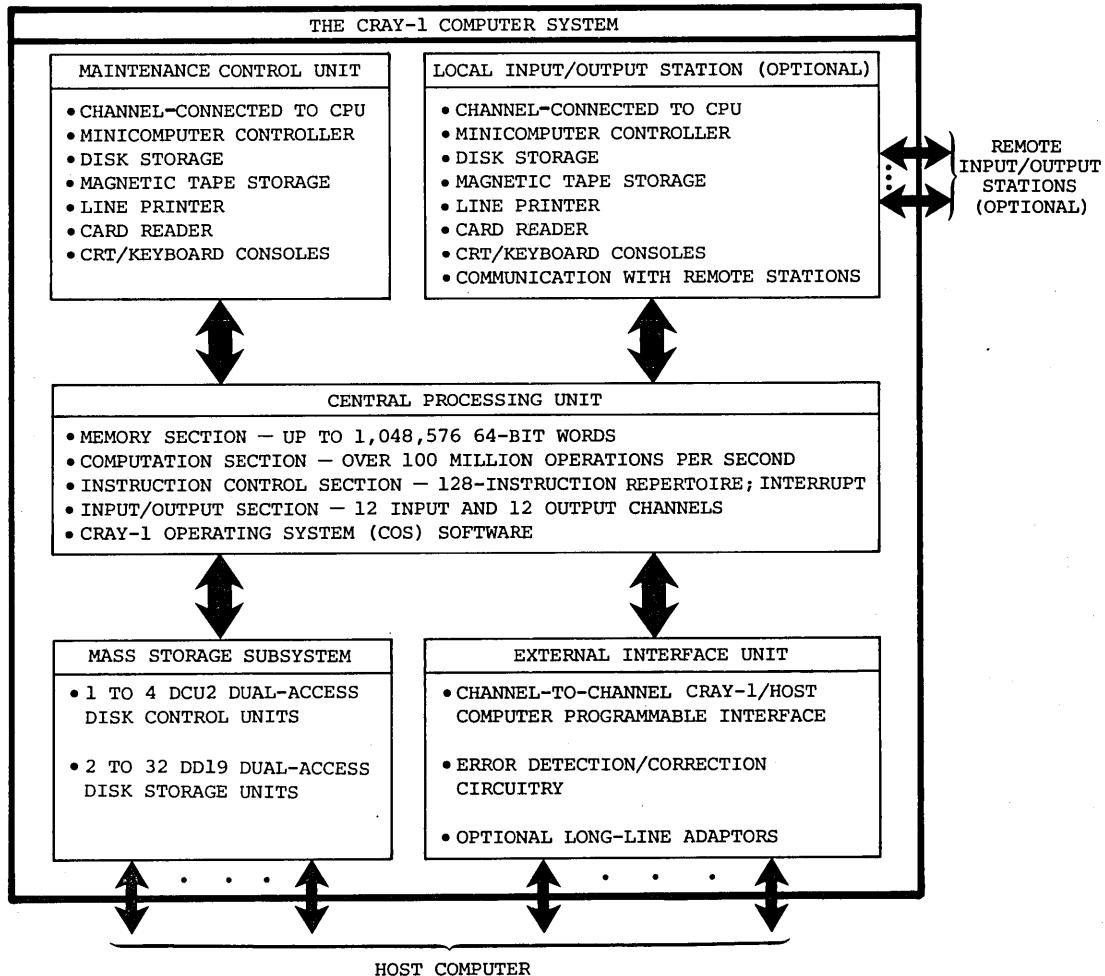


Figure 2-1. The CRAY-1 Computer System

The Mass Storage Subsystem consists of from one to four DCU-2 Disk Control Units, each capable of interfacing four DD-19 Disk Storage Units to the CPU. Each DD-19 Disk Storage Unit can hold 37 million 64-bit words and can transfer these to and from the CPU at a sustained rate of 500,000 words per second.

During operational use, one or more external interface units control the high-speed transfer of program and data information between the CRAY-1 Computer System and one or more host processors. These units support intercomputer data transmission protocol, compensate for differences in data representation between the two systems, and perform extensive data validation and correction operations.

The Maintenance Control Unit (MCU) is a minicomputer connected to the CPU to support field engineering in their maintenance of the CRAY-1 Computer System.

One or more local input/output stations may be channel-connected to the CPU to control card readers, line printers, magnetic tape drives, and similar peripheral equipments to be used for data input and output. Additionally, each can control communications with one or more remote input/output stations.

The CRAY-1 Operating System (COS) resides in the CPU during operation of the CRAY-1 Computer System. COS coordinates all system components. COS activities depend upon information accompanying the jobs it processes, conditions in the CRAY-1, and commands received from one or more host computer systems. The multiprogramming environment in which jobs are processed by the CRAY-1 is established and maintained by COS.

2.3 THE CENTRAL PROCESSING UNIT

The Central Processing Unit (CPU) of the CRAY-1 Computer System (Figure 2-2) consists of:

- The memory section,
- The instruction control section,
- The computation section, and
- The input/output section.

The CRAY-1 Operating System maintains constant control of the CPU and its interaction with other system components during normal operation. The CPU is described further in Appendix G.

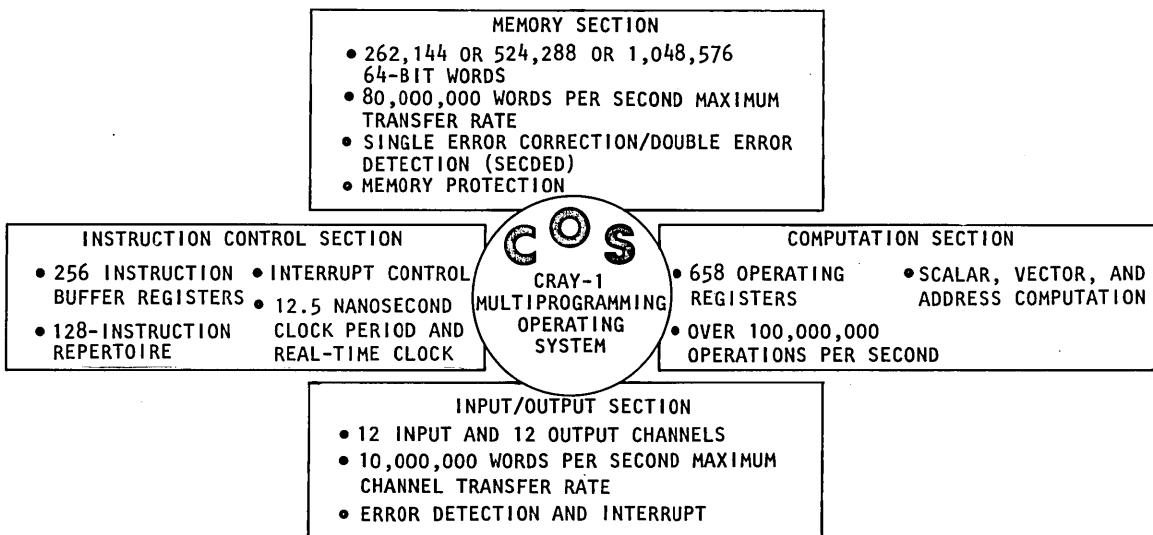


Figure 2-2. The CRAY-1 central processing unit

3 THE CRAY-1 FORTRAN LANGUAGE

3.1 GENERAL DESCRIPTION

FORTRAN is a system of notation devised for easy and accurate computer program specification. As such, it is a language in which the clarity of a mathematically oriented grammar is readily learned and applied. Ordered sets of alphabetic, numeric, and special characters are used to construct FORTRAN statements which, in turn, are ordered to describe a computer program. FORTRAN permits computer program specification with little dependence upon the characteristics of the computer system to be used.

This section progressively develops the means for program specification using the CRAY-1 FORTRAN language. The fundamentals of its notation and syntax are introduced first, then are used as a basis for progressing into the complete range of capabilities afforded by CRAY-1 FORTRAN.

3.1.1 THE CRAY-1 FORTRAN COMPILER

The CRAY-1 FORTRAN Compiler (CFT) converts statements in the FORTRAN language to the binary machine language of the CRAY-1 Computer. In so doing, CFT constructs machine-language instruction sequences that cause the full range of CRAY-1 features and capabilities to be applied during program execution. As long as the same standards of FORTRAN language expression are adhered to, FORTRAN compilers of other computer systems will accept and convert the same statements into machine-language instruction sequences for those systems.

3.1.2 CONFORMANCE TO THE ANSI STANDARD

Specifications for the CRAY-1 FORTRAN language are based upon standards established in 1966 by the American Standards Association and documented in the American National Standards Institute (ANSI) publication "ANSI FORTRAN X3.9-1966". Extensions to these standards afford the CRAY-1 FORTRAN language programmer a broader range of capabilities. Most have been developed to correspond to many of those proposed by the American National Standards Committee X3J3 as part of a new FORTRAN standard. Notes are included at appropriate points throughout this section identifying deviations from the 1966 ANSI Standard.

3.1.3 SYNTAX DESCRIPTION

A conventional notation is used in this publication to describe the syntax of FORTRAN statement forms. It consists of ordered sequences of the following elements:

- The square brackets, [], indicate the optional use of the items they enclose.
- An italicized lower-case letter or letters, *thus*, identifies the use of a certain type of item.
- Numbers, symbols, and upper-case letters (e.g., E+23) indicate their actual use.
- The ellipsis, ... , indicates the optional use of the preceding item one or more times in succession.

Except where specifically stated otherwise, blank characters are ignored and may be used to enhance readability.

For example,

```
PRINT f[, iolist ]
```

where *f* is a FORMAT statement identifier and
iolist is an I/O list

describes the syntactical construct that begins with the letters "PRINT" followed by those symbols identifying a FORMAT statement identifier and, optionally, a comma and one or more sets of symbols identifying I/O list items separated by commas. The FORTRAN language statements

```
PRINT 88
PRINT 1234,A,B,C,X,Y,Z
PRINT6,VALUE
PRINT 0054,ALPHA,BETA,GAMMA,DELTA,ETCETERA
```

thus comply with this form assuming the use of FORMAT statement identifiers and I/O list items is proper.

3.2 FUNDAMENTALS

Many of the basic terms and concepts used throughout this section are introduced below. Words and phrases appearing in italics indicate their being defined at that point.

3.2.1 NOTATION

The FORTRAN language embodies a syntax described in the following paragraphs. Components of this notation are numbers and letters and the special characters identified in the character sets below. Certain sequences of these are called syntactic items and may be grouped into FORTRAN statements which, in turn, are ordered into program units.

Character sets

Two sets of characters are used in CRAY-1 FORTRAN language notation. Upper-case letters, digits, and certain special characters belong to the FORTRAN character set. All other characters representable in the CRAY-1 computer belong to the auxiliary character set. Appendix A describes these characters and their internal codes.

The FORTRAN character set - The *FORTRAN character set* consists of: the twenty-six upper-case letters, A-Z; the ten digits, 0-9; and the fourteen *special characters* described in table 3-1. An *alphanumeric character* is any letter or digit.

The internal code for each of these characters is described in Appendix A. The relative magnitudes of these character codes establishes their *collation sequence*. Note that digits precede letters in this collation sequence.

The auxiliary character set - Appendix A contains the complete CRAY-1 set of characters and the codes used for the internal representation of each. Those characters not in the FORTRAN character set are members of the *auxiliary character set* and are of limited use.

The ANSI FORTRAN Standard does not specify an auxiliary character set or a collation sequence.

Sequences

A *sequence* is a set of n elements ordered in a one-to-one correspondence with the ordinals 1,2, ... ,n. An *empty sequence* contains no elements.

Table 3-1. Special characters

Symbol	Name
	Blank or space
=	Equals
+	Plus
-	Minus or hyphen
*	Asterisk
/	Slash
(Left parenthesis
)	Right parenthesis
,	Comma
.	Decimal point
\$	Dollar sign or currency symbol
'	Apostrophe
"	Quotation mark
:	Colon

The ANSI FORTRAN Standard does not provide for apostrophes, quotation marks, or colons.

Syntactic items

Syntactic items of the FORTRAN language are formed with sequences of FORTRAN character set elements, and include:

- Constants,
- Symbolic names,
- Statement labels,
- Keywords, and
- Operators.

Constants - A *constant* is a syntactic item representing an unvarying value. Several types of constants are illustrated below and are more fully described in 3.3.3.

Examples:

<u>Representation</u>	<u>Type</u>	<u>Value</u>
1024	Integer	1024
10.E1	Real	100
1.5	Real	1.5
.FALSE.	Logical	false
.TRUE.	Logical	true
72.	Real	72
6HCRAY-1	Hollerith	CRAY-1
75.63D-2	Double precision	.7563
(6.1,-3.2)	Complex	6.1+(-3.2 $\sqrt{-1}$)

Symbolic names - A *symbolic name* declares or references a program unit, procedure, or value. It is composed of from one to eight alphanumeric characters the first of which must be a letter. Leading, trailing, and embedded blank characters are ignored.

The ANSI FORTRAN Standard limits a symbolic name to a maximum of six characters, and does not provide for the symbolic name of constants.

Examples:

DATAONE	F293	SIN ALPHA
DATA1	U 238	TEST1234
D A T A 1	ARRAY TWO	MAIN PROG

Statement labels - A *statement label* uniquely identifies a statement in a program unit to permit its being referenced by other statements in the same program unit. A statement label is composed of from one to five digits. Leading zeros and leading, trailing, and embedded blank characters do not alter the identity of a statement label. They are counted as part of a five character maximum only when the statement label appears with the statement it identifies.

Examples:

12345	6681
1 2 3 4 5	470
1 2 345	00673
96	673
8277	2 2

Keywords - A *keyword* is a prespecified sequence of letters having special significance in FORTRAN language statements. Leading, trailing, and embedded blanks occurring in a keyword are ignored. Examples of keywords used in FORTRAN statements appear in table 3-11. The possible similarity of a keyword and a symbolic name poses no problem because of the context in which each is used.

Operators - An *operator* is expressed as one or two special characters augmented, in certain cases, with letters. Leading, trailing or embedded blanks do not affect the identity of an operator. Operators are used to specify arithmetic, relational, and logical operations within program units.

Examples:

<u>Representation</u>	<u>Type</u>	<u>Meaning</u>
+	Arithmetic	Addition
**	Arithmetic	Exponentiation
.AND.	Logical	Conjunction
.OR.	Logical	Disjunction
.EQ.	Relational	Equal
.GT.	Relational	Greater than

Additional syntactic items - Certain special characters can be used in a manner that defines their being syntactic items. Their forms and uses will be described at appropriate points. Adjacent blank characters have no effect on their interpretation.

Lists and list items

A *list* is a sequence of syntactic items separated from each other by the special character comma. The syntactic items appearing in a list are called *list items*. Blank characters preceding and following list items do not affect their interpretation.

Examples:

A,B,C,D,E	701,55,100
ARRAY1, VALUE2,X,ABC	UNO,DOS,TRES

3.2.2 STATEMENTS

A FORTRAN *statement* is a sequence of syntactic items beginning, in most cases, with a keyword. It serves as a fundamental component in a FORTRAN program specification by describing either the form of data and program elements or the actions to be taken by the program. A statement label may precede a statement, but is not a part of the statement itself. The type of a statement is indicated by the keyword it contains or by its form. Examples appear throughout 3.6. The total number of characters used to express a statement is limited to 1,320 and includes blank characters. Aside from this character-count limitation, leading, trailing, and embedded blank characters do not affect statement interpretation.

A statement is classified as either an *executable statement* or a *non-executable statement*. An executable statement is one that specifies an action. A non-executable statement is an inactive descriptor of data or program form.

3.2.3 THE FORTRAN CODING FORM

A FORTRAN coding form is a convenience in expressing FORTRAN language notation and facilitates its transcription into punched cards or some other computer input medium. Figure 3-1 illustrates this form.

FORTRAN coding form format

Each horizontal row on the form provides 80 positions (or columns) for the specification of 80 characters from the character set. Only the leftmost 72 columns (columns 1 through 72) are used to specify a FORTRAN program. The content of the remaining eight columns is commonly used for identifying and sequencing punched cards but may be unused or used for other purposes.

The first 6 columns have positional significance in FORTRAN and are delimited on the form by heavier vertical lines.

All or part of a FORTRAN statement can be expressed in columns 7 through 72 of the form.

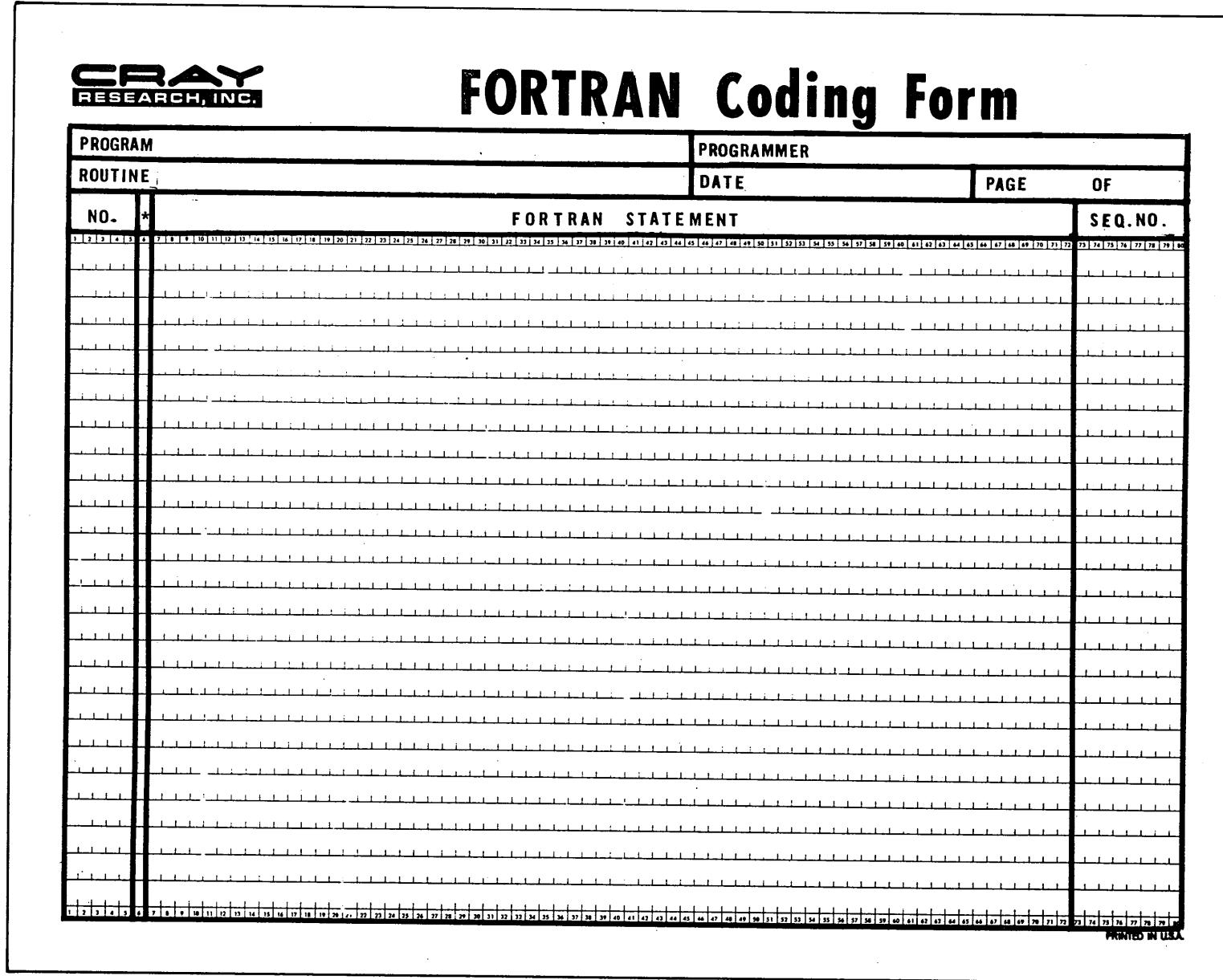


Figure 3-1. FORTRAN coding form

Lines

A single row of information on a FORTRAN coding form is referred to as a *line*. A line contains a sequence of 72 characters. (Note that an unentered position of the row and an unpunched column of the card represent the special character "blank".) All notation required to describe a FORTRAN program is expressed as an ordered sequence of lines of which there are five types:

- Comment
- Initial
- Continuation
- Terminal
- Compiler directive

Comment lines - A line having the letter C or an asterisk in column one may be a *comment line*. The content of columns 2 through 72 of a comment line has no effect on the FORTRAN program being created. These columns typically contain descriptive commentary and are inserted before and after relevant lines of the program (See also "Compiler directive lines" below.)

The ANSI FORTRAN Standard does not provide for the use of an asterisk in column 1 to denote a comment line.

Initial lines - The *initial line* expresses all or the initial part of a single FORTRAN statement in columns 7 through 72. It may have a statement label of five digits and/or blank characters in columns 1 through 5. An initial line has neither the letter C nor an asterisk in column 1, and must have either the digit 0 or a blank character in column 6. A terminal line is a special form of initial line.

Continuation lines - One or more *continuation lines* may be used to extend the capacity of an initial line when expressing a single FORTRAN statement. A continuation line has neither the letter C nor an asterisk in column 1, has a character other than zero or blank in column 6, and contains a portion of a FORTRAN statement in columns 7 through 72. A sequence of one initial line followed by up to nineteen continuation lines may be used for a single FORTRAN statement. This sequence of lines may have any number of comment lines interspersed. The initial line of such a sequence must not appear to be a terminal line.

Terminal lines - A single *terminal line* must be used as the last line of every program unit. A terminal line is a special form of initial line that completely contains an END statement (i.e., the letters "E", "N", and "D" appearing in that order anywhere in columns 7 through 72).

Compiler directive lines - A line having the characters CDIR\$ in columns 1 through 5 is a *compiler directive line* and may be used to contain one or more compiler directives. These lines and the compiler directives they contain are described in Section 5.

3.2.4 THE EXECUTABLE PROGRAM

An *executable program* is an ordered set of FORTRAN statements grouped into one or more program units. Certain program units are capable of referencing pre-established procedures. Thus, computer program specifications are established from two sources:

- The FORTRAN statements comprising the executable program, and
- Pre-established procedures referenced from certain of these FORTRAN statements.

Figure 3-2 illustrates these program units, the procedures they reference, and the overall organization of these entities in the executable program.

The ANSI FORTRAN Standard separately provides for basic external functions; CRAY-1 FORTRAN includes these in its set of intrinsic functions. The ANSI FORTRAN Standard categories of function procedure, subroutine procedure, and external subroutine have also been eliminated.

Program units

The set of *program units* used in an executable program must include one main program and may also include one or more subprograms.

The main program - An executable program must have one and only one *main program*. Execution of an executable program begins with its main program. The main program may reference none, one, or more than one procedure during its execution provided each is defined prior to main program initiation.

The main program usually, but not necessarily, terminates the execution of an executable program. The first statement of a main program's specification may be a PROGRAM statement. The last line must contain an END statement.

The ANSI FORTRAN Standard does not provide for the use of a PROGRAM statement.

Subprograms - The two types of *subprograms* are: specification and procedure. Specification subprograms are non-executable; procedure subprograms are executable.

Specification subprograms

The only form of a *specification subprogram* is the block data subprogram.

Block data subprograms - *Block data subprograms* associate data with a type, form, name, and initial value prior to executable program execution. The use of block data subprograms does not preclude other means by which an executable program might be provided data prior to its execution. Block data subprograms cannot contain any executable statements except the END statement. Its first statement must be a BLOCK DATA statement; the last line must contain an END statement.

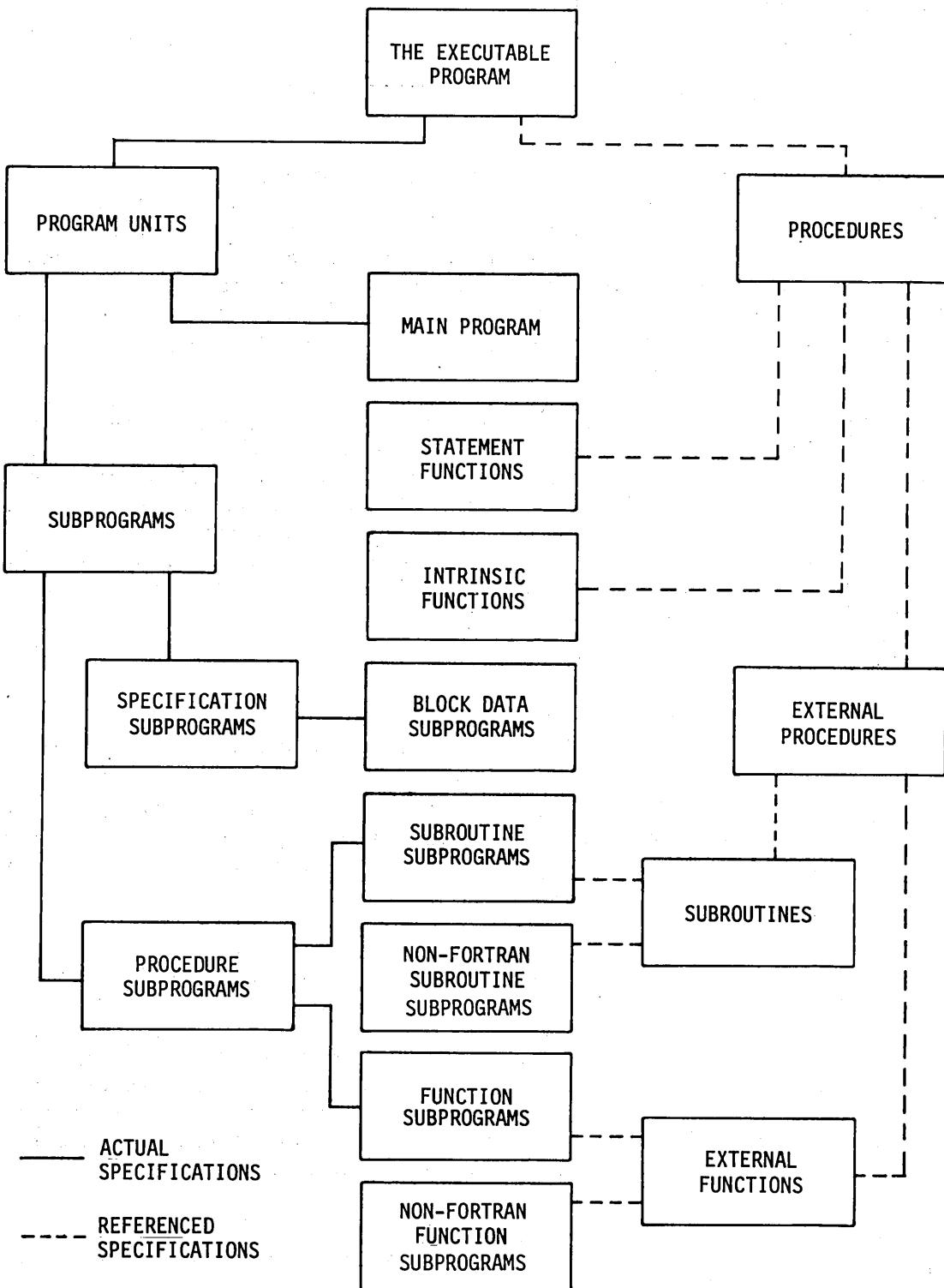


Figure 3-2. The executable program

Procedure subprograms

Procedure subprograms are of two types; subroutine subprograms and function subprograms. Both types are executable, but differ in the manner by which they are defined and referenced.

Subroutine subprograms - *Subroutine subprograms* establish a common set of executable code that can be referenced from a main program or a procedure subprogram. The first statement must be a SUBROUTINE statement; the last line must contain an END statement. A subroutine subprogram must be referenced through use of a CALL statement in the referencing program unit. A subroutine subprogram is classified as a procedure as well as a program unit in the hierarchy of an executable program.

Function subprograms - *Function subprograms* also establish a common set of executable code that can be referenced from a main program or a procedure subprogram. The first line must contain a FUNCTION statement; the last line must contain an END statement. A function subprogram, unlike a subroutine subprogram, is referenced by the appearance of its identifier in certain types of statements. A function subprogram is classified as a procedure as well as a program unit in the hierarchy of an executable program.

Procedures

Procedures are optional in an executable program and include statement functions, intrinsic functions, and external procedures.

Statement functions - *Statement functions* can be specified within a main program, a function subprogram, or a subroutine subprogram. A given statement function can only be referenced from a statement within the same program unit containing its specification.

Intrinsic functions - Numerous commonly-used operations called intrinsic functions have prespecified identities and functions. An *intrinsic function* may be reference by a main program or a procedure subprogram. The entire set of operations called for by the ANSI FORTRAN Standard is included. Appendix B summarizes this set. CRAY-1 FORTRAN also provides for utility procedures. These procedures are classified as intrinsic functions, are described in 3.6.12 and in Appendix C, and perform numerous utility operations not called for in the ANSI Standard.

The ANSI FORTRAN Standard Basic External Functions are included in the CRAY-1 FORTRAN set of Intrinsic Functions. The ANSI FORTRAN Standard does not provide for Utility Procedures.

External procedures - An *external procedure* is named for its ability to be specified external to any other procedure referencing it. The two types of external procedure are subroutines and external functions.

Subroutines

Subroutines are of two types; the subroutine subprogram, and the non-FORTRAN subroutine subprogram.

Subroutine subprograms - As stated earlier, the *subroutine subprogram* is categorized as both a program unit and a procedure. Its description in both cases is the same.

Non-FORTRAN subroutine subprograms - The *non-FORTRAN subroutine subprogram* is a set of executable code that functions the same as a subroutine subprogram. It is prepared by some means other than FORTRAN. Typically, the non-FORTRAN subroutine subprogram is written in CRAY-1 assembly language, in a high-order language other than FORTRAN, or in a version of FORTRAN not compatible with the one in use. Such are separately compiled or assembled and are available in binary form upon reference during program execution. (Appendix F describes the creation of non-FORTRAN subroutine subprograms using CAL, the CRAY-1 Assembly Language.)

External functions

External functions are of two types; the function subprogram, and the non-FORTRAN function subprogram.

Function subprograms - As stated earlier, the *function subprogram* is categorized as both a program unit as well as a procedure. Its description in both cases is the same.

Non-FORTRAN function subprograms - The *non-FORTRAN function subprogram* is a set of executable code that functions the same as a function subprogram during execution of the executable program, but which is prepared by some means other than FORTRAN. The non-FORTRAN function subprogram is typically written in CRAY-1 assembly language, in a high-order language other than FORTRAN, or in a version of FORTRAN not compatible with the one in use. Such are separately compiled or assembled and are available in binary form upon reference during program execution. (Appendix F describes the creation of non-FORTRAN function subprograms using CAL, the CRAY-1 Assembly Language.)

3.2.5 DATA REPRESENTATIONS

The following are fundamentals of data representation and processing. A complete description of data types, their specification, and other characteristics are further developed in 3.3.

Data types

The seven types of data that can be specified in CRAY-1 FORTRAN are:

- *Integer* data, which are whole, signed values;
- *Real* data, which are signed mixed-value approximations (whole number plus fraction);
- *Double-precision* data, which are signed, mixed-value approximations extended to approximately twice the precision of real data;
- *Complex* data, which approximate complex values as pairs of signed, mixed-value approximations (the first member of a pair for the real part, the second for the imaginary part);
- *Logical* data, which represent the truth values "true" and "false";
- *Boolean* data, which are octal values representing the binary contents of CRAY-1 computer words; and
- *Hollerith* data, which are sequences of characters.

The ANSI FORTRAN Standard does not provide for Boolean data specification.

Data specification

Data are specified as constants in a FORTRAN program. The form in which a constant is expressed identifies the type of datum it represents, and is described in 3.3.

Variables, arrays, array elements, and functions

Data may be contained in entities known as variables, arrays, and array elements. A *variable* or an *array element* contains a single datum. An *array* contains one or more array elements arranged into as many as seven dimensions.

All data in an array are of the same type. A function creates a datum when referenced.

The ANSI FORTRAN Standard permits a maximum of three dimensions in an array.

Definition

During program execution, the content of a given variable or array element is either *defined* or *undefined*. A *defined* variable or array element contains a value. An *undefined* variable or array element does not contain a predictable value. Once defined, a variable or array element contains a specific value until it is undefined or is redefined with a different value. All variables and array elements are initially undefined and remain so until action prior to or during program execution defines them. Variables and array elements defined prior to program execution are known as being *initially defined*. Constants are always defined and are never redefined. A function's value is defined only at that point in program execution where it is required.

Reference

A constant, variable, array element, or function *reference* occurs when its symbolic name appears in a context where a value is required. A reference to a variable or array element provides the value currently contained by that entity without modifying that value. Reference to a constant provides its invariant value, which cannot be modified. Reference to a function causes a value to be defined.

Storage

A *storage sequence* is a sequence of storage units. A storage unit corresponds to a 64-bit CRAY-1 computer word. An integer, real, or logical datum occupies one *storage unit*; a double precision or complex datum occupies two storage units. Datum requiring more than one storage unit in a storage sequence occupies consecutive locations in memory.

The ANSI FORTRAN Standard does not specify the relationship between storage units and computer words.

The term *storage sequence* describes relationships that associate variables, array elements, arrays, and common blocks.

Association

Association occurs when a datum may be identified by different symbolic names or from different program units.

3.2.6 ORDER OF STATEMENTS AND LINES

The first statement of a main program may be a PROGRAM statement. The first statement of a subprogram must be a FUNCTION, SUBROUTINE, or BLOCK DATA statement.

Within a program unit permitting them:

- FORMAT statements may appear anywhere,
- All specification statements must precede all DATA statements, statement definition function statements, and executable statements,
- All statement definition function statements must precede all executable statements, and
- DATA statements may appear anywhere after the specification statements.
- Within the specification statements of a program unit, IMPLICIT statements must precede all specification statements other than PARAMETER statements.
- A PARAMETER statement must precede all other statements containing the symbolic names of constants that it specifies.
- The last line of a program unit must contain an END statement.

Table 3-2 illustrates the required order of statements and lines for a program unit. Vertical lines delimit varieties of statements that may be interspersed. For example, FORMAT statements may be interspersed with PARAMETER, DATA, executable, and statement function definition statements. Horizontal lines delimit varieties of statements that must not be interspersed. For example, statement function definition statements must not be interspersed among executable statements. The top-to-bottom order indicates the first-to-last appearance of lines and statements in a program unit. Note that an END statement must appear in the last line of a program unit and cannot be followed by a comment line intended as a part of that same program unit.

Table 3-2. Required order of lines and statements

PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA statement			
Comment and compiler directive lines	ENTRY and FORMAT statements	PARAMETER statements†	IMPLICIT statements †
			Other specifi- cation state- ments
		DATA statements	Statement func- tion definition statements
			Other executable statements
END statement			

† Note the restrictions described for the interspersing of IMPLICIT and PARAMETER statements and for compiler directive lines (see Section 5).

3.2.7 NORMAL EXECUTION SEQUENCE

A *normal execution sequence* is an execution of executable statements in their order of appearance in a program unit. Executable program execution begins with the first executable statement of the main program. When a subprogram is referenced, execution begins with the first executable statement of that subprogram. When a subprogram entry is referenced, execution begins with the first executable statement following the ENTRY statement named in that reference.

Statements that may alter the normal execution sequence are:

- Unconditional, assigned, and computed GO TO,
- Two- and three-branch arithmetic IF,
- RETURN,
- PAUSE,
- STOP,
- Indirect logical IF,
- Direct logical IF statements containing any of the above forms,

- DO,
- The terminal statement of a DO-loop, and
- END.

The ANSI FORTRAN Standard does not provide for a two-branch arithmetic IF statement, an indirect logical IF statement, or for a DO statement altering the normal execution sequence. The direct logical IF statement is identical to the logical IF statement described in the ANSI FORTRAN Standard. The three-branch arithmetic IF statement corresponds to the ANSI arithmetic IF statement.

Normal execution sequence is not affected by the interspersing of non-executable statements or comment lines among executable statements.

A procedure subprogram must not be referenced twice without the execution of a RETURN or END statement in that procedure.

3.3 DATA

3.3.1 TYPES OF DATA

The types of data are:

- Integer,
- Real,
- Double precision,
- Complex,
- Logical,
- Boolean, and
- Hollerith.

The specification and internal representation differs for each type.

The ANSI FORTRAN Standard does not provide for Boolean data.

3.3.2 DATA IDENTIFIERS

The symbolic name of a constant, variable, array, array element, or function identifies its data type. Once a name is identified with a particular type, that type is implied for all references to that name.

A constant, variable, array, external function or statement function may have its type specified by the appearance of its symbolic name in a type statement. Otherwise, the type is implied by the first letter of its name. A first letter of I, J, K, L, M, or N implies type integer; any other first letter implies type real. An IMPLICIT statement can be used to change *implied typing*.

The data type of an array element is the same as the data type of the array within which it exists. An array name is the initial part of the names of each of its array elements.

The data type of a function establishes the type of datum provided when the function is referenced in an expression.

The name of an intrinsic function is (1) prespecified to agree with the type of datum provided, (2) cannot be retyped, and (3) is defined in Appendix B for each function.

The correspondence between an external function name and the type of datum provided when it is referenced is established the same as for array and variable names.

The data type of a function subprogram can be specified by its name, in a type statement, or in the FUNCTION statement used to name the subprogram. An IMPLICIT statement within a function subprogram may affect the type of the subprogram name.

A PARAMETER statement can assign a symbolic name to a constant. The initial letter of a constant name specifies its type.

Examples:

<u>The symbolic name ...</u>	<u>Identifies a(n) ...</u>	<u>of type ...</u>	<u>if also appearing in the following statement:</u>
FLOAT6	Constant	Real	PARAMETER
NUMBER1	Constant	Integer	PARAMETER
COUNTER	Variable	Real	- - -
INDEX4	Variable	Integer	- - -
ARRAY	Array	Real	COMMON, DIMENSION, or REAL
MATRIX	Array	Integer	COMMON, DIMENSION, or INTEGER
ARRAY(2,4)	Array element	Real	- - -
MATRIX(6,100,2)	Array element	Integer	- - -
CM(IN)	Function	Real	FUNCTION or statement function
METERS(YDS,FT)	Function	Integer	FUNCTION or statement function

3.3.3 CONSTANTS

Within an executable program, all constants expressed in the same form have the same invariant value. The value zero is considered neither positive nor negative. A signed zero has the same value as an unsigned zero.

The form of the character sequence representing a constant specifies both its value and its data type. A PARAMETER statement allows a constant to be given a symbolic name. The first letter of a constant name specifies its type.

Except within Hollerith constants, blank characters occurring in a constant have no effect on its value.

Integer, real, double precision, and complex constants are *arithmetic constants*.

An *unsigned constant* is an arithmetic constant without a leading sign. A *signed constant* has a leading plus or minus. An *optionally signed constant* may be either signed or unsigned. Arithmetic constants are optionally signed except where otherwise specified.

Integer constants

An integer datum is always an exact representation of an integer value. It may assume a positive or negative integral value or a zero value. An integer datum occupies one numeric storage unit in a storage sequence.

The form of an *integer constant* is an optional sign followed by a nonempty sequence of digits specifying a decimal integer value.

Integer constants are represented in the CRAY-1 computer by integral binary values (I) in the range

$$-2^{63} \leq I < 2^{63}$$

This is approximately the decimal range

$$0 \leq |I| < 10^{19}$$

(A special form of integer constant is discussed in Section 5 under "Compiler directives".)

See examples on page 3-22.

The ANSI FORTRAN Standard does not specify a range of values for integer constants.

Real constants

A real datum is an approximation to the value of a real number. It may assume a positive, negative, or zero value. A real datum occupies one numeric storage unit in a storage sequence.

A *real constant* may be expressed as:

- A basic real constant,
- A basic real constant followed by a real exponent, or
- An integer constant followed by a real exponent.

The ANSI FORTRAN Standard does not provide for this use of an integer constant.

A *basic real constant* consists of an optional sign, an integer part, a decimal point, and a fractional part, in that order. Both the integer part and the fractional part are sequences of digits representing integral and fractional decimal values, respectively. Either but not both of these parts may be omitted. A basic real constant may be written with more digits than can be used to approximate its value.

Examples:

VALUE	INTEGER CONSTANT	REAL CONSTANT	DOUBLE-PRECISION CONSTANT	COMPLEX CONSTANT
0	0	0.	0D	(0.,0.)
692	692	692. 692.0 692E0 692.E0 692.0E0 6920E-1 .692E3 6.92E2	692D0 692.D0 692.0D0 6920D-1 .692D3 6.92D2	(692.,0.) (692.0,0.) (692E0,0.) (692.E0,0.) (692.0E0,0.) (6920E-1,0.) (.692E3,0.) (6.92E2,0.)
6.128547472		6.128547472 6.128547472E0 6128547472E-9 6128547472.0E-9 .6128547472E1 612.8547472E-2	6.128547472D0 6128547472D-9 6128547472.0D-9 .6128547472D1 612.8547472D-2	(6.128547472,0.) (6.128547472E0,0.) (6128547472E-9,0.) (6128547472.0E-9,0.) (.6128547472E1,0.) (612.8547472E-2,0.)
.875·√-1				(0.,.875) (0.,875E-3) (0.,.875E0) (0.,8.75E-1) (0.,.000000875E6)
692+.875·√-1				(692.,.875) (692E0,0.875) (69.2E1,875E-3) (.692E3,875.E-3) (6.92E2,8.75E-1)

NOTE: The negative of a non-zero constant, exponent value, or complex portion is formed by preceding its expression with a minus (-). The use of a plus (+) in this position or the absence of either sign denotes a positive.

The form of a *real exponent* is the letter E followed by an optionally signed integer constant. A real exponent denotes that power of ten by which the basic real constant is to be multiplied during its interpretation as a decimal value. The decimal point in a basic real constant is optional if there is no fractional part and if a real exponent is specified.

Non-zero real constants are represented in the CRAY-1 computer by normalized floating-point binary values (R) in the range

$$2^{-8192} \cdot (\frac{1}{2}) \leq |R| < 2^{8192} \cdot (\frac{1}{2})$$

and with 48 significant binary digits of precision. This approximates to the decimal range

$$10^{-2500} < |R| < 10^{2500}$$

and to 15 decimal digits of precision.

See examples on page 3-22.

The ANSI FORTRAN Standard does not specify a range of values for real constants.

Double-precision constants

A double-precision datum is an approximation to the value of a real number and has approximately twice the precision of a real datum. A double-precision datum may be positive, negative, or zero, and occupies two consecutive numeric storage units in a storage sequence.

A *double-precision constant* may be expressed as:

- A basic real constant followed by a double-precision exponent, or
- An integer constant followed by a double-precision exponent.

The ANSI FORTRAN Standard does not provide for this use of an integer constant.

The form of a *double-precision exponent* is the letter D followed by an optionally signed integer constant. A double-precision exponent denotes that power of ten by which the basic real constant is multiplied during its interpretation as a decimal value.

Non-zero double-precision constants are represented in the CRAY-1 computer by normalized floating-point binary values (D) in the range

$$2^{-8192} \cdot (\frac{1}{2}) \leq |D| < 2^{8192} \cdot (\frac{1}{2})$$

and with 96 significant binary digits of precision. This approximates to the decimal range

$$10^{-2500} < |D| < 10^{2500}$$

and to 29 decimal digits of precision.

See examples on page 3-22.

The ANSI FORTRAN Standard does not specify a range of values for double-precision constants.

Complex constants

A complex datum approximates the value of a complex number and is represented by a pair of real data. The first member of the pair represents the real part and the second the imaginary part of the datum. A complex datum occupies two consecutive numeric storage units in a storage sequence; the first for the real part and the second for the imaginary part.

The form of a *complex constant* is a left parenthesis followed by an ordered pair of optionally signed real constants separated by a comma, and followed by a right parenthesis. The first real constant of the pair is the real part of the complex constant and the second is the imaginary part.

Non-zero complex constant components (where $C = C_r + i \cdot C_i$) are represented in the CRAY-1 computer by two normalized floating-point binary values (C_r, C_i) in the range

$$2^{-8192} \cdot (\frac{1}{2}) \leq |C_r|, |C_i| < 2^{8192} \cdot (\frac{1}{2})$$

and with 48 significant binary digits of precision for each. This approximates to the decimal range

$$10^{-2500} < |C_r|, |C_i| < 10^{2500}$$

and to 15 decimal digits of accuracy for each.

See examples on page 3-22.

The ANSI FORTRAN Standard does not specify a range of values for complex constant components.

Logical constants

A logical datum may assume only the logical values true and false. A logical datum occupies one numeric storage unit in a storage sequence.

The forms, values, and internal representations of a *logical constant* are:

Form	Value	Internal representation
.TRUE. or .T.	true	a negative value
.FALSE. or .F.	false	a zero or positive value

The ANSI FORTRAN Standard does not provide for the .T. or .F. form of logical constant.

Boolean constants

A Boolean datum is a non-numeric set of binary zeros and ones that accounts for the content of each bit position in a single storage unit (64-bit CRAY-1 computer word).

The ANSI FORTRAN Standard does not provide for Boolean constants.

The form of a *Boolean constant* is from 1 to 22 octal digits (0 through 7) followed by the letter B. When all 22 octal digits are used to express a Boolean constant, their binary equivalents directly establish the content of every bit position in the storage unit (64-bit word). In this case, the first octal digit can only be a zero or one and specifies the content of the leftmost bit position (0). Each successive octal digit specifies the contents of the next three bit positions until the last octal digit specifies the contents of the last three bit positions (61, 62 and 63). Fewer than 22 octal digits can be used to directly specify the contents of the right-most bit positions and to place zeros into those (leftmost) bit positions not directly specified.

Examples:

<u>Boolean constant</u>	<u>Internal representation(octal)</u>
1274653312572676113745B	1274653312572676113745
OB	000000000000000000000000
177777777777777777777777B	177777777777777777777777
77740B	0000000000000000000077740
00776B	000000000000000000000776

Hollerith constants

A Hollerith datum is a sequence of any characters capable of internal representation as specified in Appendix A. Its length is the number of characters in the sequence, including blank characters. Each character occupies a position within the storage sequence identified by one of the numbers 1, 2, 3, ... indicating its placement from the left (position 1). A Hollerith datum must contain at least one character.

A Hollerith constant is expressed in either of two forms. The first of these is specified as a non-zero integer constant followed by the letter H and as many characters as equal the value of the integer constant. Thus, the character sequence, ABC 12, would be represented as: 6HABC 12. The second form of Hollerith constant specification delimits the character sequence between a pair of apostrophes. The letter H may optionally follow the second delimiting apostrophe but has no effect on the constant. The character sequence, AB C12, could be represented 'AB C12' or 'AB C12'H. Two adjacent apostrophes appearing within the bounds of two delimiting apostrophes are interpreted and counted as a single apostrophe within the sequence. Thus, the character sequence, DON'T USE "*" would be specified as 'DON''T USE "*" or 'DON''T USE "*"H.

The ANSI FORTRAN Standard does not provide for the apostrophe form of Hollerith constant expression.

Each character of a Hollerith constant character sequence is represented internally by its unique eight-bit code (see Appendix A) with up to eight such codes contained in a single 64-bit CRAY-1 computer word. The codes corresponding to character positions one through eight of a Hollerith constant are sequentially represented from left to right in a CRAY-1 word. Successive groups of eight codes are similarly represented in as many successive CRAY-1 computer words. When the last position of a sequence is not an even multiple of eight, the unused portion of the computer word it occupies is to its right and contains up to seven blank character codes (040_8).

The ANSI FORTRAN Standard does not specify the characteristics of Hollerith constant placement in computer words, or the content of unused computer word portions.

When the number of characters in a character sequence is fewer than eight, the single CRAY-1 computer word used can be caused to have its unused portion contain up to seven null character codes (000). This is accomplished by substituting the letter L for the letter H in the first form of Hollerith constant expression, or by suffixing the second apostrophe delimiter with the letter L in the second form.

When fewer than eight characters appear in a Hollerith constant, the single CRAY-1 computer word used can be caused to have its unused portion contain up to seven null character codes (000) to the left of the one or more codes representing the character sequence. This is accomplished by substituting the letter R for the letter H in the first form of Hollerith constant expression or by suffixing the second apostrophe delimiter with the letter R in the second form.

The following Hollerith constant expressions all yield the same Hollerith constant and differ only in specifying the content and placement of the unused portion of the single CRAY-1 computer word containing the constant:

<u>Hollerith constant</u>	<u>Internal representation (64-bit CRAY-1 computer word)</u>							
(bit position)	(0-7)	8-15	16-23	24-31	32-39	40-47	48-55	56-63
6HCRAY-1	C	R	A	Y	-	1	(040 ₈) (040 ₈)	
'CRAY-1'	C	R	A	Y	-	1	(040 ₈) (040 ₈)	
'CRAY-1'H	C	R	A	Y	-	1	(040 ₈) (040 ₈)	
6LCRAY-1	C	R	A	Y	-	1	(000) (000)	
'CRAY-1'L	C	R	A	Y	-	1	(000) (000)	
6RCRAY-1	(000)	(000)	C	R	A	Y	-	1
'CRAY-1'R	(000)	(000)	C	R	A	Y	-	1

The ANSI FORTRAN Standard does not provide for this use of the letters L or R or for the apostrophe form of Hollerith constant expression.

A Hollerith constant is limited to a maximum of eight characters except when specified in a CALL or DATA statement. An all-zero computer word follows the last word containing a Hollerith constant specified as an actual argument in a function reference or in a CALL statement.

The ANSI FORTRAN Standard does not provide for a limit on the number of characters per Hollerith constant, or for their use in other than CALL and DATA statements.

3.4 ARRAYS

An *array* contains a sequence of data represented in an ordered set of array elements. Each *array element* provides storage for one datum of this sequence. An *array name* identifies an array and the type of datum it contains. An *array element name* is an array name suffixed by a subscript that indicates the placement of an element in the array. In certain instances, an array name denotes its entire set of array elements. An element of an array can be designated only by its array element name. The name of an array and the names of its elements are local to the program unit in which each appears.

3.4.1 ARRAY DECLARATORS

An *array declarator* specifies an array's name and properties. An array can be specified only once within a given program unit. Array declarators are expressed as list items in certain non-executable FORTRAN statements.

Form of an array declarator

The form of an array declarator is

$\alpha (d [,d] \dots)$

where α is the name of the array, and

d is a dimension declarator.

The number of dimensions of an array is the number of dimension declarators appearing in its array declarator. The minimum number of dimensions is one and the maximum is seven.

The ANSI FORTRAN Standard provides for a maximum of three dimensions.

A *dimension declarator* specifies the number of array elements in one dimension of an array and is expressed as a non-zero, positive integer constant or variable.

Kinds of array declarators

Each array declarator is either a constant array declarator or an adjustable array declarator. A *constant array declarator* is an array declarator in which each dimension declarator is expressed as an integer constant. An *adjustable array declarator* is an array declarator containing one or more dimension declarators expressed as variables. Adjustable array declarators are only specified in procedure subprograms. The arrays they describe are defined upon procedure subprogram entry with suitable arguments for establishing array association and dimensionality.

An array declarator is either a dummy array declarator or an actual array declarator. A *dummy array declarator* is a constant or adjustable array declarator that appears only in function or subroutine subprograms. A dummy array declarator is permitted in a DIMENSION or type statement but not in a COMMON statement. An array name used as a dummy array declarator in a function or subroutine subprogram must also appear as an argument in its FUNCTION, SUBROUTINE, or ENTRY statement.

An *actual array declarator* is a constant array declarator having an array name that is also an actual argument. An actual array declarator is permitted in a DIMENSION, COMMON, or type statement.

3.4.2 PROPERTIES OF AN ARRAY

The data type of an array and of its array elements is identified by its name. The properties of an array specified by its array declarator are the number of its dimensions, the size of each dimension and, thereby, the number of array elements it contains.

Data type of an array and an array element

The name of an array identifies the type of data it contains. This same name, when suffixed to become an array element name, implicitly identifies all elements of that array as being of the same data type as the array.

Dimensionality of an array

The number of dimensions of an array is called its *dimensionality* and is equal to the number of dimension declarators in the array declarator.

Size of a dimension

The *size of a dimension* is the non-zero value of its dimension declarator.

Size of an array

The *size of an array* is the number of elements in the array and is equal to the product of the sizes of all dimensions (or to the product of all dimension declarators) for that array.

Technically, a maximum array size corresponding to 4,194,304 CRAY-1 computer words is provided for. Of significance to a practical maximum array size are considerations of memory capacity in the particular CRAY-1 being used, memory required for other than executable program and related data storage purposes, and the size of the executable program itself.

The ANSI FORTRAN Standard does not specify a maximum for array size.

Array element order

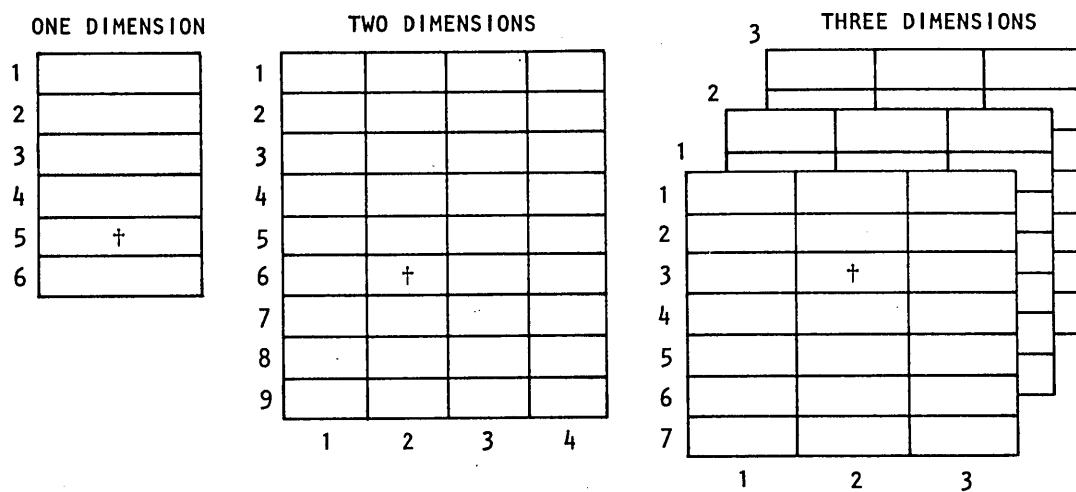
The subscript portion of an array element name has a value that identifies its placement in that array. Subscript values are ordinal and are in one-to-one correspondence with the storage units containing array elements. Thus, a subscript value of one corresponds to the first array element, two corresponds to the second, etc. The last element of the array has a subscript value equal to the size of the array.

An array name used to designate an entire array implies the sequential specification of all subscripts and the processing of all elements in that order.

Array storage sequence

An array has a storage sequence defined by the storage sequence of its elements. The number of storage units (words) in an array is the product of the number of the elements in the array times the number of storage units required for each element.

Examples:



NAME:	VECTOR	TWO DIM	IID
DATA TYPE:	REAL	REAL	INTEGER
DECLARATOR:	VECTOR (6)	TWO DIM (9,4)	IID (7,3,3)
DIMENSIONALITY:	1	2	3
DIMENSION SIZES:	6 ELEMENTS	9 ELEMENTS AND 4 ELEMENTS	7 ELEMENTS, 3 ELEMENTS, AND 3 ELEMENTS
ARRAY SIZE:	6 ELEMENTS	36 ELEMENTS	63 ELEMENTS
† ARRAY ELEMENT NAME:	VECTOR (5)	TWO DIM (6,2)	IID (3,2,1)

3.4.3 ARRAY ELEMENT NAMES

The form of an array element name is

$\alpha (s [,s] \dots)$

where α is the array name,
 $(s [,s] \dots)$ is a subscript, and
 s is a subscript expression.

The number of subscript expressions must equal the number of dimension declarators in the array declarator for that array.

Subscript expression

A *subscript expression* yields a positive integer value when evaluated and may contain references to constants, variables, functions, and array elements of type integer. The evaluation of a subscript expression must not alter the value of other expressions within the same statement.

The ANSI FORTRAN Standard does not provide for specifying integer functions, integer array elements, division or exponentiation in subscript expressions.

Subscript values

The value of a subscript identifies an array element. A subscript value depends on the values of all subscript expressions in the subscript, and on the dimensions of the array as specified in the corresponding array declarator. If the subscript value is k , the k th element of the array is identified. Table 3-3 describes the evaluation of a subscript for k .

See example on page 3-33.

3.4.4 DUMMY AND ACTUAL ARRAYS

A *dummy array* is declared by a dummy array declarator. An *actual array* is declared by an actual array declarator. A dummy array is permitted only in function or subroutine subprograms. Each array in a main program is an actual array specified by a constant array declarator. Actual arrays may also be specified in function and subroutine subprograms.

In a reference to a subprogram containing a dummy array, the actual argument corresponding to the dummy array name must be either an array name or an array element name. If it is an array name, the size of the dummy array must not exceed the size of the actual array. If the actual argument is an array element name with a subscript value of s in an array of size n , the size of the dummy array must not exceed $n-s+1$. Each dummy array must be associated through one or more levels of external procedure references with an actual array or an actual array element.

Table 3-3. Subscript values

Dimen- sionality	Dimension declarator(s)	Subscript	Subscript value (k)	Maximum subscript value
1	A	(a)	a	A
2	A,B	(a,b)	$a+A \cdot (b-1)$	$A \cdot B$
3	A,B,C	(a,b,c)	$a+A \cdot (b-1)$ $+A \cdot B \cdot (c-1)$	$A \cdot B \cdot C$
4	A,B,C,D	(a,b,c,d)	$a+A \cdot (b-1)$ $+A \cdot B \cdot (c-1)$ $+A \cdot B \cdot C \cdot (d-1)$	$A \cdot B \cdot C \cdot D$
5	A,B,C,D,E	(a,b,c,d,e)	$a+A \cdot (b-1)$ $+A \cdot B \cdot (c-1)$ $+A \cdot B \cdot C \cdot (d-1)$ $+A \cdot B \cdot C \cdot D \cdot (e-1)$	$A \cdot B \cdot C \cdot D \cdot E$
6	A,B,C,D,E,F	(a,b,c,d,e,f)	$a+A \cdot (b-1)$ $+A \cdot B \cdot (c-1)$ $+A \cdot B \cdot C \cdot (d-1)$ $+A \cdot B \cdot C \cdot D \cdot (e-1)$ $+A \cdot B \cdot C \cdot D \cdot E \cdot (f-1)$	$A \cdot B \cdot C \cdot D \cdot E \cdot F$
7	A,B,C,D,E,F,G	(a,b,c,d,e,f,g)	$a+A \cdot (b-1)$ $+A \cdot B \cdot (c-1)$ $+A \cdot B \cdot C \cdot (d-1)$ $+A \cdot B \cdot C \cdot D \cdot (e-1)$ $+A \cdot B \cdot C \cdot D \cdot E \cdot (f-1)$ $+A \cdot B \cdot C \cdot D \cdot E \cdot F \cdot (g-1)$	$A \cdot B \cdot C \cdot D \cdot E \cdot F \cdot G$

Notes: a,b,c,d,e,f and g are subscript expressions.

A,B,C,D,E,F and G are dimension sizes.

The ANSI FORTRAN Standard provides for no more than three dimensions.

Example:

ARRAY STORAGE
SEQUENCE

ARRAY ELEMENT NAME = ARRAY NAME + SUBSCRIPT

SUBSCRIPT VALUE	1	X(1,1,1)
2		X(2,1,1)
3		X(3,1,1)
4		X(4,1,1)
5		X(5,1,1)
6		X(1,2,1)
7		X(2,2,1)
8		X(3,2,1)
9		X(4,2,1)
10		X(5,2,1)
11		X(1,3,1)
12		X(2,3,1)
13		X(3,3,1)
14		X(4,3,1)
15		X(5,3,1)
16		X(1,1,2)
17		X(2,1,2)
18		X(3,1,2)
19		X(4,1,2)
20		X(5,1,2)
21		X(1,2,2)
22		X(2,2,2)
23		X(3,2,2)
24		X(4,2,2)
25		X(5,2,2)
26		X(1,3,2)
27		X(2,3,2)
28		X(3,3,2)
29		X(4,3,2)
30		X(5,3,2)

ARRAY CHARACTERISTICS

NAME: X
 DATA TYPE: REAL
 DECLARATOR: X(5,3,2)
 DIMENSIONALITY: 3
 DIMENSION SIZES: 5 ELEMENTS,
 3 ELEMENTS, AND
 2 ELEMENTS
 ARRAY SIZE: 30 ELEMENTS

SUBSCRIPT VALUE FORMULA:

$$\text{SUBSCRIPT VALUE} = a + 5 \cdot (b-1) + 3 \cdot (c-1)$$

WHERE a = FIRST SUBSCRIPT TERM,
 b = SECOND SUBSCRIPT TERM, AND
 c = THIRD SUBSCRIPT TERM.

ARRAY CONCEPTUALIZATION

2		
	X(1,1,2) 16	X(1,2,2) 21
	X(2,1,2) 17	X(2,2,2) 22
	X(3,1,2) 18	X(3,2,2) 23
	X(4,1,2) 19	X(4,2,2) 24
1	X(5,1,2) 20	X(5,2,2) 25
1	X(1,1,1) 1	X(1,2,1) 6
2	X(2,1,1) 2	X(2,2,1) 7
3	X(3,1,1) 3	X(3,2,1) 8
4	X(4,1,1) 4	X(4,2,1) 9
5	X(5,1,1) 5	X(5,2,1) 10
		X(5,3,1) 15
1		
2		
3		

1 2 3

ARRAY ELEMENT:

[ELEMENT NAME]

+SUBSCRIPT VALUE

3.4.5 ADJUSTABLE ARRAYS AND ADJUSTABLE DIMENSIONS

An *adjustable array* is declared by an adjustable array declarator in which dimension declarators may be specified as variables. Such are called *adjustable dimensions*. The name of an adjustable array must appear in the dummy argument list of a subprogram. Its adjustable array declarators must be dummy array declarators. A variable used as a dimension declarator must be named in the dummy argument list containing the array name or in a COMMON statement in the same subprogram.

The ANSI FORTRAN Standard does not provide for a dimension declarator variable being named in a COMMON statement.

Each actual argument corresponding to a dummy argument and each variable in common must be defined with an integer value before being used. The values of those dummy arguments or variables in common and any constants appearing in the dummy array declarator determine the size of the corresponding adjustable dimension for that execution of the subprogram. The sizes of the adjustable dimensions and of any constant dimensions appearing in an adjustable array declarator determine the number and order of elements in the array. Each reference to a subprogram may define different properties (size of dimensions, number of elements, element ordering) for each adjustable array in that subprogram. These properties depend on the values of any actual arguments and variables in common when the subprogram is referenced.

Adjustable array properties of dimension number and array size do not change during subprogram execution. Variables defining an adjustable dimension may be redefined or become undefined during execution of the subprogram with no effect on these properties.

3.4.6 USE OF ARRAY NAMES

In a program unit, each appearance of an array name must be as part of an array element name except:

- In a list of dummy arguments,
- In a COMMON statement,
- In a type-statement,
- In an array declarator,
- In an EQUIVALENCE statement,
- In a DATA statement,
- In the list of actual arguments in a reference to an external procedure,
- In the list of an input/output statement, and
- As the format identifier in an input/output statement.

3.5 EXPRESSIONS

An expression calls for the evaluation of one or more operands and may include operators and parentheses to specify the manner and order of their concatenation in yielding a single value. Operands are constants, symbolic names of constants, variables, array elements and function references. The type of an operand is implicit to or defined for its name or form. Operators specify the arithmetic, relational, or logical operations to be performed on these operands. Their sequence of presentation among operands combines with the use of parentheses to specify the desired manner and order of evaluation.

The ANSI FORTRAN Standard does not provide for symbolic names of constants.

Expressions are of five types:

- Arithmetic,
- Hollerith,
- Relational,
- Logical, and
- Boolean.

3.5.1 ARITHMETIC EXPRESSIONS

An arithmetic expression specifies a numeric computation. Its evaluation produces a single numeric value.

The simplest form of arithmetic expression is an unsigned constant or the symbolic name of a constant, variable, array element, or function. More complicated arithmetic expressions are formed by using one or more arithmetic operands with arithmetic operators and parentheses. Arithmetic operands must be of type integer, real, double precision, or complex.

Arithmetic operators

The arithmetic operators are given in table 3-4.

Table 3-4. Arithmetic operators

Operator	Operation
**	Exponentiation
/	Division
*	Multiplication
-	Subtraction or negation
+	Addition or identity

Each arithmetic operator can operate on a pair of operands and is written between them. Additionally, either of the operators + and - can operate on a single operand when it precedes that operand.

Form and interpretation of arithmetic expressions

The interpretation of expressions formed with each arithmetic operator is shown in table 3-5. (X and Y are operands.)

Table 3-5. Interpretation of operators in expressions

Use of operator	Interpretation
X**Y	Exponentiate X to the power Y
X/Y	Divide X by Y
X*Y	Multiply X by Y
X-Y	Subtract Y from X
-Y	Negate Y
X+Y	Add X to Y
+Y	(Same as Y)

The interpretation of a division may depend on the data types of the operands.

A precedence among arithmetic operators determines the order in which operands are to be combined (unless changed by the use of parentheses). This precedence is shown in table 3-6.

Table 3-6. Precedence of arithmetic operators

Operator	Precedence
**	Highest
* and /	Intermediate
+ and -	Lowest

For example, in the expression

$-A^{**} 2$

the exponentiation operator (**) has precedence over the negation operator (-). Therefore, the operands of the exponentiation operator are combined, then used as the operand of the negation operator. Thus, the interpretation of the above expression is the same as the mathematical interpretation of the expression

$-(A^{**} 2)$.

Arithmetic operands are:

- Primaries
- Factors,
- Terms, and
- Arithmetic expressions.

The following describes the forms of combining operands and operators in arithmetic expressions.

Primaries are:

- Unsigned arithmetic constants,
- Symbolic names of arithmetic constants,
- Variable references,
- Array element references,
- Function references, and
- Arithmetic expressions enclosed in parentheses.

Examples:

<u>PRIMARY</u>	<u>DESCRIPTION</u>
23D9	Unsigned double-precision constant
KVALUE	Integer constant name if named in a PARAMETER statement
COUNTER8	Real variable name
IMAG(3,52,75)	Complex array element name if declared in a COMPLEX statement
EVAL(A,B,C)	Real function name if declared in a FUNCTION or statement function statement
(arithmetic expression)	Parenthesized arithmetic expression

The forms of a *factor* are:

- primary
- primary ** factor

Thus, a factor is a sequence of one or more primaries with its elements separated by the exponentiation operator. The second form indicates that in interpreting a factor containing two or more exponentiation operators, the primaries must be combined from right to left. For example, the factor

2**3**2

has the same interpretation as the factor

2**(3**2).

The forms of a *term* are:

- factor
- term / factor
- term * factor

Thus a term is a factor or a sequence of factors with its elements separated by a multiplication or a division operator. The last two forms indicate that the factors are combined from left to right in interpreting a term containing two or more multiplication or division operators.

The forms of an *arithmetic expression* are:

term
+ term
- term
arithmetic expression + term
arithmetic expression - term

Thus an arithmetic expression is a term or a sequence of terms with its elements separated by an addition (+) or a subtraction (-) operator. The first term in an arithmetic expression may be preceded by an identity (+) or negation (-) operator. The last two forms imply that terms are combined from left to right in interpreting an arithmetic expression containing two or more addition or subtraction operators.

These formation rules do not permit expressions containing two consecutive arithmetic operators such as $A^{**}-B$ or $A+-B$. However, expressions such as $A^{**}(-B)$ and $A+(-B)$ are permitted.

Examples:

<u>PRIMARIES</u>	<u>FACTORS</u>	<u>TERMS</u>	<u>ARITHMETIC EXPRESSIONS</u>
5	5	5	5 +5 -5
CON	CON	CON	CON +CON -CON 5+CON CON-5
CON**5	CON**5		CON**5 +CON**5 -CON**5 5+CON-CON**5 -CON**5+5
TBL(7,10)	TBL(7,10)	TBL(7,10)	TBL(7,10) +TBL(7,10) -TBL(7,10)
	CON**5/TBL(7,10)	CON**5/TBL(7,10)	+CON**5/TBL(7,10) -CON**5/TBL(7,10)
	CON**5*TBL(7,10)	CON**5*TBL(7,10)	+CON**5*TBL(7,10) -CON**5*TBL(7,10) CON**5/TBL(7,10)+CON -CON**5*TBL(7,10)-TBL(7,10)

Type and interpretation of arithmetic expressions

The form of a constant determines its data type. The data type of a named constant, variable, array element, or function reference is determined by its name. The data type of an arithmetic expression containing one or more arithmetic operators is determined from the data types of the operands.

Integer expressions, real expressions, double-precision expressions, and complex expressions are arithmetic expressions having values of type integer, real, double precision, and complex, respectively.

When a + or - operates on a single operand, the data type of the resulting expression is the same as the data type of the operand.

The data types of arithmetic expressions are given in table 3-7. In this table, each letter designates the type of operand or result as integer (I), real (R), double precision (D), or complex (C).

To use the table, locate the types of the first and second operands in the first and second columns, respectively. The third column contains the type of the expression formed when these operands are processed by an arithmetic operator or when there is only one operand in the expression. The remaining columns of the table relate to the possible re-typing of the value of these expressions upon assignment to variables and array elements of different types.

For example, using the integer variable I with the double-precision array element D(1,2,3) in the expression

D(1,2,3) * I

yields a result of type double precision as specified in the ninth row of table 3-7.

Except for a value raised to an integer power, table 3-7 specifies that if two operands are of different types, the one differing in type from the prescribed result is first converted to the type of the result, then the concatenation called for by the operator is effected. When a primary of real, double-precision, or complex type is raised to an integer power, the integer operand is not converted.

The ANSI FORTRAN Standard relationships are a subset of those illustrated in table 3-7.

Table 3-7. Arithmetic operand, expression, and result typing relationships

			INTEGER	REAL	DOUBLE PRECISION	COMPLEX
I	I	I	(I)	(I)	(I)	(I)
I	R	R	(R)	R	(R)	(R)
I	D	D	(D)	(D)	D	(D)
I	C	C	(C)	(C)	(C)	C
R	I	R	(R)	R	(R)	(R)
R	R	R	(R)	R	(R)	(R)
R	D	D	(D)	(D)	D	(D)
R	C	C	(C)	(C)	(C)	C
D	I	D	(D)	(D)	D	(D)
D	R	D	(D)	(D)	D	(D)
D	D	D	(D)	(D)	D	(D)
D	C	†	†	†	†	†
C	I	C	(C)	(C)	(C)	C
C	R	C	(C)	(C)	(C)	C
C	D	†	†	†	†	†
C	C	C	(C)	(C)	(C)	C

Legend -

x,y

Arithmetic operands

⊕

Arithmetic operator

Arithmetic expression or single arithmetic operand

z

Arithmetic result

()

Conversion required before assignment to result.

†

No conversion required; operands and result agree in type.

Prohibited

The type and interpretation of an expression that consists of an operator operating on either a single operand or a pair of operands are independent of the context in which the expression appears. In particular, the type and interpretation of such an expression are independent of the type of any other operand of any larger expression in which it appears.

Integer quotients

An *integer quotient* is the integer portion of a mathematical quotient having an integral divisor and dividend. For example, the expression $-5/2$ yields an integer quotient of -2 .

Type conversion

Type conversion of operands may occur during an expression's evaluation or when the results of an expression's evaluation are to be stored into a variable or array element. Type conversion may be based upon the following two operations:

- (a) Integer to real conversion creates a real value from an integer value. The maximum absolute value of the integer must be less than 2^{46} . There is no warning if the value exceeds this range.
- (b) Real to integer conversion creates a 64-bit integer value from a real value. The maximum absolute value of the number being converted must be less than 2^{46} . The fractional part is truncated. There is no warning if the value exceeds the range.

Zero values are of identical form in both types of data and do not require conversion.

Type integer - Type integer conversion to:

- Type real occurs as described in (a) above.
- Type double precision occurs as described in (a) above but with 48 binary zeros added to extend the precision.
- Type complex occurs as described in (a) above. The result becomes the real portion of the complex value and zero is established as the imaginary portion.

Type real - Type real conversion to:

- Type integer occurs as described in (b) above.
- Type double precision is accomplished by extending the precision of the real value through the addition of 48 binary zeros.
- Type complex is accomplished by establishing the real value as the real portion of the complex value and by establishing zero in the imaginary portion.

Type double precision - Type double precision conversion to:

- Type integer creates a 64-bit integer value of less than 2^{63} from the integral portion of the double precision value. Any fractional portion is lost and no rounding occurs.

- Type real is accomplished by eliminating the least significant 48 bits of the double-precision value and establishing the result as the real value. No rounding occurs.
- Type complex is accomplished by eliminating the least significant 48 bits of the double-precision value, establishing the result as the real portion of the complex value, and establishing zero as the imaginary portion of the complex value. No rounding occurs.

Type complex - Type complex conversion to:

- Type integer is accomplished by converting the real portion of the complex value as described for the real value in (b) above.
- Type real is accomplished by establishing the real portion of the complex value as the real value.
- Type double precision is accomplished by extending the precision of the real portion of the complex value through the addition of 48 binary zeros.

3.5.2 HOLLERITH EXPRESSIONS

Hollerith expressions contain no operators and only a single operand. A Hollerith expression is evaluated to yield a sequence of characters. Its value is that sequence. The forms of a Hollerith expression appear below:

A Hollerith constant

The name of a variable containing a Hollerith datum

The name of an array element containing a Hollerith datum

The name of a function that provides a Hollerith datum when referenced

A Hollerith constant comprising a Hollerith expression is limited to eight characters.

The data type of the name referencing a variable or array element containing a Hollerith datum may affect its evaluation during program execution. A variable or array element of type integer or real contains eight Hollerith characters. A variable or array element of type double precision or complex contains eight characters in its first storage unit (computer word) and may contain the value zero or an additional eight characters in its second. A variable or array element of type logical cannot contain Hollerith characters.

A Hollerith datum provided when a function is referenced contains as many characters as a variable or array element of corresponding type.

3.5.3 RELATIONAL EXPRESSIONS

A relational expression compares the values of two arithmetic or Hollerith expressions, producing a result of type logical with a value of true or false.

Relational expressions can appear within logical expressions.

Relational operators

The relational operators are shown in table 3-8.

Table 3-8. Relational operators

Operator	Operation (comparison)
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GE.	Greater than or equal to
.GT.	Greater than

Arithmetic relational expressions

The form of an *arithmetic relational expression* is

$e_1 \text{ relop } e_2$

where e_1 and e_2 are each integer, real, double-precision, or complex expressions, and

relop is a relational operator.

A complex expression is permitted only when the relational operator is .EQ. or .NE.

The ANSI FORTRAN Standard does not provide for relational expressions containing complex expressions.

An arithmetic relational expression is interpreted as the logical value true if the values of the expressions satisfy the relation specified by the operator; false if they do not.

If the two arithmetic expressions are of different types, the types of the operands are converted as if the expression were

$$((e_1)-(e_2)) \text{ relop } 0$$

(See table 3-7.)

Examples:

```
INDEX.EQ.ENDVALU
J(1,6,6)*COS(ALPHA/10.).GT.Z
A.LE.B
3.1415927.LT.(22./7.)
CMPLXM.NE.COMPLXN
```

Hollerith relational expressions

The form of a Hollerith relational expression is

$$e_1 \text{ relop } e_2$$

where e_1 and e_2 are Hollerith expressions, and
 relop is a relational operator.

A Hollerith relational expression is interpreted as the logical value true if the values of the operands satisfy the relation specified by the operator; false if they do not.

The Hollerith expression e_1 is considered less than e_2 if its value precedes the value of e_2 in the collating sequence or is considered greater if its value follows the value of e_2 in the collating sequence. If the operands are of unequal length, the shorter operand is considered as if it were extended on the right with blanks to the length of the longer operand.

Examples:

The following are evaluated as true if the integer variable LOCK contains the Hollerith characters K, E, and Y in that order and left justified with five trailing blank character codes:

```
3HKEY.EQ.LOCK
'KEY'.EQ.LOCK
LOCK.EQ.LOCK
'KEY1'.GT.LOCK
'KEY0'H.GT.LOCK
```

3.5.4 LOGICAL EXPRESSIONS

A logical expression expresses a logical computation. Evaluation of a logical expression produces a result of type logical with a value of true or false.

The simplest form of a logical expression is the specification of a:

- Logical constant,
- Logical constant name,
- Logical variable reference,
- Logical array element reference,
- Logical function reference, or
- Relational expression.

More complicated logical expressions are formed by using one or more logical operands with logical operators and parentheses.

Logical operators

Table 3-9 presents the *logical operators* and their order of precedence.

Table 3-9. Logical operators

Operator	Operation	Precedence
.NOT. or .N.	Logical negation	Highest
.AND. or .A.	Logical conjunction	Higher
.OR. or .O.	Logical inclusive disjunction	Lower
.XOR. or .X. -- .NEQV. -- .EQV.	Logical exclusive disjunction or Logical non-equivalence Logical equivalence	Lowest

The ANSI FORTRAN Standard does not provide for .XOR., .EQV., .NEQV., .N., .A., .O., or .X..

Note that the logical operators .XOR., .X., and .NEQV. perform the same logical operation.

Form and interpretation of logical expressions

A set of formation rules establishes the interpretation of a logical expression containing two or more logical operators. The precedence among the logical operators determines the order in which they are to be combined (unless changed by the use of parentheses). For example, in the expression

A .OR. B .AND. C

the .AND. operator has higher precedence than the .OR. operator. Therefore, the interpretation is the same as the interpretation of

A .OR. (B .AND. C)

Logical operands are:

- Logical primaries,
- Logical factors,
- Logical terms,
- Logical disjuncts, and
- Logical expressions.

The following paragraphs describe the forms of combining operands and operators in logical expressions.

Logical primaries are:

- Logical constants,
- Symbolic names of logical constants,
- Logical variable references,
- Logical array element references,
- Logical function references,
- Relational expressions, and
- Logical expressions enclosed in parentheses.

The forms of a *logical factor* are:

- [.NOT.] logical primary
- [.N.] logical primary

The forms of a *logical term* are:

- [logical term .AND.] logical factor
- [logical term .A.] logical factor

Thus, a logical term is a sequence of logical factors separated by an .AND. or .A. operator. The form indicates that in interpreting a logical term containing two or more .AND. or .A. operators, the logical factors are combined from left to right.

The forms of a *logical disjunct* are:

- [logical disjunct .OR.] logical term
- [logical disjunct .O.] logical term

A logical disjunct is, therefore, a sequence of logical terms separated by an .OR. or .O. operator. The form indicates that in interpreting a logical disjunct containing two or more .OR. or .O. operators, the logical terms are combined from left to right.

The forms of a *logical expression* are:

[logical expression .XOR.]	logical disjunct
[logical expression .X.]	logical disjunct
[logical expression .EQV.]	logical disjunct
[logical expression .NEQV.]	logical disjunct

A logical expression is, therefore, a sequence of logical disjuncts separated by .XOR., .X., .EQV., or .NEQV. operators. These forms indicate that in interpreting a logical expression containing two or more .XOR., .X., .EQV., and/or .NEQV. operators, the logical disjuncts are combined from left to right.

Note that these forms allow the logical operator .NOT. to follow immediately after any other logical operator. For example, the logical term

LOGICALX .AND. .NOT. LOGICALY

is permitted.

Values of logical factors, terms, disjuncts, and expressions

The value of a logical factor involving .NOT. or .N. is shown below:

X	.NOT.X .N. X
true	false
false	true

The value of a logical term involving .AND. or .A. is shown below:

X_1	X_2	$X_1 \text{ .AND. } X_2$ $X_1 \text{ .A. } X_2$
true	true	true
true	false	false
false	true	false
false	false	false

The value of a logical disjunct involving .OR. or .O. is shown below:

X_1	X_2	$X_1 \text{ .OR. } X_2$ $X_1 \text{ .O. } X_2$
true	true	true
true	false	true
false	true	true
false	false	false

The values of logical expressions involving .XOR., .X., .EQV., and .NEQV. are shown below:

X_1	X_2	$X_1 \text{ .XOR. } X_2$ $X_1 \text{ .X. } X_2$	$X_1 \text{ .EQV. } X_2$	$X_1 \text{ .NEQV. } X_2$
true	true	false	true	false
true	false	true	false	true
false	true	true	false	true
false	false	false	true	false

Note that the logical operators .XOR., .X., and .NEQV. perform identical logical operations.

3.5.5 BOOLEAN EXPRESSIONS

The ANSI FORTRAN Standard does not provide for Boolean expressions.

Boolean expressions contain no operators and only a single operand. A Boolean expression is evaluated to yield a string of 64 binary digits representative of bit positions in a storage unit (64-bit CRAY-1 computer word). The forms of a Boolean expression appear below:

A Boolean constant

The name of a variable containing a Boolean datum

The name of an array element containing a Boolean datum

The name of a function that provides a Boolean datum when referenced

Boolean expressions may be combined with expressions of other types by using arithmetic, relational, and logical operators. A Boolean datum is never converted to a different type. Evaluation of an arithmetic or relational operator processes a Boolean expression with no type conversion, but as though it were of integer type.

A logical operator processing a Boolean expression performs a bit-by-bit (masking) operation. The result of such operations may be of type integer, real, logical or Boolean depending on other operands in the expression. The values of logical factors, terms and expressions described in 3.5.4 apply, but with results of binary one and zero corresponding to the logical results of true and false, respectively, and in each of 64 bit positions. These values are summarized in the chart below:

X ₁	X ₂	.NOT.X ₁ .N.X ₁	X ₁ .AND.X ₂ X ₁ .A.X ₂	X ₁ .OR.X ₂ X ₁ .O.X ₂	X ₁ .XOR.X ₂ X ₁ .X.X ₂	X ₁ .EQV.X ₂	X ₁ .NEQV.X ₂
1100	1010	0011	1000	1110	0110	1001	0110

3.5.6 PRECEDENCE OF ALL OPERATORS

The foregoing has established precedences within groups of arithmetic and logical operators. Relational operators have no precedence within their own group. Precedences among all types of operators are presented in table 3-10.

Table 3-10. Precedence among all operators

Operator	Precedence
Arithmetic	Highest
Relational	Intermediate
Logical	Lowest

An expression may contain more than one kind of operator. For example, the logical expression

L .OR. A + B .GE. C

where A, B, and C are of type real and L is type logical, contains an arithmetic operator, a relational operator, and a logical operator. This expression would be interpreted the same as the expression

L .OR. ((A + B) .GE. C)

3.5.7 SUMMARY OF RULES OF INTERPRETATION

The order in which primaries are combined using operators is determined by the:

- Use of parentheses,
- Precedence of operators,
- Right-to-left interpretation of exponentiations in a factor,
- Left-to-right interpretation of multiplications and divisions in a term,
- Left-to-right interpretation of additions and subtractions in an arithmetic expression,
- Left-to-right interpretation of conjunctions in a logical or Boolean term,
- Left-to-right interpretation of inclusive disjunctions in a logical or Boolean disjunct, and
- Left-to-right interpretation of exclusive disjunctions, equivalences and non-equivalences in a logical or Boolean expression.

3.5.8 EVALUATION OF EXPRESSIONS

Any variable, array element, or function referenced as an operand in an expression must be defined at the time the reference is executed. Any names of constants must have been established in a PARAMETER statement preceding the statement of first reference.

Any arithmetic operation whose result cannot be defined mathematically produces unpredictable results in an executable program. Examples are dividing by zero, raising zero to a zero or negative power, and raising a negative value to a non-integer power.

The execution of a function reference in a statement must not alter the value of any other entity within the same statement. Nor may it alter the value of any entity in common that affects the value of any other function reference in that same statement. If a function reference in a statement causes definition of an actual argument of the function, that argument or any associated entities must not appear elsewhere in the same statement. For example, in the statements

$$A(I) = F(I) \text{ and}$$

$$Y = G(X) + X$$

where F and G are functions, unpredictable results occur when the reference to F defines I or the reference to G defines X .

The data type of an expression in which a function reference appears neither affects nor is affected by the evaluation of the actual arguments of the function.

The data type of an expression in which an array element is referenced neither affects nor is affected by the evaluation of the subscript

There are certain conditions under which evaluation of an expression may differ from evaluation according to the interpretation rules stated thus far. These are described in the following paragraphs.

Order of evaluation of functions

The order of evaluation of multiple function references within a single statement is fixed only within a direct logical IF statement and within nested function references.

Examples:

- In the statement $IF(F(Y))A=F(Y)$ where F is a function name, the function reference in the conditional statement $A=F(Y)$ is evaluated last.
- In the statement $A=F(G(X))$ where F and G are functions, G is evaluated first.

In other statements that contain more than one function reference, the value provided by each function reference may be unpredictably affected by the order in which the other function references are evaluated.

Example:

In the statement $A=F(G(H(X)),E(Y))$ where E, F, G, and H are functions H is evaluated before G, and E and G (and H) are evaluated before F, but H is not necessarily evaluated before E.

Parentheses and expressions

Any parenthesized expression is treated as an entity. For example, in evaluating the expression $A*(B*C)$, the product of B and C is evaluated and then multiplied by A. Parenthesized expressions may contain one or more parenthesized expressions, each of which may contain one or more parenthesized expressions, etc. This nesting of parenthesized expressions can be specified to 63 levels.

The ANSI FORTRAN Standard does not limit the number of levels of nested parentheses.

Evaluation of arithmetic expressions

The rules for the form and interpretation of arithmetic expressions (3.5.1) describe the interpretation of an arithmetic expression.

Two arithmetic expressions are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal. However, mathematically equivalent arithmetic expressions may produce different computational results.

The mathematical definition of integer division is described in 3.5.1. The difference between the value of the expression $5/2$ and $5./2.$ is mathematical and is not a computational difference. The difference between $5./10.$ and $5.*.1$ is computational.

In addition to parentheses required for the desired interpretation, other parentheses may be included to control the magnitude and accuracy of intermediate values developed during the evaluation of an expression. For example, in the expression

$$A+(B-C)$$

the term $(B-C)$ is evaluated and then added to A. Including parentheses could change the mathematical value. For example, the two expressions:

$$A*I/J$$

$$A*(I/J)$$

may have different mathematical values if I and J are integer factors and if A is real.

Evaluation of Hollerith expressions

The rules of 3.5.2 describe the interpretation of Hollerith expressions. Two Hollerith expressions are equivalent if their values are equal for all possible values of their specification.

Evaluation of relational expressions

The rules of 3.5.3 describe the interpretation of relational expressions. Two relational expressions are relationally equivalent if their logical values are equal for all possible values of their primaries.

Evaluation of logical expressions

The rules of 3.5.4 describe the interpretation of a logical expression. Two logical expressions are logically equivalent if their values are equal for all possible values of their primaries.

Evaluation of Boolean expressions

The rules of 3.5.5 and 3.5.4 describe the interpretation of a Boolean expression. Two Boolean expressions are equivalent if their values are equal for all possible values of their primaries.

3.6 STATEMENTS

3.6.1 STATEMENT CLASSIFICATION

Every statement is classified as either executable or non-executable.

An executable statement specifies an action. When compiled with other executable statements, an execution sequence is produced.

A non-executable statement:

- Contains input/output editing information,
- Specifies statement functions,
- Classifies program units, or
- Specifies the characteristics, arrangement, and initial values of data.

A non-executable statement forms no execution sequence. Statement labels associated with non-executable statements must not be referenced to control the execution sequence.

Statements are recognized by keyword or form as executable or non-executable (see table 3-11).

Any statement of either classification may be identified with a statement label which must be unique among all statement labels used in that program unit..

3.6.2 DATA SPECIFICATION

The kinds of specification statements are:

- DIMENSION;
- EQUIVALENCE;
- COMMON;
- INTEGER, REAL, DOUBLE PRECISION, DOUBLE, COMPLEX, and LOGICAL type statements;
- IMPLICIT;
- PARAMETER; and
- EXTERNAL.

All are non-executable.

The ANSI FORTRAN Standard does not provide for IMPLICIT, PARAMETER, or DOUBLE statements.

Table 3-11. Statement classification

<u>Executable statements</u>	<u>Non-executable statements</u>
Arithmetic assignment	DIMENSION
Logical assignment	EQUIVALENCE
ASSIGN	COMMON
Unconditional GO TO	INTEGER type
Computed GO TO	REAL type
Assigned GO TO	DOUBLE PRECISION type
Three-branch arithmetic IF	DOUBLE (precision) type
Two-branch arithmetic IF	COMPLEX type
Direct logical IF	LOGICAL type
Indirect logical IF	IMPLICIT
DO	PARAMETER
CONTINUE	EXTERNAL
STOP	DATA
PAUSE	FORMAT
Formatted READ	PROGRAM
Unformatted READ	Statement function definition
Formatted WRITE	FUNCTION
Unformatted WRITE	INTEGER FUNCTION
PRINT	REAL FUNCTION
PUNCH	DOUBLE PRECISION FUNCTION
BACKSPACE	DOUBLE FUNCTION
ENDFILE	COMPLEX FUNCTION
REWIND	LOGICAL FUNCTION
BUFFER IN	SUBROUTINE
BUFFER OUT	ENTRY
ENCODE	BLOCK DATA
DECODE	
CALL	
RETURN	
END	

The ANSI FORTRAN Standard does not provide for the following statements:

- PRINT • DOUBLE • PROGRAM • DOUBLE FUNCTION • ENTRY
- PUNCH • IMPLICIT • ENCODE • BUFFER IN • Two -branch arithmetic IF
- END • PARAMETER • DECODE • BUFFER OUT • Indirect logical IF

The three-branch arithmetic IF statement is the arithmetic IF statement in the ANSI FORTRAN Standard. The direct logical IF statement is the logical IF statement in the ANSI FORTRAN Standard.

DIMENSION statements

DIMENSION statements specify the symbolic names and dimension specifications of arrays.

The form of a DIMENSION statement is

DIMENSION $\alpha(d)$ [, $\alpha(d)$] ...

where each $\alpha(d)$ is an array declarator (3.4.1).

Each symbolic name α appearing in a DIMENSION statement declares α to be an array in that program unit. An array name can appear only once as an array declarator in a program unit. Array declarators may also appear in COMMON statements and in type statements.

Examples:

```
DIMENSION ARRAY (34,24,34), VECTOR (64), Z7144X (10,20)
DIMENSION MATRIX (ROWS,COLUMNS)
DIMENSION TABLE (3,IVAL, MATRIX,2,2)
```

Note that in the last two examples, the use of variables defines adjustable dimensioning as permitted only in procedure subprograms.

EQUIVALENCE statements

An EQUIVALENCE statement specifies the sharing of one or more storage units by two or more entities in a single program unit. This causes the association of those entities.

If associated entities are of different data types, the EQUIVALENCE statement does not cause type conversion or imply mathematical equivalence. If a variable and an array are associated, the variable does not assume the properties of an array and the array does not assume the properties of a variable.

Form of an EQUIVALENCE statement - The form of an EQUIVALENCE statement is

EQUIVALENCE ($nlist$) [,($nlist$)] ...

where $nlist$ is a list of two or more variable names, array element names, and/or array names separated by commas.

Names of dummy arguments of a subprogram cannot appear in $nlist$. A variable name that is also a function name cannot appear in $nlist$.

Each subscript expression in $nlist$ must be an integer constant expression.

Equivalence association - An EQUIVALENCE statement specifies that the storage sequence of each entity in a list *nlist* shares the same first storage unit. This causes the association of all entities in the list and may also cause indirect association of other entities.

Array names and array element names - If an array element name appears in an EQUIVALENCE statement, the number of subscript expressions must equal the number of dimensions in the array declarator for the array.

The use of an array name in an EQUIVALENCE statement has the same effect as using the name of the first array element.

Restrictions on EQUIVALENCE statements - An EQUIVALENCE statement must not specify that the same storage unit is to occur more than once in a storage sequence. For example,

```
DIMENSION A(2)  
EQUIVALENCE (A(1),B), (A(2),B)
```

is prohibited, because it would specify the same storage unit for A(1) and A(2).

An EQUIVALENCE statement must not specify that consecutive storage units are to be non-consecutive. For example, the following is prohibited:

```
REAL A(2)  
DOUBLE PRECISION D(2)  
EQUIVALENCE (A(1),D(1)),(A(2),D(2))
```

An EQUIVALENCE statement must not associate the storage sequences of two different common blocks in the same program unit. EQUIVALENCE statement association must not cause a common block storage sequence to be extended by adding storage units preceding the first storage unit of the first entity specified in a COMMON statement for the common block. For example,

```
COMMON /X/A  
REAL B(2)  
EQUIVALENCE (A,B(2))
```

is not permitted since it would attempt association of array element B(1) with a storage unit preceding A in common block X.

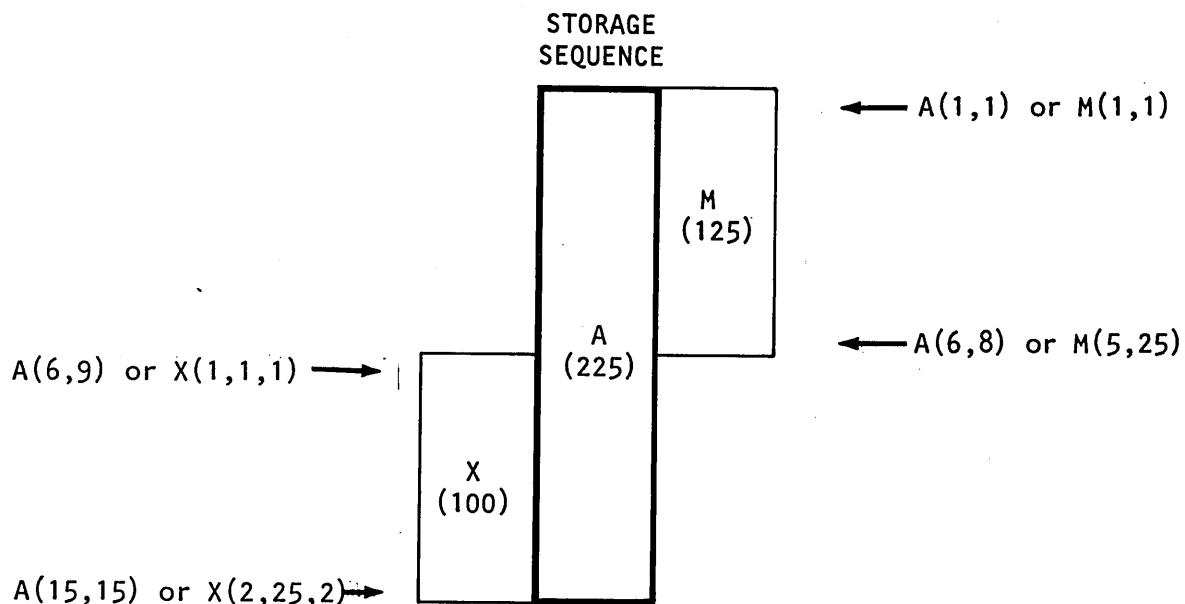
Example:

The appearance of

DIMENSION A(15,15), M(5,25), X(2,25,2)

EQUIVALENCE (A,M), (A(6,9),X)

in a program unit causes the first 125 array elements of A to coincide with the 125 array elements of M, and the last 100 array elements of A to coincide with the 100 array elements of X.



COMMON statements

The COMMON statement associates entities in different program units. This allows different program units to share storage units and to define and reference the same data.

Form of a COMMON statement - The form of a COMMON statement is

COMMON [/ [cb] /] nlist [/ [cb] /] nlist ...

where *cb* is the symbolic name of a common block, and

nlist is a list of variable names, array names, and array declarators separated by commas. Names of dummy arguments of a subprogram cannot appear in the list.

In each COMMON statement, the entities occurring in *nlist* following a block name *cb* are declared to be in common block *cb*. The blank (unnamed) common block is specified when a *cb* does not appear between slashes. If the first *cb* is omitted, its enclosing slashes are optional and all entities in the *nlist* are specified to be in blank common.

Any *cb* (or an omitted *cb* for blank common) may occur more than once in one or more COMMON statements in a program unit. The *nlist* following each successive appearance of the same common block name continues the preceding list for that common block name.

Common block storage sequence - For each common block, a *common block storage sequence* is formed as follows:

- A storage sequence is formed consisting of the storage sequences of all entities in the lists *nlist* for the common block. The order of the storage sequence is determined by the order of the appearance of the lists *nlist* in the program unit.
- This storage sequence is extended to include all storage units of any storage sequence associated with it by EQUIVALENCE statement association. The sequence may be extended only by adding storage units beyond the last storage unit. Entities associated with an entity in a common block are considered to be in that common block.

Size of a common block - The size of a common block is the size of its common block storage sequence, including any extensions of the sequence resulting from EQUIVALENCE statement association.

Within an executable program, the size of a named common block is established during compilation of the first program unit specifying its name. This size cannot be exceeded in specifying the same named common block in subsequent program units, but may be the same or less. Blank common blocks within an executable program are not required to be of the same size and may increase, decrease or remain the same as each program unit is compiled.

Common association - Within an executable program, the common block storage sequences of all common blocks with the same name share the same first storage unit. The same is true of all blank common blocks. This results in the association of entities among different program units.

Differences between named common and blank common - A blank common block has the same properties as a named common block, except that only entities in named common blocks may be initially defined by DATA statements.

Restrictions on COMMON and EQUIVALENCE statements - An EQUIVALENCE statement must not associate the storage sequences of two different common blocks in the same program unit. EQUIVALENCE statement association must not attempt to extend common block storage sequence by adding storage units preceding the first storage unit of the first entity specified in a COMMON statement for the common block.

Example:

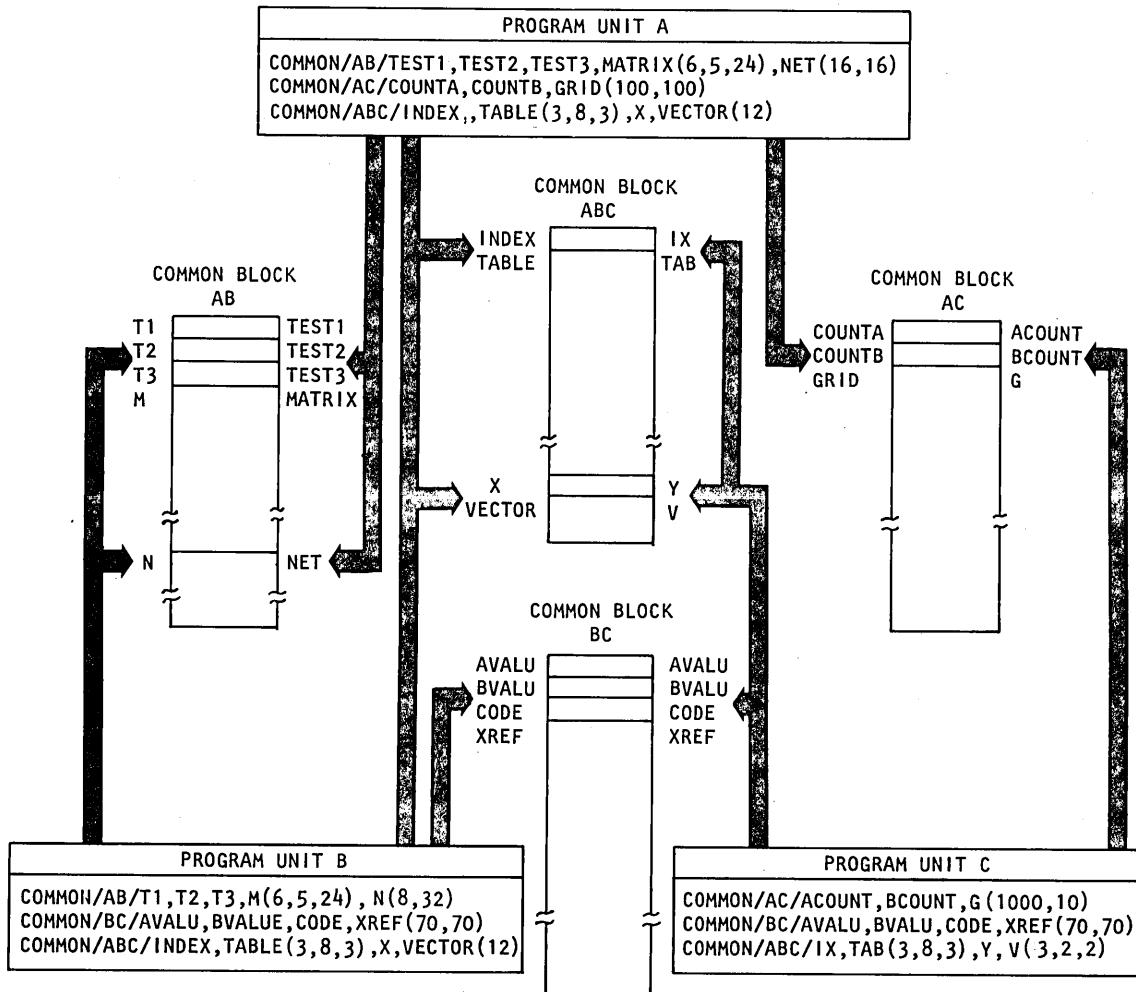
Two program units in the same executable program each contain the following statements:

```
COMMON /C1/X,Y,Z(10,10)  
DIMENSION C1(1,2,3)
```

Common block C1 contains the variables X and Y and 100-element array Z. Since all were declared in identical COMMON statements appearing in both program units, each program unit may access the same data. Additionally, identical DIMENSION statements appearing in both program units declare two identical but completely independent six-element arrays, C1, each accessible only to the program unit in which it was so declared. Note that the common block containing X, Y, and Z has the name C1 and that this name in no way conflicts with or pertains to the array name C1 specified in the DIMENSION statement.

Example:

The following illustrates a more complex example of COMMON statement use in which three program units in the same executable program share four common blocks:



Type statements

A type statement either overrides or confirms implicit typing and may specify dimension information.

The appearance of the symbolic name of a constant, variable, array, or function in a type statement specifies the data type for all appearances of that name in the program unit. Within a program unit, a name must not have its type explicitly specified more than once.

Subroutine names, main program names, and block data subprogram names must not appear in a type statement.

If a specific intrinsic function name appearing in a type statement conflicts with that function's type as specified in Appendix B, the name loses its intrinsic function property in the program unit containing that type statement.

The forms of type statements are:

INTEGER	$v[,v]...$
REAL	$v[,v]...$
DOUBLE PRECISION	$v[,v]...$
DOUBLE	$v[,v]...$
COMPLEX	$v[,v]...$
LOGICAL	$v[,v]...$

where INTEGER, REAL, DOUBLE PRECISION (or DOUBLE), COMPLEX,
 or LOGICAL specifies the desired data type, and

v is a constant, variable, array, function, or dummy
 procedure name, or is an array declarator.

Either DOUBLE PRECISION or DOUBLE causes double-precision typing. Note that the space between DOUBLE and PRECISION is optional.

The ANSI FORTRAN Standard does not permit the use of DOUBLE instead of DOUBLE PRECISION.

(A special form of integer typing is discussed in Section 5 under "Compiler directives".)

Examples:

```
INTEGER NPAK(60,230),RTEST,XREF(20,2),ARRAY
REAL FSTOP,PH(103),IMG(2,69),FUNCTION
DOUBLE PRECISION ANG(1014,8),KLIM,PTEST(10)
COMPLEX IMAG,COMARR(30,3),ZREF,KITEMS(64)
LOGICAL KEY2,BOOLSET(64,64),TTABLEB(2,20,15)
```

IMPLICIT statements

The ANSI FORTRAN Standard does not provide for an IMPLICIT statement.

An IMPLICIT statement changes or confirms the data typing of constants, variables, arrays, and functions according to the first letter of their symbolic names.

The form of an IMPLICIT statement is

```
IMPLICIT type (a , a ...) [,type (a , a ...)]...
```

where *type* is INTEGER, REAL, DOUBLE PRECISION (or DOUBLE), COMPLEX, or LOGICAL to specify the desired data type, and

a is a single letter or is a range of single letters denoted by the first and last letter of the range separated by a hyphen. Writing a range of letters $a_1 - a_n$ has the same effect as writing a list of the single letters a_1, a_2, \dots, a_n and where a_1 precedes a_n in this alphabetically ordered sequence.

An IMPLICIT statement specifies a type for all constant, variable, array, and function (except intrinsic function) names that begin with any letter appearing singly or within a range in the specification. IMPLICIT statements do not change the types of intrinsic functions. An IMPLICIT statement applies only to the program unit containing it. Note that the hyphen is syntactically significant.

The appearance of a constant, variable, array, or function name in a type statement overrides or confirms type specification by an IMPLICIT statement. An explicit type specification in a FUNCTION statement overrides IMPLICIT statement typing for the name of that function sub-program.

Within the specification statements of a program unit, IMPLICIT statements must precede all specification statements other than PARAMETER statements. A PARAMETER statement must follow an IMPLICIT statement for it to affect the typing of constants named in the PARAMETER statement.

A letter can be specified (or implied within a range of letters) only once in all of the IMPLICIT statements in a program unit.

Examples:

```
IMPLICIT INTEGER(A,B,F-K),REAL(M-W,Z)  
IMPLICIT LOGICAL(L),DOUBLE(D,E)  
IMPLICIT DOUBLE PRECISION(X,Y),COMPLEX(C)
```

PARAMETER statements

The ANSI FORTRAN Standard does not provide for the PARAMETER statement.

A PARAMETER statement assigns a symbolic name to a constant.

The form of a PARAMETER statement is

PARAMETER (p = e[,p = e]...)

where *p* is a symbolic name and

e is an expression containing only constants and
symbolic names of other constants

A symbolic name in a PARAMETER statement can be specified as to type by its appearance in a previous type statement, by a previous IMPLICIT statement specifying its first letter, or by default if not so specified. A symbolic name *p* of type integer, real, double precision, or complex can only be followed by an arithmetic expression *e* containing arithmetic constants or the names of arithmetic constants that have been previously defined in the same or an earlier PARAMETER statement. The evaluation of arithmetic expressions in a PARAMETER statement yields results that are made to agree in type with the corresponding symbolic names. A symbolic name *p* of type logical can only be followed by a logical expression. A symbolic name can be assigned an arithmetic or logical value only once in a program unit. Constants named in a PARAMETER statement may be referenced in any subsequent statement in the same program unit except for a FORMAT statement. Symbolic names of constants cannot be used in format specifications or to form a part of any other constant.

Examples:

```
IMPLICIT LOGICAL(A-B)
PARAMETER (PI=3.1415926, C=1.86E5)
PARAMETER (JOULE=10000000, KELVIN=-273)
PARAMETER (BOOLEAN=.TRUE., ABOOLEAN=.FALSE.)
```

EXTERNAL statements

An EXTERNAL statement identifies a symbolic name as representing an external procedure and permits its use as an actual argument.

The form of an EXTERNAL statement is

EXTERNAL proc [,proc]...

where *proc* is the name of an external procedure.

The appearance of a name in an EXTERNAL statement declares that name to be an external procedure name. If an external procedure name is used as an actual argument in a program unit, it must appear in an EXTERNAL statement in that program unit. A statement function name must not appear in an EXTERNAL statement.

If an intrinsic function or utility procedure name appears in an EXTERNAL statement, that name becomes the name of some external procedure and the intrinsic function or utility procedure of the same name is not available for reference in that program unit. If this external procedure is not available during loading of the executable program, the intrinsic function or utility procedure will be loaded in its place.

A given symbolic name may appear only once in all of the EXTERNAL statements of a program unit.

Example:

MAIN is the main program of an executable program that includes the functions STAT, STDEV, and MEAN. Considering just the main program, the syntax in which the symbolic name STAT appears defines its being the name of a function. The names STDEV and MEAN, however, appear in a syntax incapable of defining their being function names. This definition is established by the EXTERNAL STDEV, MEAN statement in the second line.

PROGRAM MAIN	FUNCTION STDEV(S)
EXTERNAL STDEV,MEAN	:
:	STDEV = RMSD
X = STAT(STDEV,SIGMA)	END
Y = STAT(MEAN,SIGMA)	REAL FUNCTION MEAN(S)
:	:
END	MEAN = AVG
FUNCTION STAT(OP,VALU)	END
:	:
STAT = OP(VALU)	
END	

3.6.3 DATA INITIALIZATION

A DATA statement provides initial values for variables, arrays, and array elements. A DATA statement is non-executable and may appear in a program unit following any specification statements. Only those entities named in DATA statements become defined prior to executable program execution. All other entities are undefined at this time.

Form of a DATA statement

The form of a DATA statement is

DATA *nlist / clist / [[,]nlist / clist] ...*

where *nlist* is a list of variable names, array names, array element names, and implied-DO lists separated by commas, and

clist is a list of the form

[*r**] *c* [, [*r**] *c*] ...

in which *c* is a constant or the symbolic name of a constant, and

r is a nonzero, unsigned, integer constant or the symbolic name of such a constant.

The *r*c* form is interpreted to provide *r* successive appearances of the constant *c*.

An implied-DO list in a DATA statement has the form

(*dlist, i = e1, e2 [, e3]*)

where *dlist* is a list of array element names and implied-DO lists separated by commas,

i is the name of an integer variable called the *implied-DO variable*

e1, e2, and e3 are each integer expressions containing only integer constants, the names of integer constants, and implied-DO variables of other implied-DO lists containing this implied-DO list within their ranges. If omitted, *e3* is assumed specified as 1.

The range of an implied-DO list is the list *dlist*. The iteration count and values of the implied-DO variable *i* are established the same as for a DO-loop except that the iteration count must be greater than zero. Interpretation of an implied-DO list in a DATA statement causes each item in the list *dlist* to be specified once for each iteration, and for appropriate

values to be substituted where implied-DO variables are referenced. Each subscript expression in the list *dlist* must contain only integer constants, integer constant names, and implied-DO variables of implied-DO lists containing the subscript expression within their ranges. Names of implied-DO variables are local to the implied-DO lists containing them.

DATA statement restrictions

Names of constants, dummy arguments, functions, and entities in blank common (including entities associated with an entity in blank common) must not appear in *nlist*. Names of entities in a named common block may appear in *nlist*.

The same number of items must be specified by each *nlist* and its corresponding *clist*. The initial values of the entities are defined by this correspondence. There must be one constant for each element of an array whose name appears in the list without subscripting unless named as the last item of an *nlist*. In this case, the values in *clist* may specify any number of consecutive array element values, beginning with the first. The type of each *nlist* entity and the type of the corresponding *clist* constant must agree when either is of type complex or logical. When the *nlist* entity is of type integer, real, or double precision, the corresponding *clist* constant is converted, if necessary, to the type of the *nlist* entity according to the rules for arithmetic conversion. A Hollerith constant can be specified to correspond to entities of any type except logical.

Any variable or array element may be initially defined except for:

- An entity that is a dummy argument,
- An entity in blank common, which includes an entity associated with an entity in blank common, or
- A variable in a function subprogram whose name is also the name of the function subprogram.

If a variable, an array element, or an entity associated with either is defined by a DATA statement more than once in an executable program, that nearest its end is the only definition to apply.

The ANSI FORTRAN Standard does not permit *nlist* to contain an unsubscripted array name, or for a DATA statement to initialize entities in named common blocks except in block data subprograms.

Examples:

```
DIMENSION GRID (2,3),KBUF(10,200,2)
PARAMETER (XCON=6.0)
DATA GRID /11.0,21.0,12.0,22.0,13.0,23.0/,KBUF/4000*XCON/
DATA I/1/J/0/K/2000/
```

3.6.4 DATA ASSIGNMENT

Data assignment defines variables and array elements during execution of the executable program. The four kinds of data assignment statements are:

- Arithmetic,
- Logical,
- Formatted, and
- ASSIGN (statement label).

Assignment statements defines entities. Formatted assignment statements are described in 3.6.9.

Arithmetic assignment statements

The form of an arithmetic assignment statement is

$$\boxed{v = e}$$

where v is the name of a variable or array element of type integer, real, double precision, or complex, and

e is an arithmetic, Hollerith, or Boolean expression.

Execution of an arithmetic assignment statement causes the evaluation of the expression e , conversion of e to the type of v (if required), and definition of v with the resulting value. Once evaluated, an arithmetic expression may require conversion to the data type of the entity v . Table 3.7 relates such conversions to the data types of arithmetic operands, expressions and evaluations.

Examples:

<u>The statement ...</u>	<u>Assigns to a(n) ...</u>
$L = 12$	Integer variable
$C = (0.8,16.5) - (16.32,-6.1)$	Complex variable
$X = -B + (B^{**2}-4*A*C)^{**0.5}$	Real variable
$A = B + L$	Real variable
$ROOT = SQRT(65536.0)$	Real variable
$ARRAY(6,2,1)=0$	Real array element
$MATRIX(I,J,K)=MATRIX(I,J,K)+1$	Integer array element

Logical assignment statements

The form of a logical assignment statement is

$$v = e$$

where v is the name of a logical variable or array element, and e is a logical expression.

Execution of a logical assignment statement causes the evaluation of the expression e and the definition of v with the value of e . e must either be evaluated as true or false.

Examples:

All variable and array element names are assumed to be of type logical except for E and F which are type real.

```
T = .FALSE.  
A = B  
C = (A .AND. B) .OR. (C .AND. D)  
T = .NOT. T  
TRUTAB(I,J,K,L) = .T.  
T = E.GE.F .OR. E/F .LT..4
```

ASSIGN statements

The form of an ASSIGN (statement label) statement is

$$\text{ASSIGN } s \text{ TO } i$$

where s is a statement label, and

i is an integer variable name.

An ASSIGN statement assigns the statement label s to the integer variable i . s must be the label of an executable statement or a FORMAT statement in the same program unit as the ASSIGN statement.

Execution of an ASSIGN statement is the only way to define a variable with a statement label.

A variable defined with a statement label can be referenced only in an assigned GO TO statement, or as a FORMAT statement identifier in an input/output statement, a formatted assignment statement, or an argument. While so defined, the variable i cannot be referenced for any other purpose. It may be redefined with the same or a different statement label or with an integer, Hollerith, or Boolean value.

Note that "TO" is a keyword extension in the ASSIGN statement and is not a part of the keyword "ASSIGN" itself. Both must appear in an ASSIGN statement.

Examples:

```
ASSIGN 910 TO JUMPTO  
ASSIGN 6 TO NEXTPATH  
ASSIGN 12345 TO NUFORMAT
```

3.6.5 PROGRAM CONTROL

The following statements control an execution sequence:

- Unconditional GO TO
- Computed GO TO
- Assigned GO TO
- Three-branch arithmetic IF
- Two-branch arithmetic IF
- Direct logical IF
- Indirect logical IF
- DO
- CONTINUE
- STOP
- PAUSE
- END
- CALL } described in 3.6.11
- RETURN }

Unconditional GO TO statements

The form of an unconditional GO TO statement is

GO TO s

where *s* is the statement label of an executable statement in the same program unit.

Execution of an unconditional GO TO statement causes a transfer of control to the statement identified by the statement label.

Note that the space between GO and TO is optional.

Examples:

```
GOTO 910  
GO TO 6  
GO TO12345
```

Computed GO TO statements

The form of a computed GO TO statement is

GO TO (*s*[,*s*]...) [,]*e*

where *e* is an integer expression, and

s is the statement label of an executable statement that appears in the same program unit as the computed GO TO statement. A given statement label may appear more than once in a computed GO TO statement.

Execution of a computed GO TO statement causes the expression *e* to be evaluated for an integer result, *i*. A transfer of control to the statement identified by the *i*th statement label in the list of *n* statement labels is then executed if $1 \leq i \leq n$. If $i < 1$ or $i > n$, the execution sequence proceeds as though a CONTINUE statement were executed. If the evaluation of *e* for *i* produces a non-integer result, the fractional portion is discarded without rounding.

The ANSI FORTRAN Standard does not specify the action of a computed GO TO statement when $i < 1$ or $i > n$, and requires that *e* be an integer variable preceded by a comma.

Note that the space between GO and TO is optional and that an optional comma may precede *e*.

Examples:

```
A =3.  
GO TO (2,4,8,16)A  
IX=MSIZE/2  
GO TO (0031,59,728)IX  
GO TO (0031,59,728)MSIZE/2  
NBRANCH =4  
GOTO (6,3,6,6,7,2,7),NBRANCH
```

Assigned GO TO statements

The form of an assigned GO TO statement is

GO TO i [,] (s [,] s ...)

where i is an integer variable name, and

s is the statement label of an executable statement that appears in the same program unit as the assigned GO TO statement. A given statement label may appear more than once in this statement.

Execution of an assigned GO TO statement causes that value of i , as assigned by prior execution of an ASSIGN statement, to define a statement label of an executable statement. Execution of the assigned GO TO statement causes a transfer of control to the statement identified by that statement label. Definition of the variable used in an assigned GO TO statement must occur through use of an ASSIGN statement previously executed in the same program unit.

The optional parenthesized statement label list should, in principle, contain at least one specification of the statement label assigned to i . If this is not the case, transfer of control will still occur to that statement identified by the label in i .

Note that the space between GO and TO is optional and that an optional comma may follow i if a list of statement labels is specified.

The ANSI FORTRAN Standard requires the parenthesized statement label list and the comma preceding it, and does not specify what action is taken if there is no s to match i .

Examples:

```
ASSIGN 76 TO LAB
```

```
⋮
```

```
GO TO LAB
```

```
ASSIGN 999 TO KFIN
```

```
⋮
```

```
GO TO KFIN (997,997,999)
```

```
ASSIGN 1 TO JAIL
```

```
⋮
```

```
GO TO JAIL,(1,2,3,4,5)
```

Three-branch arithmetic IF statements

The form of a three-branch arithmetic IF statement is

```
IF ( e ) s1,s2,s3
```

where e is an integer, real, or double-precision expression, and

s_1, s_2 , and s_3 are statement labels of executable statements that appear in the same program unit as the three-branch arithmetic IF statement. The same statement label may appear more than once in this statement.

Execution of a three-branch arithmetic IF statement causes evaluation of the expression e followed by a transfer of control to one of the statements identified by s_1 , s_2 , or s_3 , as the value of e is less than zero, equal to zero, or greater than zero, respectively.

The ANSI FORTRAN Standard arithmetic IF statement and the three-branch arithmetic IF statement are the same.

Examples:

```
IF (VTEST) 20,21,20  
IF (B**2-4*A*C) 70,80,90  
IF (SIN(ALPHA)) 6482,4826,8264
```

Two-branch arithmetic IF statements

The form of a two-branch arithmetic IF statement is:

```
IF ( e ) s1,s2
```

where e is an integer, real, or double-precision expression, and s_1 and s_2 are statement labels of executable statements that appear in the same program unit as the two-branch arithmetic IF statement.

The ANSI FORTRAN Standard does not provide for the two-branch arithmetic IF statement.

Execution of a two-branch arithmetic IF statement causes evaluation of the expression e followed by a transfer of control to the statement identified by s_1 if e is non-zero or to the statement identified by s_2 if e is zero.

Examples:

```
IF (I+J*K) 100,101  
IF (SIN(ALPHA)) 31,49627  
IF (2+3-M) 44,20
```

Direct logical IF statements

The form of a direct logical IF statement is

```
IF ( e ) st
```

where e is a logical expression, and

st is any executable statement other than a DO statement, an END statement, or another direct logical IF statement and is called a *conditional statement*.

Execution of a direct logical IF statement causes evaluation of the expression e for a logical value. If the value of e is true, statement st is executed. If the value of e is false, statement st is not executed and the execution sequence proceeds as though a CONTINUE statement were executed. The execution of a function reference in the expression of a direct logical IF statement may affect related entities in the statement st .

The ANSI FORTRAN Standard refers to the direct logical IF statement as the logical IF statement.

Examples:

```
LOGICAL K  
IF(K) K=.NOT.K  
  
INTEGER A,B  
IF (A.EQ.B) GO TO 100  
  
LOGICAL M,N,P,Q  
IF (M.AND.N.OR.P.AND.Q) M=P.OR.Q.OR.N
```

Indirect logical IF statements

The form of an indirect logical IF statement is

```
IF (e) s1,s2
```

where e is a logical expression and s_1 and s_2 are statement labels of executable statements appearing in the same program unit as the indirect logical IF statement.

The ANSI FORTRAN Standard does not provide for the indirect logical IF statement.

Execution of an indirect logical IF statement causes evaluation of the expression e for a logical value followed by a transfer of control. If the value of e is true, the statement identified with statement label s_1 is executed next. If the value of e is false, the statement identified with statement label s_2 is executed next.

Examples:

X and Y are numeric variables; all others are logical

```
IF((A.OR.B).AND.C)210,220  
IF(X.GE.Y)148,9999  
IF(L1.EQV.L2)3,44
```

DO statements

A DO statement specifies a DO-loop. A *DO-loop* enables repeated execution of the set of executable statements within its range.

The form of a DO statement is

```
DO s[,]i = e1 ,e2 [, e3]
```

where

s is the statement label of an executable statement called the *terminal statement* that physically follows the DO statement in the same program unit;

i is the name of an integer, real, or double-precision variable, called the *DO-variable*; and

e_1 , e_2 , and e_3 are integer, real, or double-precision expressions.

The ANSI FORTRAN Standard requires that i be an integer variable, that e be an integer variable or integer constant, and that both i and e be greater than zero when the DO statement is executed. The ANSI FORTRAN Standard does not provide for a comma between s and i .

The terminal statement of a DO-loop must not be an unconditional GO TO, assigned GO TO, indirect logical IF, two- or three-branch arithmetic IF, RETURN, PAUSE, STOP, END, or another DO statement.

The ANSI FORTRAN Standard specifies that the terminal statement of a DO-loop may not be a DO, PAUSE, STOP, RETURN, unconditional GO TO, assigned GO TO, computed GO TO, or arithmetic IF. Nor is a (direct) logical IF statement permitted if its conditional statement is one of these.

Range of a DO-loop - The range of a DO-loop consists of all executable statements between and including the first executable statement following the DO statement, and the terminal statement of the DO-loop.

A DO-loop may appear within a DO-loop and must be entirely contained within the outer DO-loop range. More than one DO-loop may have the same terminal statement.

Active and inactive DO-loops - A DO-loop is either active or inactive. A DO-loop is initially inactive and becomes active only when its DO statement is executed.

An active DO-loop becomes inactive when:

- Its iteration count is zero,
- A RETURN, PAUSE, STOP, or END statement is executed in the same program unit,
- It is in the range of another DO-loop that becomes inactive, or
- It is in the range of another DO-loop whose DO statement is executed.

Note that transfer of control out of the range of a DO-loop does not inactivate the DO-loop. However, the DO-loop becomes inactive if the DO-variable becomes undefined or is redefined while outside the range.

When a DO-loop becomes inactive, the DO-variable retains its last defined value unless it became undefined due to earlier action.

Executing a DO statement - Executing a DO statement initiates the following sequence of steps:

- The *initial parameter* m_1 , the *terminal parameter* m_2 , and the *incrementation parameter* m_3 are established by evaluating e_1 , e_2 , and e_3 , respectively, including any necessary conversion to the type of the DO-variable according to the rules for arithmetic conversion. If e_3 does not appear, m_3 is assigned a value of one. m_3 may be positive or negative but must not be zero.

- The DO-variable i becomes defined with the value of the initial parameter m_1 .
- The *iteration count* is established as an integer value equal to the integer portion of the expression

$$(m_2 - m_1 + m_3) / m_3$$

or as zero in the event that

$$m_1 > m_2, \text{ and } m_3 > 0 \text{ or}$$

$$m_1 < m_2, \text{ and } m_3 < 0.$$

Once the iteration count is established, the DO-variable and entities named in the initial, terminal, and incrementation parameters (e_1 , e_2 , and e_3) may be redefined with no effect on loop control processing.

At the completion of DO-statement execution, loop control processing begins.

Loop control processing - Loop control processing determines if execution of the range of the DO-loop is required. If the iteration count is not zero, control transfers to the first statement in the range of the DO-loop. If the iteration count is zero, the DO-loop becomes inactive. If, as a result, all DO-loops sharing the terminal statement of this DO-loop are inactive, control is transferred to the first executable statement after the terminal statement. However, if any DO-loops sharing the terminal statement are active, execution resumes with incrementation processing as described in the third paragraph following.

The ANSI FORTRAN Standard specifies that the test for a zero iteration count is to occur at the conclusion of incrementation processing.

Execution of the range - Statements in the range of a DO-loop are executed until the terminal statement is reached.

Terminal statement execution - Execution of the terminal statement occurs during a normal execution sequence or through transfer of control. If execution of the terminal statement does not cause a transfer of control, execution continues with incrementation processing, as described below.

Incrementation processing - Incrementation processing has the effect of performing the following steps in sequence:

- The value of the DO-variable is incremented by the value of m_3 .
- The iteration count is decremented by one.

The ANSI FORTRAN Standard specifies the tests for iteration count at this point. A non-zero iteration count results in continuation as described below; a zero iteration count is processed as already described for such.

- Execution continues with loop control processing of the same DO-loop whose iteration count was decremented.

Note that a DO-variable may increase or decrease in value during incrementation processing.

The ANSI FORTRAN Standard does not permit the value of the DO-variable to decrease.

Examples:

PARAMETER(N=50)	PARAMETER(I=2,J=200)
DIMENSION TABLE (N)	DIMENSION GRID(I,J), PGRID(I,J)
DO 2 I=1,N	DO 22 K=I,1,-1
IF(TABLE(1))2,2,1	DO 22 L=J,1,-1
1 TABLE(I)=-TABLE(I)	PGRID(K,L) = GRID(K,L)
2 TABLE(I)=-TABLE(I)	IF(PGRID(K,L))21,22,22
⋮	21 PGRID(K,L) = -PGRID(K,L)
	22 GRID(K,L) = 0
	⋮
M=0	
DO 100 I=1,10	
J=I	
DO 100 K=1,5	
L=K	
100 M=M+1	

In the last example, I=11, J=10, K=6, L=5, and M=50 after the last statement is executed for the last time.

Transfer into the range of a DO-loop - Control must not transfer into the range of an inactive DO-loop. Control may transfer to any executable statement in the range of an active DO-loop.

CONTINUE statements

The form of a CONTINUE statement is

```
CONTINUE
```

Execution of a CONTINUE statement has no effect.

A CONTINUE statement is commonly used as the terminal statement of a DO-loop. As with any statement so used, the next statement executed depends on the result of DO-loop incrementation processing. Note that this action is the result of DO-loop processing and not of CONTINUE statement execution.

Examples:

```
PARAMETER (N6=5050)
DIMENSION ARRAY6(16)
DO 22,I=16,1,-1
IF(ARRAY6(I).NE.0) ARRAY6(I)=1.0/ARRAY6(I)
22 CONTINUE
```

STOP statements

The form of a STOP statement is

```
STOP[id]
```

where *id*, if used, must be:

- A string of up to eight digits,
- A Hollerith constant of up to eight characters contained in parentheses, or
- The symbolic name of a variable, array element, or function containing (or providing) eight Hollerith characters.

The ANSI FORTRAN Standard limits the string *id* to five octal digits

A STOP statement is used in a main program, subroutine subprogram, or function subprogram to terminate execution of the executable program. Specification or non-specification of *id* has no effect on the executable program. The Hollerith characters specified by *id* appear in a logfile message to identify the STOP statement encountered during executable program execution.

PAUSE statements

The form of a PAUSE statement is

PAUSE[*id*]

where *id*, if used, must be:

- A string of up to eight digits,
- A Hollerith constant of up to eight characters contained in parentheses, or
- The symbolic name of a variable, array element, or function containing (or providing) eight Hollerith characters.

The ANSI FORTRAN Standard limits the string *id* to five octal digits.

A PAUSE statement is used in a main program, subroutine subprogram, or function subprogram to terminate execution of the executable program. Specification or non-specification of *id* has no effect on the executable program. The Hollerith characters specified by *id* appear in a logfile message to identify the PAUSE statement encountered during executable program execution.

The ANSI FORTRAN Standard provides for continuation of executable program execution following a PAUSE statement conditional upon external factors.

END statements

The ANSI FORTRAN Standard provides for an END line, but not for an END statement.

The form of an END statement is

END

An END statement is required at the physical end of the sequence of statements and lines of every program unit. When executed in a subprogram, it has the effect of a RETURN statement. When executed in a main program, it has the effect of a STOP statement.

No other statement in a program unit may be expressed with an initial line that appears to contain an END statement.

The last line of every program unit must be an initial line that contains a complete END statement. This special form of initial line is called a terminal line. Note that a single END statement can appear with one or more STOP statements or with one or more RETURN statements in the same program unit.

3.6.6 INPUT/OUTPUT OPERATIONS

Input statements provide the means of transferring data from the mass storage subsystem to the memory section of the CPU. This process is called *reading*. *Output statements* provide the means of transferring data from memory to mass storage. This process is called *writing*. *Editing* of the data may also be performed.

In addition to statements that transfer data, auxiliary input/output statements are provided for the manipulation of external media.

Input/output records, files, units, and formats

Records - A record is a sequence of values or characters. For example, a punched card is usually considered to be a record. A record may or may not correspond to a physical entity.

The three kinds of records are:

- Formatted,
- Unformatted, and
- End-of-file or endfile.

Formatted records

A formatted record consists of a sequence of characters. Its length is measured in characters, depends primarily on the number of characters transferred when written. The length may also depend on characteristics of the peripheral device (i.e., line printer, card reader) serving as the origin or ultimate destination for the data. Formatted records may be read or written by formatted input/output statements, or may be prepared by means other than FORTRAN. Unformatted and buffered input/output statements can also read and write formatted records, but do so in a manner that ignores their formatted characteristics.

Unformatted records

An unformatted record consists of a sequence of character and/or noncharacter data. The length of an unformatted record is measured in storage units (words).

Unformatted records can be read or written by unformatted and buffered input/output statements.

End-of-file (endfile) records

An endfile record is written by an ENDFILE statement, must occur only as the last record of a file, and has no length property.

Files - A *file* is a sequence of records.

File existence

A set of files *exists* for an executable program when each can be identified and/or referenced by name. A file may be present yet not exist for an executable program at a particular time. A file may also exist that contains no records.

Creating a file causes a file to exist that did not previously exist. *Deleting* a file terminates its existence.

All input/output statements may refer to files that exist. Output statements may also refer to files that do not exist.

Record and file positions - Because records and files both exist as elements in sequences, the position of a record or a file may be described in terms of its position in a sequence. Certain circumstances can cause this position to become indeterminate.

In a sequence, the *initial point* is the position just before the first element. The *terminal point* is the position just after the last element.

If a sequence is positioned at a point within an element, that element is the *current element*; otherwise, there is no current element.

A *preceding element* is that element preceding the current element or terminal point. No preceding element exists for the initial point of a sequence or for the terminal point if the sequence is empty.

The *next element* of a sequence immediately follows the current element. No next element exists for the terminal point of a sequence or for the initial point if the sequence is empty.

Each file contains all formatted or all unformatted records and is terminated with an endfile record.

Datasets - A *dataset* is a sequence of all files associated with a particular unit during execution of the executable program. Association of a dataset with a particular unit is not under control of the executable program. Datasets are described in Cray Research publication 2240011, "CRAY-OS Version 1.0 External Reference Specification".

Internal records, files and datasets - *Internal records, internal files, and internal datasets* are analogous to records, files, and datasets except for there being no associated unit and their pertaining only to formatted data assignment operations performed by the ENCODE and DECODE statements.

Sequential and random input/output operations - All input/output operations are based upon the sequential storage in memory of files within datasets and records within files. *Random input/output operations* access these entities in a non-sequential order. The two utility procedures enabling random input/output operations and techniques for their use are described in Appendix H.

Units - A *unit* is a means of referring to a file.

Unit existence

At any given time, a set of units *exists* for an executable program. All input/output statements may refer to units that exist.

Unit identifiers

The form of a *unit identifier* is

u

A unit identifier is specified as an integer expression or as a Hollerith expression of up to eight characters.

The ANSI FORTRAN Standard does not provide for *u* being specified as an expression, and allows only an integer constant or variable to be used.

The assignment of unit identifiers to specific files is not under the control of the executable program. *u* is converted to an integer value to correspond with an assigned identifier. The unit so identified is the same for all program units of the executable program.

Formats - The form of a *format identifier* is

f

A format identifier must be either an array name or the label of a FORMAT statement in the same program unit.

READ, WRITE, PRINT and PUNCH statements

The READ statement is the data transfer input statement. WRITE, PRINT, and PUNCH statements are data transfer output statements. The forms of these statements are:

```
READ (cclist) [iolist]
READ f [,iolist]
WRITE (cclist) [iolist]
PRINT f [,iolist]
PUNCH f [, iolist]
```

where *cclist* is a control information list that includes a reference to the source or destination of the data to be transferred and an optional format identifier for editing processes,

f is a format identifier, and

iolist is an input/output list specifying the data to be transferred.

The ANSI FORTRAN Standard does not provide for READ f[,iolist], PRINT and PUNCH statements.

Control information lists - The form of a *control information list* *clist* is

$u[,f]$

A control information list contains one unit identifier and may contain a format identifier.

If the control information list contains a format identifier, the statement is a *formatted input/output statement*; otherwise, it is an *unformatted input/output statement*.

Examples:

```
READ (98,12345) ...
WRITE (K-1,306) ...
READ (5,ARRAYF) ...
WRITE ('$OUT',99) ...
PRINT 22
```

Input/output lists - An *input/output list*, *iolist*, specifies entities whose values are transferred by input/output statements. This list is composed of one or more input/output list items separated by commas and may include one or more optional implied-DO lists. An *input/output list item* is either an input or an output list item.

An array name appearing as an input/output list item is treated as if all elements of the array were specified in the order given by array element ordering.

Input list items

An *input list item* must be:

- A variable name,
- An array element name, or
- An array name.

Only input list items may appear in an input statement.

Output list items

An *output list item* must be:

- A variable name,
- An array element name,
- An array name, or
- An expression

Input/output lists are specified with ENCODE and DECODE statements the same as for input/output statements.

Examples:

	READ(23)	WRITE(7)
	READ(23,1066)	WRITE(7,2066)
1066	FORMAT ...	2066 FORMAT ...
	READ(23,FARRAY)	WRITE(7,FMATRIX)
	READ(23) BINDATA1,BINDATA2,...	WRITE(7) BINDATA1,BINDATA2,...
	READ(23,1067) A,B,C, ...	WRITE(7,2067)A,B,C,...
1067	FORMAT ...	2067 FORMAT ...
	READ(23,FVECTOR) X1,X2, ...	WRITE(7,FVECTOR) X1,X2, ...
	READ 1068	PRINT 2068
1068	FORMAT ...	2068 FORMAT ...
	READ 1069,A,X,B,Y, ...	PUNCH 2069,A,X,B,Y ...
1069	FORMAT ...	2069 FORMAT ...
	READ('\$INPUT') ...	WRITE('BINOUT') ...

Implied-DO lists

An *implied-DO list* is of the form

(*dlist*, *i* = *e*₁, *e*₂ [,*e*₃])

where *i*, *e*₁, *e*₂, and *e*₃ are as specified for the DO statement, and
dlist is an input/output list.

The range of an implied-DO list is the list *dlist*. Note that *dlist* may itself contain one or more implied-DO lists. The iteration count and the value of the DO-variable *i* are established from *e*₁, *e*₂, and *e*₃ exactly as for a DO-loop. Once the values of *i* and of the iteration count are established, *i*, *e*₁, *e*₂, and *e*₃ may be redefined with no effect on the loop control process. The DO-variable *i* may be specified as a subscript to array elements named in *dlist* for both input and output list items. When an implied-DO list appears in an input/output list, it is treated as if *dlist* were specified once for each iteration of the implied-DO list.

Examples:

```
PRINT 311,(VECTOR(I),I=1,100)
311 FORMAT ...
READ(12,345)((XREF(M,N),M=1,2),N=1,3)
345 FORMAT ...
WRITE(6,350)(M,(N,XREF(M,N),N=1,3),M=2,1,-1)
350 FORMAT ...
```

Execution of data transfer input/output statements

The effect of executing a data transfer input/output statement must be as if the following operations were performed in the order specified:

- Determine the direction of data transfer.
- Identify the unit.
- Establish the format (if specified).
- Transfer data between the file and the entities specified by the input/output list (if any).

Direction of data transfer - Execution of a READ statement causes values to be transferred from a file to the entities specified by the input list, if present.

Execution of a WRITE, PRINT or PUNCH statement causes values to be transferred to a file from the entities specified by the output list and format specification (if any). The WRITE, PRINT, and PUNCH statements are treated identically in this regard. Execution of a WRITE, PRINT, or PUNCH statement for a file that does not exist creates that file.

Identifying a unit - A READ statement that does not contain a control information list specifies a particular predetermined unit. PRINT and PUNCH statements similarly specify separate and unique predetermined units. Unit prespecification is not under the control of the executable program.

The unit identified by a data transfer input/output statement must be associated with a file when the statement is executed.

Establishing a format - The presence of a format identifier in a control information list identifies a format specification.

File position prior to data transfer - The position of a file is not changed prior to data transfer. The next record becomes the current record during the transfer or during positioning caused by T or X edit descriptors (as described in later paragraphs).

Data transfer - Data are transferred between records and entities specified by the input/output list. List items are processed in the order of their left-to-right appearance in the input/output list.

All values needed to determine which entities are specified by an input/output list item are determined at the beginning of the processing of that item. In the example,

```
N(1) = 3  
READ (8) N(N(1))
```

a value reads into N(3). Note that the array element item is a single input/output list item.

All values are transmitted to or from the entities specified by a list item prior to the processing of any succeeding list item. In the example,

```
READ (3) N, A(N)
```

the first value read is assigned to N, and the second is assigned to A(N) where the new value of N is used as the subscript.

A DO-variable in an implied-DO list becomes defined at the beginning of processing the implied-DO list as an input/output list item.

Reading an endfile record causes the contents of input list items specified in the READ statement to become undefined.

An input list item, or any entity associated with it, must not affect any portion of the established format specification.

Unformatted data transfer

During unformatted data transfer, data are transferred without editing between the current record and the entities specified by the input/output list. Exactly one record is read or written.

On input, the file must be positioned so that the record read is an unformatted record or an endfile record. The number of values required by the input list must be less than or equal to the number of values in the record and must not require more values than the record contains.

Formatted data transfer

During formatted data transfer, data are transferred with editing between the entities specified by the input/output list and the file. The current record and possibly additional records are read or written.

On input, the record read must be a formatted record or an endfile record.

The input/output list and format specification must not require more characters than a record contains.

On output, the output list and format specification must not specify more than 152 characters per record.

The ANSI FORTRAN Standard does not provide for a maximum number of characters per record.

The transfer of formatted record information to certain devices is termed printing. The first character of a formatted record is not printed. The remaining characters of the record, if any, are printed in one line beginning at the left margin.

The first character of such a record determines the vertical spacing to occur before printing. The character codes specifying vertical spacing (carriage) control are shown in table 3-12.

Table 3-12. Print control characters

Character	Vertical spacing before printing
Blank	Advance one line
0	Advance two lines
1	Advance to first line of next page
+	No advance
All other	Advance one line

If there are no characters in the record, an advance of one line occurs. Nothing is printed in that line.

Note that a PRINT statement does not necessarily result in a printing operation.

File position after data transfer - If an error condition exists, the position of the file is indeterminate.

If an end-of-file condition exists as a result of reading an endfile record, the file is positioned after the endfile record.

If no error condition or end-of-file condition exists, the file is positioned after the last record read or written.

BACKSPACE, ENDFILE, and REWIND statements

The forms of the BACKSPACE, ENDFILE, and REWIND statements are:

BACKSPACE	<i>u</i>
BACKSPACE	(<i>u</i>)
ENDFILE	<i>u</i>
ENDFILE	(<i>u</i>)
REWIND	<i>u</i>
REWIND	(<i>u</i>)

where *u* is a unit identifier.

The unit specified by a BACKSPACE, REWIND, or ENDFILE statement must be associated with a file. Parentheses must delimit a unit identifier specified as a Hollerith constant.

BACKSPACE statements - A BACKSPACE statement causes the file related to the specified unit to be positioned before the preceding record. If there is no preceding record, the position of the file is not changed. Note that if the preceding record is an endfile record, the file is positioned before it.

ENDFILE statements - An ENDFILE statement writes an endfile record as the next record of the file. The file is then positioned after the endfile record. Execution of an ENDFILE statement for a file that does not exist creates that file.

REWIND statements - A REWIND statement causes the specified file to be positioned at its initial point. If the file is already positioned at its initial point, execution of this statement has no effect on the position of the file.

Restrictions on input/output statements

A function must not be referenced in an input/output statement if it causes an input/output statement to be executed.

An input/output statement must not reference a unit or file not having all the properties required for its execution.

3.6.7 FORMAT SPECIFICATION

A *format specification* provides explicit editing information to formatted input/output and formatted assignment statements to direct the editing of data between its internal representation and the corresponding character strings required. Format specifications may be given in FORMAT statements or as character values in arrays.

A FORMAT statement is not required if the format identifier in a formatted input/output or formatted assignment statement is an array name. The initial and following elements of that array must be defined with character data that constitute a format specification when the input/output statement is executed. The specification is considered to be contained in a concatenation of all elements in the array in the order given by array element ordering. The opening parenthesis must be in the first or ninth array element. If in the ninth array element, the content of the first eight elements has no effect on program execution.

A format identifier that is a statement label must be the label of a FORMAT statement in the same program unit. The format specification contained in that FORMAT statement is applied when the formatted input/output or assignment statement is executed.

In either case, a format specification begins with a left parenthesis and ends with a right parenthesis. A complete format specification may contain a format specification which, in turn, contains a format specification. Deeper nesting of format specifications must not be specified. Character data following the right parenthesis of a complete format specification is ignored only when the specification is contained in an array.

FORMAT statements

The form of a FORMAT statement is

FORMAT ([*flist*])

where *flist* is a format specification.

FORMAT statements must always be labeled.

Form of a format specification

The form of a format specification is

([*flist*])

where *flist* is a list in which each list item has one of the forms:

ned
ed
[*r*] (*flist*)

and where

- ned* is a nonrepeatable edit descriptor,
- ed* is a repeatable edit descriptor,
- r* is a nonzero, unsigned integer constant called a repeat specification, and
- flist* is a format specification with a non-empty list.

Commas may separate list items in *flist* but are required only between:

- Two adjacent digits where each belongs to different list items, and
- Between two adjacent apostrophe or quotation mark delimiters of separate edit descriptors.

The ANSI FORTRAN Standard does not provide for the optional use of commas except before and after the slash edit descriptor.

Examples:

```
1999 FORMAT (1H1,5X,6F6.2)
1234 FORMAT (6HABC123,2X,"=",D15.5,2X,16)
```

Array form:

FMT1	(6HABC12
	3,2X,"=
	,D15.5,2
	X,16)

Edit descriptors

An *edit descriptor* is either a repeatable edit descriptor *ed* or a non-repeatable edit descriptor *ned*.

The forms of *repeatable edit descriptors* are:

[r]Iw
[r]Fw.d
[r]Ew.d
[r]Dw.d
[r]Gw.d
[r]Ow
[r]Zw
[r]Lw
[r]Aw
[r]Rw

where; I, F, E, D, G, O, Z, L, A, and R indicate the manner of editing,
w and r are nonzero, unsigned integer constants, and
d is an unsigned integer constant.

The *repeat specification*, r, optionally precedes any repeatable edit descriptor.

The ANSI FORTRAN Standard does not provide for the O, Z, or R edit descriptors.

Examples:

13	F8.5	E19.12	D8.1	G13.3	023	Z6	L7	A8	R6
519	2F6.0	12E7.2	3D10.0	29G5.0	6023	2Z10	7L7	3A5	4R4

The forms of *nonrepeatable edit descriptors* are:

' $h_1 h_2 \dots h_n$ '	(apostrophe)
* $h_1 h_2 \dots h_n$ *	(asterisk)
" $h_1 h_2 \dots h_n$ "	(quotation mark)
$nHh_1 h_2 \dots h_n$	
Tc	
[b] x	
[b] /	(slash)
:	(colon)
kP	

where; apostrophe, asterisk, quotation mark, H, T, X, slash, colon, and P indicate the manner of editing,

h is one Hollerith character,

b, c, and n are nonzero, unsigned integer constants, and

k is an optionally signed integer constant.

The ANSI FORTRAN Standard does not provide for the apostrophe, asterisk, quotation mark, or T edit descriptors, or for the b/ and X forms of the slash and X edit descriptors.

Examples:

```
'AN APOSTROPHE EDIT DESCRIPTOR'  
*AN ASTERISK EDIT DESCRIPTOR*  
"A QUOTATION-MARK EDIT DESCRIPTOR"  
20HAN H EDIT DESCRIPTOR  
T112  
55X  
/  
6/  
:  
3P
```

Interaction between input/output lists and format specifications

The beginning of execution of a formatted input/output statement or a formatted assignment statement initiates format control. Each action of format control depends on information from:

- The next edit descriptor provided by the format specification, and
- The next item in the input/output list, if one exists.

If a statement has an input/output list, at least one repeatable edit descriptor must exist in the format specification.

An empty format specification of the form () may be used unless contained within another format specification. An empty format specification causes one input or internal record to be skipped or one output or internal record containing no characters to be written. There must be no input/output list items corresponding to an empty format specification. Except for repeated edit descriptors and embedded format specifications, a format specification is interpreted from left to right. An embedded format specification or edit descriptor preceded by an *r* is processed as a list of *r* format specifications or edit descriptors identical to that preceded by the *r*. An omitted repeat specification is treated the same as a repeat specification whose value is one.

Each repeatable edit descriptor interpreted in a format specification corresponds to one item specified by the input/output list, except that an item of type complex requires the interpretation of two F, E, D, G, A, or R edit descriptors. An input/output list contains no items corresponding to non-repeatable edit descriptors.

When format control encounters a repeatable edit descriptor, it determines whether there is a corresponding item specified by the input/output list. If there is, format control transmits appropriately edited information between the item and the record, then proceeds. If there is not, format control terminates.

If a colon edit descriptor is encountered and if there are no more input/output list items to be processed, format control is terminated. If there are more input/output list items, the colon edit descriptor is ignored.

Format control also terminates if it encounters the rightmost parenthesis of a complete format specification and if no additional input/output list items are specified. If another list item is specified, the file is positioned before the next record and format control reverts to the beginning of that format specification terminated by the next-to-last right parenthesis. If there is none, format control reverts to the left parenthesis of the complete format specification. If reversion occurs, the reused portion of the format specification must contain at least one repeatable edit descriptor. If format control reverts to a parenthesis that is immediately preceded by a repeat specification, the repeat specification is reused. Reversion of format control, of itself, has no effect on the scale factor (see P editing).

Positioning by format control

If a T or X edit descriptor is the first edit descriptor encountered after format control is initiated, its action causes the next record to become the current record.

After processing each repeatable edit descriptor or an H, apostrophe, asterisk, or quotation mark edit descriptor, the file is positioned after the last character read or written in the current record.

After processing a T, X, slash or colon edit descriptor, the file is positioned as separately described for each.

If format control reverts, the file is positioned in a manner identical to that when a slash edit descriptor is processed.

After a read operation, any unprocessed characters of the record read are skipped.

When format control terminates, the file is positioned after the current record.

Editing

Edit descriptors specify the form of a record and direct the editing between characters in a record and their corresponding internal representation.

A field is a part of a record that is read or written when format control processes a single repeatable edit descriptor or an H, apostrophe, asterisk, or quotation mark edit descriptor. *Field width* is the size of the field in characters.

The internal representation of a datum corresponds to the internal representation of a constant of similar type.

Apostrophe, quotation mark, and asterisk editing - An apostrophe, quotation mark, or asterisk edit descriptor has the form of a character constant and causes characters to be written from the delimited characters (including blanks) of the edit descriptor itself. These edit descriptors only apply to output. The width of the field is the number of characters contained between (but not including) the delimiting quotation marks, asterisks, or apostrophes. Within the field, two adjacent apostrophes or quotation marks are counted as one and not as members of a delimiting apostrophe or quotation mark character pair, respectively. Note that this does not apply in the case of asterisks used as a delimiting character pair.

The ANSI FORTRAN Standard does not provide for apostrophe, quotation mark, or asterisk editing. (The proposed ANSI FORTRAN Standard provides only for apostrophe editing; i.e., the use of quotation mark and asterisk editing may seriously affect FORTRAN program transportability.)

Examples:

```
      WRITE(6,13)
13  FORMAT(' ISN''T ''*' BETTER','' THAN ''''H''' IS")
```

H editing - The nH edit descriptor causes character information to be written from the n characters (including blanks) following the H of the edit descriptor. An H edit descriptor can be used only for output.

The ANSI FORTRAN Standard provides for use of the H edit descriptor in input as well as output.

Examples:

```
PRINT 22
22 FORMAT(27H ABCDEFGHIJKLMNOPQRSTUVWXYZ,10H1234567890)
        WRITE(41,16)
16 FORMAT(" LABEL",5H UNIT,* 41*)
```

Positional editing - The T and X descriptors specify the position at which the next character will be transmitted to or from the record.

An X edit descriptor specifies a position beyond the current position.

T edit descriptors may specify a character position in either direction from the current position. This allows portions of a record to be read more than once, possibly with different editing.

T or X edit descriptors may cause a character already in the record to be replaced. During transmission to the record, undefined positions are filled with blanks. The result is as if the entire record were initially filled with blank characters. On output, an X descriptor that specifies a move to position c causes the length of the record to be at least $c-1$ characters. T edit descriptors by themselves do not affect the length of an output record. Positions beyond the last character of the record may be specified if no characters are to be transmitted from such positions.

T editing

The Tc edit descriptor indicates that the transmission of the next character to or from a record is to occur at the c th character position.

X editing

During transmission from a record, the bX edit descriptor causes the skipping of b character positions following and including the current character position. During transmission to a record, blank characters are placed into b character positions beginning with the current character position. In both cases, the record becomes positioned to the first character following the last character processed.

Example:

EXECUTION OF -

```
PRINT 12345
12345  FORMAT(1X,'ONE',17X,*FIVE*,T5,'TWO',8X,4HFOUR,T9,'T',*HR*,'E',1HE)
```

RESULTS IN THE PRINTING OF -

POSITION	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
CONTENT	O	N	E		T	W	0		T	H	R	E	E		F	O	U	R		F	I	V	E

(Note that the first character output is used for vertical spacing control and is not printed.)

Slash editing - The slash edit descriptor indicates the end of a record. During transmission from a file, the remaining portion of any current record is skipped and the file is positioned before the next record. If there is no current record, the file is positioned after the next record. During transmission to a file, an empty record is written as the last record of the file. Thus, an empty record may be written on output and an entire record may be skipped on input.

Slash edit descriptor processing of adjacent records may be specified by the appearance of as many consecutive slashes (optionally separated by commas) or by preceding a single slash with a *b* whose value is equal to the number of records to be processed.

Examples:

```
PRINT 39
39  FORMAT(*1LINE 1*,/, * LINE 2*/" LINE 3"///7H LINE 6)
      READ(99,42) RECORD3
42  FORMAT(2/,...)
```

Colon editing - When encountered in a format specification, a colon edit descriptor terminates the formatted transfer of data if no input/output list items remain to be processed. If there are unprocessed input/output list items remaining, the colon edit descriptor has no effect on format control. Termination of format control by a colon edit descriptor causes the record being processed to become the preceding record.

The ANSI FORTRAN Standard does not provide for colon editing.

P editing - A scale factor is specified by a P edit descriptor of the form *kP*, where *k* is an optionally signed integer constant called the *scale factor*. *kP* represents 10^k as a multiplier.

The scale factor is zero at the beginning of each input/output statement. It applies to all subsequently interpreted F, E, D, and G edit descriptors until another scale factor is encountered and established. Note that reversion of format control does not affect the established scale factor.

The scale factor, k , affects editing in the following manner:

- With F, E, D, and G input editing (provided that no exponent exists in the field) and with F output editing, the scale factor causes the externally represented number to correspond to the internally represented number multiplied by ten to the k th power.
- On input with F, E, D, and G editing, the scale factor has no effect if there is an exponent in the field.
- On output with E and D editing, the basic real constant part of the quantity to be produced is multiplied by the k th power of ten and the exponent is reduced by k .
- On output with G editing, the effect of the scale factor is suspended unless the magnitude of the datum to be edited requires the use of E editing. In this case, the scale factor has the same effect as with E output editing.

Examples:

Input field	9876.54	98.7654E2	9876.54	987.654	.864786D-4	86.4786E2
FORMAT statement	FORMAT (2PF8.3, -2PE9.4, F9.4, 0PG9.4, D9.4, -2PE9.4)					
Internal representation	98.7654	9876.54	987654.	987.654	.0000864786	8647.86
Internal representation	9.87654	9876.54	9876.54	987.654	864.786	8647.86
FORMAT statement	FORMAT (2PF12.2, -2PE12.4, F12.4, 1PG12.2, D12.4, -2PE12.4)					
Output field	987.65	.0099E+06	98.7654	9.88E+02	8.6479D+02	.0086E+06

Numeric editing - The I, F, E, D, and G edit descriptors specify input/output editing of integer, real, double-precision, and complex data. The following general rules apply:

- On input, leading blanks are not significant. Plus signs may be omitted. A blank character is interpreted as the character zero. A field of all blanks is considered to be zero.
- On input with F, E, D, and G editing, a decimal point appearing in the input field overrides that portion of an edit descriptor specifying the decimal point location. The input field may have more digits than are used in approximating the value of the datum.
- On output, the representation of a positive or zero internal value in the field is prefixed with blank characters. The representation of a negative internal value in the field is prefixed with blank characters followed by a minus.
- On output, the representation is right-justified in the field. If the number of characters produced by the editing is smaller than the field width, leading blanks will be inserted in the field.
- On output, if the number of characters exceeds the field width, the entire field is filled with asterisks.

Integer editing

The Iw edit descriptor indicates that the field to be edited occupies w positions. The specified input/output list item must be of type integer. On input, the specified list item becomes defined with an integer datum. On output, the specified list item must be defined with an integer datum.

In the input field, the character string must be in the form of an optionally signed integer constant. Leading blanks in the input field are ignored.

The output field for the Iw edit descriptor consists of zero or more leading blanks followed by a minus if the value of the internal datum is negative, followed by the magnitude of the internal value in the form of an unsigned integer constant without leading zeros.

An integer constant consists of at least one digit, which may be zero.

Examples:

The record at unit 94 contains the following characters -

POSITION	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
CONTENT	3	7		4	4	-	6	9	7	1	2		2	2	4	2	6	-	9	6		

EXECUTION OF -

```
READ(94,102) (MATRIX(I),I=1,7),INDEX  
102 FORMAT(713,11)  
PRINT 103,INDEX,(MATRIX(I),I=7,1,-1)  
103 FORMAT(12,713)
```

RESULTS IN -

POSITION	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
CONTENT	6	-	9	4	2	6		2	2	7	1	2	-	6	9	4	4		3	7		

EXECUTION OF -

```
PROGRAM IEDIT  
PARAMETER (N=6)  
DIMENSION M(N)  
  
M(1)=1  
DO 1 I=2,N  
1 M(I)=2*M(I-1)  
WRITE(66,2)M  
PRINT 2,M  
2 FORMAT(6(13))  
END
```

RESULTS IN -

POSITION	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
PRINTED	1		2		4		8		1	6		3	2					
RECORDED (unit 66)		1		2		4		8		1	6		3	2				

Real, Double-precision, and complex editing

The F, E, D, and G edit descriptors specify editing for real-double-precision, and complex data. Input/output list items corresponding to F, E, or G edit descriptors must be real or complex. The D edit descriptor is applied only to double-precision input/output list items. An input list item becomes defined with a datum whose type is the same as that of the list item.

F editing - The Fw.d edit descriptor indicates that the field occupies w positions, the fractional part of which consists of d digits.

The input field consists of an optional sign followed by a string of digits optionally containing a decimal point. This basic form may be followed by an exponent of 10 having one of the following forms:

- Signed integer constant.
 - E followed by an optionally signed integer constant.
 - D followed by an optionally signed integer constant.

An exponent containing a D is processed identically to an exponent containing an E.

The output field consists of blanks, if necessary, followed by a minus if the internal value is negative, followed by a string of digits that contains a decimal point and represents the magnitude of the internal value. This representation is modified by the established scale factor and is rounded to d fractional digits. If the output field value is less than one, a single zero is written immediately to the left of the decimal point, space permitting. If the output field value is zero and d is zero, a single zero is written. In no other cases are leading zeros written.

Examples:

<u>Internal representation</u>	<u>F edit descriptor</u>	<u>Output field positions</u>
		1 2 3 4 5 6 7 8 9 10
3.1415926	F10.5	3 . 1 4 1 5 9
-3.1415926	F7.4	- 3 . 1 4 1 6
747	F4.0	7 4 7 .
0	F8.6	0 . 0 0 0 0 0 0
0	F8.5	0 . 0 0 0 0 0
0	F7.6	1 . 0 0 0 0 0 0

E editing - The $Ew.d$ edit descriptor indicates that the external field occupies w positions, the fractional part of which consist of d digits.

The form of the input field is the same as for F editing.

The form of the output field for a scale factor of zero is

$[-][0].x_1 x_2 \dots x_{d-1} x_d exp$

where $x_1 x_2 \dots x_d$ are the d most significant digits of the rounded datum, and

exp is a decimal exponent of one of the following forms:

Absolute value of exponent exp	Output form of exponent (y is a digit)
$exp = 0$	E+00
$0 < exp < 100$	E±y ₁ y ₂
$100 < exp < 1,000$	±y ₁ y ₂ y ₃
$ exp \geq 1,000$	±y ₁ y ₂ y ₃ y ₄

An $|exp| \leq 1,000$ causes the entire field to be shifted left one position to provide for y_4 . Unless blank, the initial character of the field replaces the last character of the previous field or is lost if no previous field exists. Appropriate specification of field width w circumvents this problem.

The ANSI FORTRAN Standard does not provide for more than three digits in the output form of exponent.

The scale factor k controls decimal normalization. If $-d < k \leq 0$, there are $|k|$ leading zeros and $d - |k|$ significant digits after the decimal point. If $0 < k < (d+2)$, there are k significant digits to the left of the decimal point and $d-k+1$ significant digits to the right of the decimal point. Other values of k are not permitted.

Examples:

Input field positions												E edit descriptor	Internal representation
1	2	3	4	5	6	7	8	9	10	11	12		
+	1	0	4	8	5	7	5	.	7	5		E11.2	1048575.75
-	1	0	4	8	5	7	5	.	7	5		E11.0	-1048575.75
								3	8			E11.11	.00000000038
			1	.	5	9	2	E	3			E12.3	1592.
6	5	5	3	6	E	-	5					E8.3	.00065536
6	5	5	3	6	.	E	-	5				E9.3	.65536
-	3	2	.	7	6	8	D	0	4			E10.3	-327680

Internal representation	E edit descriptor	Output field positions										
		1	2	3	4	5	6	7	8	9	10	11
365.26	E10.2			0	.	3	7	E	+	0	3	
-365.26	E11.5		-	.	3	6	5	2	6	E	+	0 3
.000000099	E11.3			0	.	9	9	0	E	-	0	7

D editing - D editing is identical to E editing with the following two exceptions:

- The list item corresponding to a D edit descriptor must be of type double precision, and
- The output form of the exponent begins with the letter D instead of the letter E when $0 \leq |exp| < 100$.

Examples:

Input field positions												D edit descriptor	Internal representation
1	2	3	4	5	6	7	8	9	10	11	12		
-	9	2	9	6	1	2	.	7	9	1		D11.3	-929612.791
8	2	4	E	0								D5.0	824.
8	2	4	.									D4.0	824.
-	.	7	5									D7.4	-.75
1	.	8	7	2	1	4	1	9	6	7		D12.9	1.872141967
4	9	.	7	1	D	1	0					D10.2	497100000000.
8	2	4										D3.3	.824

Internal representation	D edit descriptor	Output field positions
		1 2 3 4 5 6 7 8 9 10 11 12
.793351	D11.6	. 7 9 3 3 5 1 D + 0 0
-7607.	D12.3	- 0 . 7 6 1 D + 0 4
1.	D12.1	0 . 1 D + 0 1

G editing - The $Gw.d$ edit descriptor indicates that the field occupies w positions with d significant digits.

G input editing is the same as F input editing.

Representation in the output field depends on the magnitude of the datum being edited. If N is the magnitude of the internal datum, its value determines the editing as follows:

Magnitude of datum	Equivalent edit descriptors
$0.1 \leq N < 1$	$F(w-4).d,4X$
$1 \leq N < 10$	$F(w-4).(d-1),4X$
.	.
.	.
.	.
$10^{d-2} \leq N < 10^{d-1}$	$F(w-4).1,4X$
$10^{d-1} \leq N < 10^d$	$F(w-4).0,4X$
$N < 0.1 \text{ or } N > 10^d$	$kP,EW.d$

where k is the scale factor in effect.

The scale factor is effective only if the magnitude of the datum exceeds the range for effective F editing.

Examples:

Input field positions												G edit descriptor	Internal representation
1	2	3	4	5	6	7	8	9	10	11	12		
6	2	9										G5.1	6290.
-	.	6	2	9	0	0	0	0				G10.2	- .629
+	8	7	8	.	4	9	2	1				G9.4	878.4921
4	7	2	1	.	0	E	-	2				G12.1	47.21
7	2	D	1	0								G5.0	720000000000.

Internal representation	G edit descriptor	Output field positions											
		1	2	3	4	5	6	7	8	9	10	11	12
-324.876	G12.6	-	3	2	4	.	8	7	6				
.487295343397	G10.5	.	4	8	7	3	0						
-72.59	G10.3	-	7	2	.	6							
.000000000019	G12.2							.	1	9	E	-	1
.000000000019	G9.1							.	2	E	-	1	0
100.	G12.2							.	1	E	+	0	3
100.0001	G12.2	.	1	0	0	0	0	0	1	E	+	0	3

Complex editing - A complex datum consists of a pair of separate real data. Its editing must be specified by two successively interpreted F, E, or G edit descriptors. The first of the edit descriptors specifies editing for the real part; the second for the imaginary part. The two edit descriptors may differ. Note that nonrepeatable edit descriptors may appear between two successive F, E, or G edit descriptors.

O editing - The Ow edit descriptor indicates the processing of a list item of type integer, real, Hollerith, Boolean, or logical and a field width of w positions.

The ANSI FORTRAN Standard does not provide for O editing.

On input, the field must contain a string of from one to twenty-two octal digits representing a precise binary value to be stored into the list item. This value is right-justified in the list item if fewer than twenty-two octal digits are contained in the field. Unspecified bit-positions are cleared to zero. A blank field is considered a field containing all zeros. If the first nonblank character in the field is a minus, the one's complement of the value is stored.

On output, the internal representation of the list item is converted to octal and the rightmost w octal digits are right-justified in the field. If the field is larger than twenty-two positions, leading blank characters are output.

Z editing - The Zw edit descriptor indicates the processing of a list item of type integer, real, Hollerith, Boolean or logical and a field width of w positions.

The ANSI FORTRAN Standard does not provide for Z editing.

On input, the field must contain a string of from one to sixteen hexadecimal characters representing a zero or positive integral value (in the base-16 number system) to be stored into the list item. This value is right-justified in the list item if fewer than 16 hexadecimal characters are contained in the field; leading zeros are assumed. A blank field is assumed a field of all zeros. If the first nonblank character in the field is a minus, the one's complement of the value is stored.

On output, the internal representation of the list item is converted to a zero or positive hexadecimal value and the rightmost w digits are right-justified in the field. If the field is larger than 16 positions, leading blank characters are output.

L editing - The Lw edit descriptor indicates the processing of a logical list item and an input or output field width of w positions. The specified input/output list item must be of type logical. On input, the list item becomes defined with a logical datum. On output, the list item must be defined with a logical datum.

The input field consists of a T for true or an F for false optionally followed by additional characters. The field may contain leading blank characters.

If $w < 5$, the output field consists of $w-1$ blanks followed by a T or F depending on the value of the internal datum. If $w \geq 5$, the output field contains the letters TRUE or FALSE preceded by $w-4$ or $w-5$ blank characters, respectively.

Examples:

Input field positions												L edit descriptor	Internal representation
1	2	3	4	5	6	7	8	9	10	11	12		
T									L1			(true)	
T	R	U	E						L4			(true)	
F									L3			(false)	
F	A	L	S	E					L12			(false)	
T	1	2	3						L7			(true)	
F	A	B	C						L9			(false)	
					T				L12			(true)	
					F				L12			(false)	

Internal representation	L edit descriptor	Output field positions												
		1	2	3	4	5	6	7	8	9	10	11	12	
(true)	L6		T	R	U	E								
(false)	L12									F	A	L	S	E
(true)	L10									T	R	U	E	
(false)	L1				F									
(true)	L1			T										
(false)	L3				F									

A editing - The Aw edit descriptor is used with an input/output list item of type logical, integer, real, or complex. On input, the input list item becomes defined with Hollerith data. On output, the output list item must be defined with Hollerith data. Integer, real, and logical input/output list items can contain up to eight Hollerith characters; complex up to sixteen. w specifies a field of one to eight characters.

If the specified field width w for A input is greater than or equal to eight, the rightmost eight characters of the input field form the internal representation. If the specified field width w is less than eight, the w characters from the input field are left-justified with 8-w trailing blank characters added to form the internal representation.

If the specified field width w for A output is greater than eight, the output field consists of $w-8$ blanks followed by the eight characters from the internal representation. If the specified field width w is less than or equal to eight, the output field consists of the left-most w characters from the internal representation.

Input/output list items of type complex may contain up to sixteen characters in two storage units (computer words). Two A edit descriptors are required when eight or more characters are to be transferred. In this case, both are applied to a single input/output list item; the first to the first storage unit, the second to the second storage unit.

Examples:

Input field positions												Item type	A edit descriptor(s)	Internal representation
1	2	3	4	5	6	7	8	9	10	11	12			
A	B	C	D	E	F	G	H	I	J	K	L	Integer	A8	8HABCDEFGH
A	B			1	2							Real	A7	8HAB 12
I	N	D	E	X	6		Complex	A8,A3	16HINDEX....6
R	T	C										Integer	A3	8HRTC

Internal representation	Item type	A edit descriptor	Output field positions								
			1	2	3	4	5	6	7	8	9
8HABCDEFGH	Integer	A8	A	B	C	D	E	F	G	H	
8HABCDEFGH	Real	A9	A	B	C	D	E	F	G	H	
16HA1B2C3D4E5	Complex	A8,A1	A	1	B	2	C	3	D	4	E
8HA-FORMAT	Integer	A3	A	-	F						

R editing - The Rw edit descriptor is used with an input/output list item of type integer, real, or complex. On input, the input list item will become defined with w characters of Hollerith data. On output, the output list item must be defined with w characters of Hollerith data. Rw edit descriptor actions are identical to those of the Aw edit descriptor with two exceptions.

- Characters in an incompletely filled input list item are right justified with the remainder of that list item containing binary zeros.
- Partial output of an output list item's characters is from its rightmost character positions.

The ANSI FORTRAN Standard does not provide for an R edit descriptor.

Examples:

Input field positions												Item type	R edit descriptor	Internal representation
1	2	3	4	5	6	7	8	9	10	11	12			
A	B	C	D	E	F	G	H	I	J	K	L	Integer	R	8HABCDEFGH
A	B			1	2							Real	R7	7RAB 12
R	T	C										Integer	R3	3RRTC

Internal representation	Item type	R edit descriptor	Output field positions								
			1	2	3	4	5	6	7	8	9
8HABCDEFGH	Integer	R8	A	B	C	D	E	F	G	H	
8HABCDEFGH	Real	R9	A	B	C	D	E	F	G	H	
8RA-FORMAT	Integer	R6	F	O	R	M	A	T			

3.6.8 BUFFERED INPUT/OUTPUT OPERATIONS

The ANSI FORTRAN Standard does not provide for buffered input/output operations or for the BUFFER IN or BUFFER OUT statements.

Buffered input/output operations differ from those described in 3.6.6 in several respects. First, the two statements that initiate a transfer of data allow the subsequent execution sequence to proceed simultaneously with the actual transfer. Next, certain utility functions may be referenced to cause a delay in an execution sequence pending completion of a buffered input/output operation, and to determine certain characteristics of that operation upon its termination. Finally, the amount of data is specified in terms of CRAY-1 computer words with no consideration given the type or format of information contained.

The BUFFER IN and BUFFER OUT statements

The two statements used for buffered input/output operations are the BUFFER IN and BUFFER OUT statements. Their forms are

BUFFER IN	(<i>u,m</i>)	(<i>bloc,eloc</i>)
BUFFER OUT	(<i>u,m</i>)	(<i>bloc,eloc</i>)

where: *u* is a unit specifier expressed as an integer or as a Hollerith expression of up to eight characters.

m is a mode specifier expressed as an integer expression indicating full record processing if zero or greater and partial record processing if less than zero.

bloc is the symbolic name of that variable or array element marking the beginning location of the buffered I/O transfer, and

eloc is the symbolic name of that variable or array element marking the ending location of the buffered I/O transfer.

BUFFER IN causes information to be read; BUFFER OUT causes information to be written. Execution of either statement attempts initiating the transfer of (*eloc - bloc + 1*) CRAY-1 computer words between the current record at unit *u* and those contiguous memory locations beginning with *bloc* and concluding with *eloc*. If unit *u* is completing a buffered input/output operation initiated earlier, the execution sequence is delayed at the onset of BUFFER IN or BUFFER OUT statement execution until that earlier operation terminates. Upon termination, execution of the BUFFER IN or BUFFER OUT statement completes as though no delay occurred.

In determining the number of computer words to be transferred, consideration must be given the data type of the symbolic name used for *eloc*. If this name is of type double precision or complex, the location of the first word in its two-word form of representation will mark the ending location of the data transfer.

The mode specifier, *m*, controls the position of the record at unit *u* after the data transfer has taken place. If the value of *m* is greater than or equal to zero, full record processing is called for. The record position following this mode of transfer is always between the current record (the record to or from which the transfer occurred) and the next record. For a value of *m* less than zero, partial record processing takes place. In a BUFFER IN statement, this specifies that the record being transferred from will be positioned ready to transfer its (*n+1*)th word if the *n*th word was the last transferred. In a BUFFER OUT statement, this specifies that the

record will be left positioned to receive additional words. A special buffered output operation concludes a series of partial record buffered output transfers. In this case, a BUFFER OUT statement is used in which *bloc* equals *eloc+1* to produce a zero word transfer that also concludes the record being created.

File and record positioning for buffered input/output operations are as described for non-buffered input/output operations (see 3.6.6).

Example:

A BUFFER IN statement initiates the transfer of 1000 words from unit 32. Computation then proceeds on data not related to that being transferred. A second BUFFER IN statement is encountered upon completion of this computation, causes a delay in the execution sequence until the last of the 1000 words has been received, then initiates a transfer of another 500 words from unit 32. While these words are being transferred, the execution sequence proceeds. A BUFFER OUT statement initiates the transfer of the first 1000 words to unit 22. Full record processing is called for by the mode specifier being zero in all cases.

```
PROGRAM XFR
PARAMETER(INUNIT=32)
DIMENSION A(1000), B(2,10,100), C(500)
BUFFER IN(INUNIT,0) (A(1),A(1000))
DO 10 I=1,100
10  B(1,1,I)=B(1,1,I) + B(2,1,I)
BUFFER IN(INUNIT,0) (C(1),C(500))
BUFFER OUT(22,0) (A(1),A(1000))
:
END
```

The UNIT function

Once a BUFFER IN or BUFFER OUT statement has been executed, the normal execution sequence continues concurrent with the actual transfer of data. If the utility function UNIT is referenced in this execution sequence, continuation of the sequence is delayed pending completion of the transfer at that unit designated by its single integer argument.

Upon completion of the transfer, the UNIT function provides one of the following real data type values to the expression in which it is referenced:

- -2.0 to indicate successful completion of a partial record read operation (BUFFER IN with $m < 0$) without encountering the end of the current record,
- -1.0 to indicate successful completion of all other transfers,
- 0.0 to indicate reading of an end-of-file or end-of-data record,
- 1.0 to indicate occurrence of a disk parity error during reading, or
- 2.0 to indicate other disk malfunctions during reading or writing.

Example:

```
PROGRAM TESTUNIT
DIMENSION M(200,5)
10 BUFFER IN (32,0) (M(1,1),M(200,5))
    IF (UNIT(32))11,13,13
11 D012 J=1,5
    D012 I=1,200
12 M(I,J)=-M(I,J)
    BUFFER OUT (22,0) (M(1,1),M(200,5))
    IF (UNIT(22))10,13,13
13 END
```

The LENGTH function

If the utility function LENGTH is referenced while a buffered input/output operation is in progress at that unit designated by its single integer argument, the execution sequence will be delayed until the transfer is complete. LENGTH will then provide to the expression in which it is referenced an integer value reflecting the number of CRAY-1 computer words successfully transferred. This value is zero if an end-of-file or end-of-data record is read.

Example:

```
PROGRAM PGM
DIMENSION V(16384)
10 BUFFER IN (32,-1) (V(1),V(16384))
X= UNIT(32)
K= LENGTH(32)
IF(X)11,14,14
11 DO 12 I=1,K,1
12 IF(V(I).EQ.'KEY') GO TO 13
IF(X.EQ.-2.0) 10,14
13 .
.
.
14 END
```

3.6.9 FORMATTED DATA ASSIGNMENT

The ANSI FORTRAN Standard does not provide for formatted data assignment operations.

Formatted data assignment operations define entities by transferring data between input/output list items and internal records. Like other assignment statements, formatted data assignment statements only perform internal data transfers. Like formatted input/output statements, formatted data assignment statements specify an input/output list and invoke format control during their operations.

The two formatted data assignment statements are ENCODE and DECODE.

ENCODE and DECODE statements

The forms of the ENCODE and DECODE statements are:

ENCODE (n,f,dent) [iolist]
DECODE (n,f,sent) [iolist]

where *n* is the number of characters to be processed, must not exceed 152, and is specified as a non-zero integer expression;

f is a format identifier specified as the name of an array containing a format specification, the statement label of a FORMAT statement, or an integer variable containing the statement label of a FORMAT statement;

- dent* is the symbolic name of a destination variable, array element, or array that contains the *n* characters processed by the ENCODE statement;
- sent* is the symbolic name of a source variable, array element, or array from which the *n* characters to be processed by the DECODE statement originate; and
- iolist* is an input/output list specified the same as for formatted input/output statements (see 3.6.6), contains input list items in a DECODE statement, and contains output list items in an ENCODE statement.

The ENCODE statement

The ENCODE statement produces a sequence of *n* characters from values contained in the output list items specified in *iolist* under control of the format specification identified by *f*. The character sequence is stored into a variable, array element or array identified by *dent*.

The DECODE statement

The DECODE statement processes a sequence of *n* characters contained in the variable, array element, or array identified by *sent* under control of the format specification identified by *f*. The resulting values define the input list items specified in *iolist*.

3.6.10 THE MAIN PROGRAM

A *main program* is a program unit that does not contain a FUNCTION, SUBROUTINE, or BLOCK DATA statement. An optional PROGRAM statement may be the first statement of a main program. There must be exactly one main program in an executable program. Execution of an executable program begins with the first executable statement of the main program.

The PROGRAM statement

The form of a PROGRAM statement is

PROGRAM *pgm* [([*h₁h₂...h_n*])]

where *pgm* is the symbolic name of the main program in which the PROGRAM statement appears, and

h₁h₂...h_n is a character string of *n* characters.

Use of the PROGRAM statement is optional and, if used, it must be the first statement of the main program.

The symbolic name *pgm* is global and must not be the same as the name of an external procedure, block data subprogram, or common block in the same executable program. The name *pgm* must not be the same as any local name in the main program. It may be followed by a parenthesized character string that has no effect on the executable program.

The ANSI FORTRAN Standard does not provide for a block data subprogram name or for the PROGRAM statement.

Main program restrictions

A main program may contain any statements other than BLOCK DATA, FUNCTION SUBROUTINE, or RETURN statements. It may contain a single PROGRAM statement. A main program must not be referenced from a subprogram or from itself.

Examples:

```
PROGRAM A1B2C3D4
PROGRAM X  (INPUT,OUTPUT)
PROGRAM MAIN
PROGRAM Z1234567
```

3.6.11 FUNCTIONS AND SUBROUTINES

Categories of functions and subroutines

Procedures - All functions and subroutines are *procedures*. The four categories of procedures are:

- Intrinsic functions,
- Statement functions,
- External functions, and
- Subroutines.

Intrinsic functions, statement functions, and external functions are generally referred to as *functions*. (Utility procedures are separately described in 3.6.12.)

External functions

The categories of *external functions* are:

- Function subprograms, which are program units specified in the FORTRAN language, and
- Non-FORTRAN function subprograms which are not specified in the FORTRAN language.

Subroutines

There are two categories of *subroutines*:

- Subroutine subprograms, which are program units specified in the FORTRAN language, and
- Non-FORTRAN subroutine subprograms which are not specified in the FORTRAN language.

Appendix F describes the method for programming non-FORTRAN function and subroutine subprograms using CAL, the CRAY-1 Assembly Language.

Referencing functions

A function is referenced in an expression and supplies a value to the expression. This value is the value of the function at the time that the expression containing its reference is evaluated.

An intrinsic function may be referenced in the main program or in any procedure subprogram of an executable program.

A statement function may be referenced only in the program unit in which the statement function statement appears.

An external function may be referenced by function or entry name within any other procedure subprogram or the main program of the executable program. A subprogram must not reference itself, either directly or indirectly.

Form of a function reference - A function reference is used to reference an intrinsic function, statement function, or external function. The form of a function reference is

fun ([a [,a] ...])

where *fun* is the symbolic name of a function or of an entry to a function, and

a is an actual argument.

The type of the result of a statement function or external function reference is the same as the type of the function name and is specified in the same manner as for variables and arrays. The type of the result of each intrinsic function is specified in Appendix B. The type of each actual argument α and the number of actual arguments specified in a function reference must agree with the (dummy) arguments defined in the specification of the function being referenced.

Execution of function references - A function reference appears only as a primary in an arithmetic or logical expression. Execution of a function reference in an expression causes the evaluation of the function identified by *fun*.

Return of control from a referenced function completes execution of the function reference. The value of the function is then available to the expression containing the reference and being evaluated. Actual and dummy arguments associated during a function reference remain associated afterwards.

Examples:

<u>REFERENCE</u>	<u>COMMENTS</u>
SIN(ALPHA)	Intrinsic function reference
FUNCTION(INT,REAL,DBL,COMP,LOG)	Statement or external function reference
RANF()	Intrinsic function reference

Intrinsic functions

Intrinsic functions are prespecified and have special meanings. Their specific names, function definitions, and types of arguments and results appear in Appendix B. In this table, integer, real, double-precision and complex data types are denoted by the letter I, R, D and C, respectively.

The ANSI FORTRAN Standard "Basic External Functions" are incorporated into the list of intrinsic functions presented in Appendix B, and are not separately identified.

Referencing intrinsic functions - An intrinsic function is referenced by using its name, *fun*, as a primary in an expression. An intrinsic function reference produces results dependent on the values of the actual arguments. The resulting value is available to the expression that contains the function reference.

The actual arguments that constitute the argument list must agree in type, number and order with those described in Appendix B and may be any expression of the specified type.

Intrinsic function restrictions - Arguments that would cause an undefinable result or a result that would exceed the maximum numeric representation permitted cause the result of the function to become undefined.

Examples:

AMEDIAN = (AMIN1(A,B,C,D)+AMAX1(A,B,C,D))/2.0

T = SIN(THETA)/COS(THETA)

T = TAN(THETA)

Statement functions

A statement function is a procedure specified by a single statement that is similar in form to an arithmetic or logical assignment statement. This is a statement function definition statement and can only appear after the specification statements and before the first executable statement of the program unit in which it is referenced. Since it is not a part of the normal execution sequence, a statement function definition statement is classified as a non-executable statement.

Form of a statement function definition statement - The form of a statement function definition statement is

fun ([d[,d]...]) = e

where *fun* is the symbolic name of the statement function,
d is a statement function dummy argument, and
e is an expression.

The relationship between *fun* and *e* must conform to the assignment rules in table 3-7. The type of the expression *e* may be different from the type of the statement function name *fun*.

Each *d* is a variable name called a *statement function dummy argument*. The names of variables that appear as dummy arguments of a statement function have a scope of that statement only. A given symbolic name may appear only once in a single dummy argument list. Statement function dummy arguments serve only to indicate the order, number, and type of arguments for a single statement function. The same symbolic names may be used to identify dummy arguments of the same type in a different statement function definition statement and variables of the same type appearing elsewhere in the program unit including dummy arguments of a subprogram. They must not be used to identify any other entity in the program unit except a common block.

Each primary of the expression e must be:

- A constant,
- The symbolic name of a constant,
- A statement function dummy argument referenced as a variable,
- A reference to a variable used elsewhere in the same program unit,
- An intrinsic function reference,
- A reference to a statement function for which the statement function definition statement appears in preceding lines of the program unit,
- An external function reference, or
- An expression enclosed in parentheses.

If a statement function dummy argument name is the same as the name of another entity, the appearance of that name in the expression portion of a statement function definition statement is a reference to the statement function dummy argument. A dummy argument that appears in a FUNCTION or SUBROUTINE statement may be referenced in the expression of a statement function statement within that subprogram.

Examples:

ROOT(A,B,C,SIGN)=(-B+SIGN*SQRT(B**2-4.*A*C))/(2.*A)

DISCRIM(A,B,C)= B**2-4.*A*C

CIRCUM(R)=6.2831852*R

AREA(R)=3.1415926*R**2

VOL(R)=4.1887901*R**3

VOL()=4.1887901*R**3 (where R appears elsewhere in the same program unit)

Referencing statement functions - A statement function is referenced by using its function reference as a primary in an expression.

Execution of a statement function reference results in:

- Evaluation of actual arguments that are expressions,
- Association of actual arguments with corresponding dummy arguments, and
- Evaluation of the expression e .

The resulting value is available to the expression that contains the function reference.

The actual arguments, which constitute the actual argument list, must agree in order, number, and type with the corresponding dummy arguments. An actual argument in a statement function reference may be any expression involving one or more operands.

When a statement function reference is executed, its actual arguments must be defined.

Statement function restrictions - A statement function may be referenced only in the program unit that contains the statement function definition statement.

A statement function definition statement can reference another statement function that precedes the reference. The symbolic name used to identify a statement function must not appear as a symbolic name in any specification statement other than a type statement (to specify the type of the function) or as the name of a common block in the same program unit.

An external function reference in the expression of a statement function definition statement must not cause a dummy argument of the statement function to become undefined or redefined.

The symbolic name of a statement function is a local name and must not be the same as the name of any other entity in the program unit other than that of a common block. The symbolic name of a statement function may not be an actual argument and must not appear in an EXTERNAL statement.

A statement function definition statement in a function subprogram must not reference that function subprogram.

Examples:

```
DISCRIM(A,B,C)=B**2-4.*A*C
ROOT(A,B,C,SIGN)=(-B+SIGN*SQRT(DSCRIM(A,B,C))) /(2.*A)
:
IF(DSCRIM(TERM1,TERM2,TERM3))25,15,15
15 ROOT1 = ROOT(TERM1,TERM2,TERM3,+1.0)
      ROOT2 = ROOT(TERM1,TERM2,TERM3,-1.0)
25 END
:
```

```

PARAMETER(M=100,N=4,PI=3.1415926)
DIMENSION RADTAB(M,N)
VOL(R) = (4.*PI*R**3)/3.
AREA(R) = PI*R**2
CIRCUM(R) = 2.*PI*R
.
.
DO 20,I=1,M
IF(RADTAB(I,1))30,20,10
10 RADTAB(I,2) = CIRCUM(RADTAB(I,1))
RADTAB(I,3) = AREA(RADTAB(I,1))
RADTAB(I,4) = VOL(RADTAB(I,1))
20 CONTINUE
.
.
30 END

```

External functions

An external function is specified external to the program unit that references it. An external function is a procedure specified by a function subprogram or some other means.

Function subprograms - A function subprogram has a FUNCTION statement as its first statement. It may also contain one or more ENTRY statements.

FUNCTION statement forms - The forms of a FUNCTION statement are:

FUNCTION	<i>fun</i> [([d[,d]...])]
INTEGER FUNCTION	<i>fun</i> [([d[,d]...])]
REAL FUNCTION	<i>fun</i> [([d[,d]...])]
DOUBLE PRECISION FUNCTION	<i>fun</i> [([d[,d]...])]
DOUBLE FUNCTION	<i>fun</i> [([d[,d]...])]
COMPLEX FUNCTION	<i>fun</i> [([d[,d]...])]
LOGICAL FUNCTION	<i>fun</i> [([d[,d]...])]

where: *fun* is the symbolic name of the function subprogram in which the FUNCTION statement appears and
d is a dummy argument representing a variable, array, or external procedure name.

The symbolic name of a function subprogram must appear as a variable name in the function subprogram. During every execution of the subprogram, this variable must become defined and, once defined, may be referenced or become redefined. The value of the function is the value of this variable when a RETURN or END statement is executed in the subprogram. The type of this value is implicit to the function name, *fun*, unless INTEGER, REAL, DOUBLE PRECISION or DOUBLE, COMPLEX, or LOGICAL is specified to cause its being overridden.

The ANSI FORTRAN Standard does not provide for the DOUBLE FUNCTION statement.

A function subprogram may define one or more of its dummy arguments to return values in addition to the value of the function.

Referencing external functions - An external function is referenced by using its reference as a primary in an expression. A reference to an entry in a function may be similarly used.

Execution of external function references - Execution of an external function reference or a reference to an external function entry results in:

- The evaluation of actual arguments that are expressions,
- The association of actual arguments with the corresponding dummy arguments, and
- The actions specified by the referenced function.

The type of the function or function entry name in the reference must be the same as the type of the function or function entry name in the referenced function.

Actual arguments for external functions - The actual arguments in an external function reference must agree in order, number, and type with the corresponding dummy arguments in the referenced function or function entry. The use of a subroutine name as an actual argument is an exception to the rule requiring agreement of type because subroutine names do not have a type.

An actual argument in an external function reference must be one of the following:

- An expression,
- An array name,
- An array element name,
- An intrinsic function name, or
- An external procedure name.

The ANSI FORTRAN Standard does not provide for intrinsic function names as actual arguments.

Note that an actual argument in a function reference may be a dummy argument that appears in a dummy argument list within the subprogram containing that reference.

Function subprogram restrictions -- A function subprogram may contain any statement other than a BLOCK DATA, SUBROUTINE, PROGRAM, or a second FUNCTION statement.

The symbolic names of external functions and external function entries are global names and must not be the same as any other global name.

The symbolic name of a function specified by a FUNCTION statement must not appear in any other non-executable statement except for a type statement, and must only appear as a variable in executable statements.

If the type of a function is specified in a FUNCTION statement, the function name must not also appear in a type statement. (Redundant type specifications are not allowed.)

In a function subprogram, the symbolic name of a dummy argument is local and must not appear in an EQUIVALENCE, PARAMETER, DATA, or COMMON statement except as a common block name.

A function specified by a subprogram may be referenced within any other procedure subprogram or the main program of the executable program. A function subprogram must not directly or indirectly reference itself.

Examples:

```
PROGRAM MAIN
PARAMETER(M=6,N=1000,KEY=6123386684)
DIMENSION MATRIX(M,N)
READ 2,((MATRIX(I,J),I=1,M),J=1,N)
2 FORMAT(1I10)
IF(MATCH(MATRIX,M,N,KEY))9,9,3
3 PRINT 4,KEY
4 FORMAT(" THE VALUE ",1I10," EXISTS IN THE INPUT DATA.")
9 END

FUNCTION MATCH(INTTBL,K,L,ITEM)
DIMENSION INTTBL(K,L)
MATCH = 0
DO 1 I=1,K
DO 1 J=1,L
1 IF(INTTBL(I,J).EQ.ITEM)MATCH=MATCH+1
END
```

Subroutines

A subroutine is a procedure that may be specified by a subroutine subprogram or some other means. It is specified external to the program unit that references it.

One or more dummy arguments of a subroutine subprogram may become defined or redefined to return results. Note that entities specified in a COMMON statement in the subroutine may also be defined for this purpose.

Subroutine subprograms - A subroutine subprogram is a program unit that has a SUBROUTINE statement as its first statement. It may also contain one or more ENTRY statements.

SUBROUTINE statement form - The form of a SUBROUTINE statement is

SUBROUTINE *sub* [(*d* [, *d*] ...)]

where *sub* is the symbolic name of the subroutine, and

d is a dummy argument representing a variable name, an array name, or an external procedure name.

Subroutine reference - A subroutine is referenced by a CALL statement.

Form of a CALL statement - The form of a CALL statement is

CALL *sub* [(*a* [, *a*] ...)]

where *sub* is the symbolic name of a subroutine or a subroutine entry, and *a* is an actual argument.

Execution of a CALL statement - Execution of a CALL statement results in:

- The evaluation of actual arguments that are expressions;
- The association of actual arguments with the corresponding dummy arguments, and
- The actions specified by the referenced subroutine.

Return of control from the referenced subroutine to the first executable statement following the CALL statement completes the execution of the CALL statement. Actual and dummy arguments associated during CALL statement execution remain associated afterwards.

A subroutine may be referenced within any other procedure subprogram or the main program of the executable program. A subprogram must not directly or indirectly reference itself.

When a CALL statement is executed, the referenced subroutine must be one of the subroutines in the executable program.

Actual arguments for subroutines - The actual arguments in a subroutine reference must agree in order, number, and type with the corresponding dummy arguments in the dummy argument list of the referenced subroutine. The use of a subroutine name as an actual argument is permitted and is an exception to the rule requiring agreement of type.

An actual argument in a subroutine reference must be one of the following:

- An expression,
- An array name,
- An array element name,
- An intrinsic function name, or
- An external procedure name.

Note that an actual argument in a subroutine reference may be a dummy argument that appears in a dummy argument list within the subprogram containing the reference.

Subroutine subprogram restrictions - A subroutine subprogram may contain any statement other than a BLOCK DATA, FUNCTION, PROGRAM, or a second SUBROUTINE statement.

The symbolic names of subroutines and subroutine entries are global names and must not be the same as any other global name or any local name in the referencing program unit.

In a subroutine subprogram, the symbolic name of a dummy argument is local and must not appear in an EQUIVALENCE, PARAMETER, DATA, or COMMON statement. It can be the same as a common block name.

Example:

```
PROGRAM RKN
:
DIMENSION TABLE(1226)
:
CALL SORT(TABLE,1226)
:
END
:
```

```
      :  
SUBROUTINE SORT(VECTOR,LENGTH)  
DIMENSION VECTOR(LENGTH)  
5   K = -1  
DO 10 I=2,LENGTH,1  
IF(VECTOR(I).GE.VECTOR(I-1)) GO TO 10  
TEMP = VECTOR(I-1)  
VECTOR(I-1) = VECTOR(I)  
VECTOR(I) = TEMP  
K = 1  
10 CONTINUE  
IF(K)15,15,5  
15 END
```

ENTRY statements

The ENTRY statement is used only in a procedure subprogram to permit its being entered at any executable statement not within a DO-loop range. A procedure subprogram may contain one or more ENTRY statements following its FUNCTION or SUBROUTINE statement.

The ANSI FORTRAN Standard does not provide for ENTRY statements.

Form of an ENTRY statement - The form of an ENTRY statement is

ENTRY *en* [([*d*[,*d*]...])]

where *en* is a function or subroutine name used as an entry in the procedure subprogram, and

d is a variable, array, or dummy procedure name used as a dummy argument.

Referencing a procedure subprogram entry - Referencing an *en* in a function or subroutine subprogram is the same as referencing the function or subroutine subprogram name. Execution begins with the first executable statement following that ENTRY statement.

The order, number, and types of names appearing as dummy arguments in an ENTRY statement must agree with the actual arguments in any reference to that ENTRY statement. They need not agree with those specified in a FUNCTION, SUBROUTINE, or other ENTRY statement in the same subprogram. Agreement of type is not required where a dummy argument corresponds to an actual argument specifying a subroutine name.

Entry association in function subprograms - The function name *en* specified in an ENTRY statement in a function subprogram is associated with all variables in that subprogram which are associated with the function name appearing in the FUNCTION statement. When any one of these variables becomes defined, all associated variables and function names of the same type also become defined; those not of the same type becoming undefined. A function name appearing in a FUNCTION statement may differ in type from function names appearing in ENTRY statements in the same subprogram.

ENTRY statement restrictions - A function or subroutine name specified in an ENTRY statement cannot be the same as any name specified in PROGRAM, BLOCK DATA, FUNCTION, SUBROUTINE, or ENTRY statements in the same executable program.

The function name specified in an ENTRY statement must not appear as a variable in any statement preceding that ENTRY statement except for a type statement.

A name appearing as a dummy argument in an ENTRY statement cannot appear in an executable statement preceding that ENTRY statement unless it also appears in a FUNCTION, SUBROUTINE or ENTRY statement that precedes the executable statement.

In a subprogram, a dummy argument specified in an ENTRY statement cannot be referenced in a statement function statement unless it also appears as a dummy argument in the statement function statement or in a preceding FUNCTION, SUBROUTINE or ENTRY statement.

If a dummy argument name is referenced in an executable statement, it must also be specified in that FUNCTION, SUBROUTINE, or ENTRY statement referenced prior to execution of the executable statement.

RETURN statements

A RETURN statement causes return of control to the referencing program unit and may appear only in a function or subroutine subprogram.

Form of a RETURN statement - The form of a RETURN statement in a subprogram is

RETURN

Execution of a RETURN statement - Execution of a RETURN statement terminates a reference to a function or subroutine subprogram. Such subprograms may contain more than one RETURN statement. A subprogram need not contain a RETURN statement since execution of an END statement in a function or subroutine subprogram has the same effect as executing a RETURN statement. In the execution of an executable program, a function or subroutine subprogram must not be referenced twice without the execution of a RETURN or END statement in that procedure having intervened.

The value of function must be defined prior to its execution of a RETURN or END statement. Execution of a RETURN or END statement in a procedure subprogram causes return of control to the current referencing program unit.

Return of control to the referencing program unit completes execution of the CALL statement.

Note that if a named common block appears in the main program, the entities in the named common block do not become undefined upon execution of any RETURN or END statement in the executable program.

Arguments and common blocks

Arguments and common blocks provide a means of communication between a referencing program unit and a referenced procedure.

Data may be communicated to a statement or intrinsic function by an argument list. Data may be communicated to and from an external procedure by an argument list or by common blocks. Procedure names may be communicated to an external procedure only by an argument list.

A dummy argument appears in the argument list of a procedure. An actual argument appears in the argument list of a procedure reference. The number, type, and order of actual arguments must be the same as the number, type and order of dummy arguments in the procedure referenced.

Dummy arguments - Statement functions, function subprograms, and subroutine subprograms use dummy arguments to indicate the types of actual arguments and whether each is a single value, an array of values or a procedure. Note that statement function dummy arguments are limited to variables.

Each dummy argument is classified as a variable, array, or procedure. Dummy argument names may appear wherever an actual name of the same class and type may appear, except where explicitly prohibited.

Dummy argument names of type integer may appear as adjustable dimension declarators in dummy array declarators. Dummy argument names may not appear in EQUIVALENCE, DATA, or PARAMETER statements or in COMMON statements except as common block names. A dummy argument name must not be the same as the procedure name appearing in a FUNCTION, SUBROUTINE, or statement function statement in the same program unit.

Actual arguments - Actual arguments specify the entities that are to be associated with the dummy arguments of a referenced subroutine or function. An actual argument must not be the name of a statement function in the referencing program unit. Actual arguments may be constants and expressions involving operators if the associated dummy argument is a variable that is not defined during execution of the referenced external procedure.

The type of each actual argument must agree with the type of its associated dummy argument, except when the actual argument is a subroutine name.

Association of dummy and actual arguments - Upon execution of a function or subroutine reference, an association is established between the corresponding actual and dummy arguments. The first actual argument becomes associated with the first dummy argument, the second actual argument becomes associated with the second dummy argument, etc.

All appearances within a function or subroutine of a dummy argument become associated with the corresponding actual argument when a reference to that function or subroutine is executed.

A valid association occurs only if the type of the actual argument is the same as the type of the corresponding dummy argument. A subroutine name has no type and must be associated with a procedure.

If an actual argument is an expression, it is evaluated just before the association of arguments takes place.

If an actual argument is an array element name, its subscript is evaluated just before the association of arguments takes place. Note that the subscript value remains constant as long as that association of arguments persists, even if the subscript contains variables that are redefined during the association.

If an actual argument is an external procedure name, the procedure must be available at the time a reference to it is executed.

If an actual argument becomes associated with a dummy argument that appears in an adjustable dimension declarator, the actual argument must be defined with an integer value at the time the procedure is referenced.

A dummy argument is undefined if it is not currently associated with an actual argument. An adjustable array is undefined if the dummy argument array is not currently associated with an actual argument array or if any variable appearing in the adjustable array declarator is not currently associated with an actual argument or is not in a common block.

Argument association may be carried through more than one level of procedure reference. A valid association exists at the last level only if a valid association exists at all intermediate levels. Note that argument association endures between repeated references to a subprogram.

Variables as dummy arguments - A dummy argument that is a variable may be associated with an actual argument that is a variable, array element, or expression.

If the actual argument is a variable name or array element name, the associated dummy argument may be defined or redefined within the subprogram. A dummy argument must not be redefined within the subprogram if the associated actual argument is:

- A constant,
- The symbolic name of a constant,
- A function reference,
- An expression involving operators, or
- An expression enclosed in parentheses.

Arrays as dummy arguments - Within a program unit, the array declarator given for an array provides all array declarator information required for execution of the program unit. The number and size of dimensions in an actual array declarator may be different from the number and size of the dimensions in an associated dummy array declarator.

A dummy argument that is an array name may be associated with an actual argument that is either an array name or an array element name.

If the actual argument is an array name, the size of the dummy argument array must not exceed the size of the actual argument array, and each actual argument array element becomes associated with that dummy argument array element that has the same subscript value as the actual argument array element.

If the actual argument is an array element name, the size of the dummy argument array must not exceed the size of the actual argument array plus one minus the subscript value of the array element. When an actual argument is an array element name with a subscript value of p , the dummy argument array element with a subscript value of q becomes associated with the actual argument array element that has a subscript value of $p+q-1$.

Procedures as dummy arguments - A dummy argument that is a procedure may be associated only with an actual argument that is a procedure.

If a dummy argument is used as if it were a function, the associated actual argument must be an intrinsic function or an external function. A dummy argument that becomes associated with an intrinsic function never has any automatic typing property, even if the dummy argument name is the same as the intrinsic function name. Therefore, the type of the dummy argument must agree with the type of the result of all specific actual arguments that become associated with the dummy argument. If a dummy argument name is used as if it were an external function and that name also appears as an intrinsic function name, the intrinsic function is not available for referencing within the subprogram.

A dummy argument that is used as a procedure name in a function reference and is associated with an intrinsic function must have arguments that agree in number and type with those specified for the intrinsic function.

If a dummy argument appears in a type statement and an EXTERNAL statement, the actual argument must be the name of a function.

If the dummy argument is referenced as a subroutine, the actual argument must be the name of a subroutine and must not appear in a type statement or be referenced as a function.

Restrictions on the association of entities - If a subprogram reference causes a dummy argument in the referenced subprogram to become associated with another dummy argument in the referenced subprogram, neither dummy argument may become defined during execution of that subprogram. For example, if a subroutine is headed by

```
SUBROUTINE XYZ (A,B)
```

and is referenced with

```
CALL XYZ (C,C)
```

then the dummy arguments A and B each become associated with the same actual argument C and, therefore, with each other. This rule prohibits both A and B from becoming defined during this execution of subroutine XYZ or by any procedures referenced by XYZ.

If a subprogram reference causes a dummy argument to become associated with an entity in a common block in the referenced subprogram, neither the dummy argument nor the entity in the common block may become defined within the subprogram. For example, if a subroutine containing statements

```
SUBROUTINE XYZ (A)
```

```
COMMON C
```

is referenced by a program unit that contains the statements

```
COMMON B
```

```
CALL XYZ (B)
```

the dummy argument A becomes associated with the actual argument B. B and C are associated in a common block. The rule states that neither A nor C may become defined during the execution of subroutine XYZ or by any procedures it references.

Common blocks

A common block provides a means of communication between external procedures or between a main program and an external procedure. The variables and arrays in a common block may be defined and referenced in all subprograms that contain a declaration of that common block. Because association is by storage rather than by name, the names of the variables and arrays may be different in the different subprograms. A reference to a datum in a common block is proper if the datum is defined and of the same type as the type of the name used to reference the datum. However, an integer variable that has been assigned an executable statement label must not be referenced in any program unit other than the one in which it was assigned.

The only difference in data type permitted between that defined and that referenced is that either part of a complex datum may be referenced as a real datum.

In a subprogram that has declared a named or blank common block, the entities in the block remain defined after the execution of a RETURN or END statement.

Common blocks also may be used to reduce the total number of storage units required for an executable program by causing two or more subprograms to share some of the same storage units. This sharing of storage is permitted if the rules for defining and referencing data are not violated.

3.6.12 UTILITY PROCEDURES

The ANSI FORTRAN Standard does not provide for utility procedures.

The CRAY-1 FORTRAN programmer can reference a number of predefined functions, subroutines, and other procedures as described in Appendix C. These *utility procedures* extend program control capabilities in the areas of:

- CRAY-1 Operating System (COS) features,
- Input/output operations, and
- Boolean, masking, shifting, and related operations.

3.6.13 BLOCK DATA SUBPROGRAMS

Block data subprograms are used to provide initial values for variables and array elements in named common blocks.

A block data subprogram is a program unit that has a BLOCK DATA statement as its first statement. A block data subprogram is non-executable.

BLOCK DATA statement form

The form of a BLOCK DATA statement is

BLOCK DATA [*sub*]

where *sub* is the symbolic name of the block data subprogram in which the BLOCK DATA statement appears.

The optional name *sub* is a global name and must not be the same as the name of an external procedure, main program, or other block data subprogram in the same executable program. The name *sub* must not be the same as any local name in the subprogram.

The ANSI FORTRAN Standard does not provide for block data subprogram names or main program names.

Block data subprogram restrictions

The BLOCK DATA statement must appear only as the first statement of a block data subprogram. The only other statements that may appear in a block data subprogram are IMPLICIT, PARAMETER, DIMENSION, COMMON, EQUIVALENCE, DATA, type, and END statements.

If a named common block has an entity initialized, all entities having storage units of the common block storage sequence must be specified even if they are not all initialized. More than one named common block may have entities initialized in a single block data subprogram. Entities not in a named common block must neither be initialized nor appear in a DIMENSION, EQUIVALENCE, or type statement in a block data subprogram.

A given named common block may be specified in more than one block data subprogram in an executable program.

There may be any number of unnamed block data subprograms in an executable program. Any number of differently named block data subprograms may be specified in an executable program.

An END statement must be contained in the last line used in specifying a block data subprogram.

Example:

```
BLOCK DATA BD1
:
COMMON/NAME1/TABLEA, TABLEB, TEST1, TEST2
DIMENSION TABLEA(10,10), TABLEB(6,2,2)
DATA TABLEA/100*123./, TABLEB/12*0., 12*1./
DATA TEST1/72.35E-20/, TEST2/'EXAMPLES'/
:
END
```

3.7 ASSOCIATION AND DEFINITION

3.7.1 STORAGE AND ASSOCIATION

Storage sequences are used to describe association among variables, array elements, common blocks, and arguments.

Storage sequences

A *storage sequence* is a sequence of storage units. Each array and common block has a storage sequence. The *size of a storage sequence* is the number of storage units it contains. A *storage unit* contains one variable or array element of type integer, real, or logical.

A double-precision or complex variable or array element has a storage sequence of two storage units. In a double-precision storage sequence, the most and least significant parts of a datum are contained in the first and second storage units, respectively. In a complex storage sequence, the real and the imaginary parts of a datum are contained in the first and second storage units, respectively.

A storage unit in the CRAY-1 computer corresponds to one 64-bit memory location; a storage sequence to a contiguously addressed set of memory locations.

The ANSI FORTRAN Standard does not relate storage units and memory locations.

Association of storage sequences

Two storage sequences are *associated* if they share at least one storage unit.

Association of entities

Two entities are associated if their storage sequences are associated. *Totally associated* entities share the same storage sequence. *Partially associated* entities share part but not all of a storage sequence.

The definition status and value of an entity affects the definition status and value of any associated entity or entities. An EQUIVALENCE statement, a COMMON statement, or argument association in a procedure reference may cause the association of storage sequences.

An EQUIVALENCE statement causes association of entities within a program unit unless one of the entities is also in a common block.

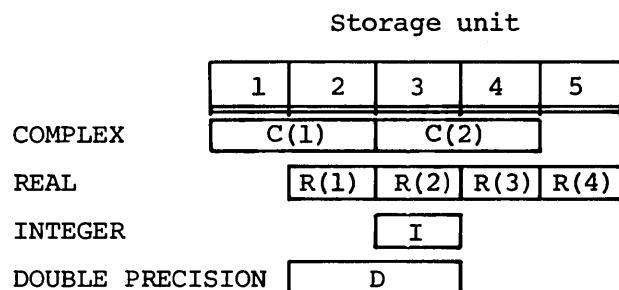
Arguments and COMMON statements cause entities in two or more program units to become associated.

Partial association may exist only between a double-precision or complex entity and a second entity of type integer, real, logical, double precision, or complex. Partial association may occur only through the use of COMMON or EQUIVALENCE statements. Partial association must not occur through argument association.

In the example

```
INTEGER I  
REAL R(4)  
COMPLEX C(2)  
DOUBLE PRECISION D  
EQUIVALENCE (C(2), R(2), I), (R,D)
```

the third storage unit of C, the second storage unit of R and the storage unit of I are specified as the same. The storage sequences may be illustrated as:



R(2) and I are totally associated. The following are partially associated: R(1) and C(1), R(2) and C(2), R(3) and C(2), I and C(2), R(1), and D, R(2) and D, I and D, C(1) and D, and C(2) and D. Note that although C(1) and C(2) are each associated with D, C(1) and C(2) are not associated with each other.

3.7.2 DEFINING ENTITIES

Variables and array elements become defined as follows:

- Execution of an arithmetic or logical assignment statement cause the entity that precedes the equals to become defined.
- As execution of an input statement proceeds, each entity that is assigned a value of its corresponding type from the input medium is defined at the time of such assignment.
- Execution of a DO statement causes the DO-variable to become defined.

- Beginning of execution of actions specified by an implied-DO list in an input/output statement causes the implied-DO-variable to become defined.
- A DATA statement causes entities to become initially defined at the beginning of execution of an executable program.
- Execution of an ASSIGN statement causes the variable in the statement to become defined with a statement label value.
- When an entity of a given type becomes defined, all totally associated entities of the same type become defined.
- A reference to a subprogram causes a dummy argument to become defined if the corresponding actual argument is defined.
- When a complex entity becomes defined, all partially associated real entities become defined.
- When both parts of a complex entity become defined as a result of partially associated real or complex entities becoming defined, the complex entity becomes defined.

3.7.3 UNDEFINING ENTITIES

Variables and array elements become undefined as follows:

- All entities are undefined at the beginning of execution of an executable program except those entities initially defined by DATA statements.
- When an entity of a given type becomes defined, all totally associated entities of different type become undefined.
- Execution of an ASSIGN statement causes the variable in the statement to become undefined as an integer. Entities of type integer that are associated with the variable are also undefined as integers.
- When an entity becomes defined, all partially associated entities become undefined. However, when an entity of type real is partially associated with an entity of type complex, the complex entity does not become undefined when the real entity becomes defined and the real entity does not become undefined when the complex entity becomes defined. When an entity of type complex is partially associated with another entity of type complex, definition of one entity does not cause the other to become undefined.
- When an error condition or an end-of-file condition occurs during execution of an input statement, all items in the input list of the statement become undefined.
- When an entity becomes undefined as a result of conditions described in the two preceding items, all totally and partially associated entities become undefined.

3.8 SCOPE AND CLASSES OF SYMBOLIC NAMES

A symbolic name consists of from one to eight alphanumeric characters, the first of which must be a letter. Some sequences of characters, such as format edit descriptors and keywords that uniquely identify certain statements (GO TO, READ, FORMAT, etc.) are not symbolic names nor do they form the first characters of symbolic names in such occurrences.

The ANSI FORTRAN Standard provides for symbolic names of up to six alphanumeric characters.

3.8.1 SCOPE OF SYMBOLIC NAMES

The scope of a symbolic name is an executable program, a program unit, or a statement function statement.

The name of the main program and the names of block data subprograms, external functions, subroutines, and common blocks have a scope of an executable program and are said to be global to that program.

The names of variables, arrays, constants, statement functions and intrinsic functions have a scope of a program unit.

The names of variables that appear as dummy arguments in a statement function statement have a scope of that statement.

Global entities

The main program, common blocks, subprograms, and external procedures are global entities of an executable program. A symbolic name that identifies a global entity must not be used to identify any other global entity in the same executable program.

A symbolic name in one of the following classes identifies a global entity in an executable program:

- Common block.
- External function.
- Subroutine.
- Main program.
- Block data subprogram.

The ANSI FORTRAN Standard does not provide for main program or block data subprogram names.

Local entities

A symbolic name of a local entity identifies that entity in a single program unit. A symbolic name that identifies a member in one class of entities local to a program unit must not also identify a member in another class of entities local to that same program unit. However, a symbolic name that identifies a local entity may, in a different program unit, identify an entity of a different class that is either local to that program unit or is global to the executable program. A symbolic name that identifies a global entity in a program unit must not be used to identify a local entity in that program unit except as noted in 3.8.2 for common block and external function names.

The symbolic name of an entity in one of the following classes identifies it as a local entity in a program unit:

- Array
- Variable
- Constant
- Statement function
- Intrinsic function

The ANSI FORTRAN Standard does not provide for the symbolic names of constants.

A symbolic name used as a dummy argument in a procedure is classified as identifying a variable, an array, or another procedure. This specification and usage must not violate the respective class rules.

3.8.2 CLASSES OF SYMBOLIC NAMES

In a program unit, a symbolic name must not correspond to more than one class except as noted in the following paragraphs. All restrictions on the appearances of the same symbolic name in different program units of an executable program are also noted here.

Common blocks

A symbolic name is the name of a common block if and only if it appears as a block name in a COMMON statement. A common block name is global to the executable program.

A common block name in a program unit may also be the name of any local entity. If a name is used for both a common block and a local entity, the appearance of that name in any context other than as a common block name in a COMMON statement identifies only the local entity.

External functions

A symbolic name is the name of an external function if it:

- Appears immediately following the keyword FUNCTION or ENTRY in a FUNCTION or ENTRY statement, or
- Is not an array name, statement function name, intrinsic function name, subroutine name, or dummy argument and every appearance is immediately followed by a left parenthesis except in a type statement, in an EXTERNAL statement, or as an actual argument.

The name of a function subprogram that appears immediately after the keyword FUNCTION or ENTRY in a FUNCTION or ENTRY statement must be the name of a variable in that subprogram. An external function name is global to the executable program.

Subroutines

A symbolic name is the name of a subroutine:

- If the name appears immediately following the keyword SUBROUTINE or ENTRY in a SUBROUTINE or ENTRY statement, or
- If the name appears immediately following the keyword CALL in a CALL statement and is not a dummy argument.

A subroutine name is global to the executable program.

The main program

A symbolic name is the name of a main program if and only if it appears in a PROGRAM statement in the main program. A main program name is global to the executable program.

The ANSI FORTRAN Standard does not provide for the PROGRAM statement or a main program name.

Block data subprograms

A symbolic name is the name of a block data subprogram if and only if it appears in a BLOCK DATA statement. A block data subprogram name is global to the executable program.

The ANSI FORTRAN Standard does not provide for a block data subprogram name.

Arrays

A symbolic name is the name of an array if it appears as the array name in an array declarator in a DIMENSION, COMMON, or type statement. An array name is local to a program unit and may be the same as a common block name.

Variables

A symbolic name is the name of a variable if it meets all of the following conditions:

- It does not appear in a PARAMETER or EXTERNAL statement,
- It is not the name of an array, subroutine, main program, or block data subprogram,
- It is not the name of an array, subroutine, main program, or subprogram,
- It appears other than as the name of a common block, the name of an external function in a FUNCTION statement, and
- It can be immediately followed by a left parenthesis (only when immediately preceded by the word FUNCTION in a FUNCTION statement).

The ANSI FORTRAN Standard does not provide for the PARAMETER statement, nor for a main program name.

A variable name is local to a program unit. A variable name in the dummy argument list of a statement function statement is local to the statement function statement in which it occurs. Note that the use of an intrinsic function name as a dummy argument of a statement function removes it from the class of intrinsic functions within the program unit containing that statement function.

A statement function dummy argument name may also be the name of a variable or common block in the same program unit. The appearance of the name in any context other than as a dummy argument of the statement function identifies a local variable or common block. The statement function dummy argument name and local variable name have the same type.

Constants

A symbolic name is the name of a constant if it appears as a symbolic name in a PARAMETER statement. A constant name is local to a program unit.

Statement functions

A symbolic name is the name of a statement function if it is not an array name and a statement function statement exists that specifies that symbolic name. A statement function name is local to a program unit. A statement function name may be the same as a common block name.

Intrinsic functions

A symbolic name is the name of an intrinsic function if it appears in Appendix B and:

- Does not appear in a conflicting type statement,
- Is not an array name, statement function name, subroutine name, or dummy argument name, and
- Every appearance of the symbolic name, except in a type statement or as an actual argument, is immediately followed by an actual argument list enclosed in parentheses.

An intrinsic function name is local to a program unit.

4 CRAY-1 FORTRAN PROGRAMMING

4.1 GENERAL

The CRAY-1 FORTRAN Compiler (CFT), in producing CRAY-1 machine language instructions from FORTRAN language statements, does so with run-time efficiency as a prime objective. Its operations include:

- Providing the most effective instruction sequence for each FORTRAN statement compiled,
- Making full use of all CRAY-1 capabilities and techniques enhancing the speed with which these sequences execute, and
- Analyzing FORTRAN statements to accomplish the foregoing.

Of particular significance is the compilation of statements describing iterative operations amenable to vector processing. When properly applied, vector processing affords dramatic decreases in computation time over equivalent scalar processing methods. Also significant is the structuring of data in memory to take advantage of its potential for 80-million words-per-second transfers during computation. The CRAY-1 FORTRAN programming techniques described in this section address both considerations.

4.2 FORTRAN PROGRAMMING FOR CRAY-1 VECTOR OPERATIONS

CFT analyzes the innermost loops of the FORTRAN programs it compiles to determine whether vector processing methods can be applied to improve overall program efficiency. If so, the execution sequence of the code it produces contains vector instructions to drive the high-speed vector and floating-point functional units and the eight vector registers in their integrated processing of the specified operation. This feature of CFT is automatically activated through compiler analysis of statements contained in certain DO-loops in the program without special notation on the part of the programmer. Thus, no special provisions are required that would encumber the programmer or affect the transportability of the programs produced or used.

Not all DO-loops are vectorizable. In determining the qualifications of a loop for vectorization, CFT examines each statement and its relationship to others in that DO-loop range. The CRAY-1 FORTRAN programmer can enhance the performance of his programs by avoiding certain constructs known to inhibit DO-loop vectorization.

To be vectorizable, a DO-loop must manipulate or perform calculations on the contents of one or more arrays. Loops containing a GO TO, IF, or CALL statement are not vectorizable. A vectorizable DO-loop may contain references to any of the CRAY-1 FORTRAN intrinsic functions and may contain expressions having arithmetic, relational and logical operators. Any procedures named in an EXTERNAL statement cause the inhibiting of vectorization of an inner DO-loop referencing it.

Further loop analysis is performed to determine that all variables defined or referenced in the DO-loop range are in one of three categories:

- Invariants,
- Constant increment integers (CII's), or
- Vector array references.

A loop that contains one or more variables outside these categories, cannot be vectorized.

An invariant is a variable that is referenced but not defined in the course of DO-loop execution.

A constant increment integer (CII) is an integer variable that is incremented by a constant value on each pass through the loop and at only one point in the DO-loop range. At this point, the expression used to define the CII may, itself, reference a single CII variable. The expression must not use operators other than plus and minus.

A vector array reference is an array element name in which one subscript expression contains a CII and all others, if any, are invariants. A vector array reference subscript expression containing a CII must be of the form:

[\pm invariant *] CII [\pm invariant expression]

In the following example, I, J, and K are CII's; A, B, and C are vector array references, and KDELTA, X, and D are invariants. The DO-loop is vectorizable.

```
DO 10 I = 3,101,2
K = K - KDELTA
J = 107 - I
A(3,I-2) = COS(B(J)) **C(M-2*K+L*M/7,L,M/L)*X*D(L,M)
10    CONTINUE
```

One common FORTRAN practice, that of using temporary variables to contain repeated subexpressions, clearly inhibits inner DO-loop vectorization. For example, the appearance of the statements

```
TEMP = A(I) + B(I) * C(I)
D(I) = TEMP + 1. / TEMP
```

in a DO-loop will disqualify it for vectorization since TEMP is not an invariant, a CII, or a vector array reference. The additional memory references required in substituting TEMP(I) for TEMP to eliminate this problem offset the advantage gained. CFT examines DO-loops for repeated subexpressions. Those encountered are evaluated only once. Consequently, their repetition does not cause an increase in execution time. Another alternative is to reference statement functions which are analyzed for possible vectorization.

Hence the statements

```
ASF(X,Y,Z) = X + Y * Z  
:  
D(I) = ASF(A(I),B(I),C(I)) + 1. / ASF(A(I),B(I),C(I))
```

or the statement

```
D(I) = C(I) * B(I) + A(I) + 1. / (A(I) + B(I) * C(I))
```

are vectorizable, involve only one evaluation of $A(I) + B(I) * C(I)$, and do not involve any unnecessary memory references.

Attention to the structure of statements within nested DO-loops often improves execution time by permitting vectorization. For example, the common matrix multiplication method specified in the statements

```
DO 10 I = 1,N  
DO 10 J = 1,M  
A(I,J) = 0.  
DO 10 L = 1,K  
10 A(I,J) = A(I,J) + B(I,L) * C(L,J)
```

does not vectorize because $A(I,J)$ is independent of the inner DO-loop index. By rewriting this execution sequence as

```
DO 10 I = 1,N  
DO 5 J = 1,M  
5 A(I,J) = 0.  
DO 10 L = 1,K  
DO 10 J = 1,M  
10 A(I,J) = A(I,J) + B(I,L) * C(L,J)
```

vectorization occurs and the loop, in actual tests, has proven to require only 15% of the original loop execution time.

Interdependencies among the elements of a vector can be expressed in FORTRAN statements, but cause CFT to inhibit the vectorization of any DO-loop in which they are encountered. A DO-loop such as

```
DO 10 I = 2,100  
10 T(I,J) = (T(I-1,J) + T(I+1,J) + T(I,J+1) + T(I,J-1)) / 4.
```

does not vectorize since the vector operations of the CRAY-1 CPU cause $T(I-1,J)$ to be calculated at the same time it is required for the calculation of $T(I,J)$.

The same dependency problem can occur among separate statements in a DO-loop. Multiple-statement dependencies pose far more complex problems for CFT analysis since the form and value range of subscripts, the order of references, and the sign of the control variable must be considered in various combination to determine whether vectorization is permissible. In the following sets of statements, positive incrementation of the control variable, I, inhibits vectorization:

A(I)=A(I-1)	B(I)=A(I-1)	A(I)=...
	A(I)=...	B(I)=A(I+1)

In the last case, the simultaneous calculation of A(I) and A(I+1) would cause B(I) to be defined with the just-calculated value of A(I+1) if vectorization were permitted.

Under the same circumstances, the following sets of statements are vectorizable:

A(I)=A(I+1)	B(I)=A(I+1)	A(I)=...
	A(I)=...	B(I)=A(I-1)

Note that in both sets of examples, applying a negatively incremented control variable reverses the dependency situation. If, for example, the DO-loop were initiated with a

```
DO 10 I = 100,1,-1
```

then the first set of examples above would be vectorizable while the second set would not.

The inhibiting of vectorization due to such dependencies can be relaxed in the case of multiply-dimensioned array processing. CFT must be able to determine that the specified array elements are in different vectors (i.e., rows, columns, planes, etc.) of the array. For example, the loop

```
DO 10 I = 2,100
    10 A(I,J) = A(I-1,J-1)
```

is vectorizable, while the similar loop

```
DO 20 I = 2,100
    20 A(I,J) = A(I-1,JMINUS1)
```

cannot be vectorized since CFT cannot determine whether J and JMINUS1 are equal.

The following summation loop cannot be vectorized:

```
SUM = 0  
DO 10 I = N,M  
10 SUM = SUM + A(I)
```

(where A is an array or an expression involving arrays).

Rewritten in the following form, the summation loop becomes highly vectorizable:

```
DO 10 I = 1,MIN0(64,M-N+1),1  
10  HELP(I) = A(N+I-1)  
JS = I+N-2  
KOUNT = (M-JS)/64  
LAST = (M-JS)-64*KOUNT  
DO 20 J = 1,KOUNT  
DO 15 I = 1,64  
15  HELP(I) = HELP(I)+ A(I+JS)  
20  JS = JS + 64  
DO 30 I = 1,LAST  
30  HELP(I) = HELP(I) + A(M+1-I)  
SUM = HELP(1)  
DO 40 I = 2,MIN0(64,M-N+1),1  
40  SUM = SUM + HELP(I)
```

\

The first loop does a vector transfer of the first elements of array A to array HELP. (Note the use of the intrinsic function MIN0.) The value of KOUNT is the number of 64 word blocks remaining and LAST is the number of words in the last partial block. Note that if M-N+1 is less than 129, KOUNT will be zero (or negative) and the loop initiated by the DO 20 ... statement will not execute. Also note that if M-N+1 is less than 64 or is a multiple of 64, LAST will be less than or equal to zero and the loop initiated by the DO 30 ... statement will not execute.

The following are comparative timings in microseconds for the simple and rewritten summation loops:

<u>M-N+1</u>	<u>Simple summation</u>	<u>Rewritten summation</u>	<u>Ratio</u>
1	3.2	6.4	2.0
3	4.2	7.2	1.7
10	7.6	10.8	1.4
30	17.3	21.1	1.2
100	51.5	40.1	.78
300	149.0	53.1	.36
1000	490.2	97.9	.20
3000	1465.2	225.2	.15
10000	4877.7	673.3	.14
30000	14627.7	1949.8	.13
100000	48752.7	6421.0	.13

The compiler directive IVDEP can be specified in advance of an inner DO-loop DO statement to cause vector dependencies to be ignored in determining whether or not to vectorize that loop. (See Section 5 for a description of IVDEP and other compiler directives.)

DO loops containing IF statements will not vectorize. The utility procedures CVMGP, CVMGM, CVMGZ, and CVMGN can often be used in lieu of IF statements to produce vectorizable loops.

The simple case

```
DO 10 I = N,M  
X(I) = A(I)  
10   IF(B(I).GT.C(I))X(I) = D(I)
```

could be rewritten as

```
DO 10 I = N,M  
10   X(I) = CVMGM(D(I),A(I),C(I)-B(I))
```

or as

```
DO 10 I = N,M  
10   X(I) = CVMGP(A(I),D(I),C(I)-B(I))
```

to produce vectorizable loops.

Similarly,

```
DO 10 I = N,M  
10   IF(X(I).GE.10.)X(I) = X(I) + 1.0
```

could be rewritten as

```
DO 10 I = N,M  
10   X(I) = CVMGP(X(I)+1.0,X(I),X(I)-10.)
```

to do the same.

The two possible drawbacks to vectorizing IF statements in this way are extra work and illegal operations.

Extra work occurs when the computer evaluates both true and false expressions before deciding which value to use. The loop

```
DO 10 I = 1,1000
10   IF(X(I) .GE..9999)X(I) = ASIN(X(I))
```

might never invoke the ASIN routine if X contains random numbers in the range from 0 to 1. However, the equivalent vector loop

```
DO 10 I = 1,1000
10   X(I) = CVMGP(ASIN(X(I)),X(I),X(I)-.9999)
```

requires 1000 evaluations of ASIN. Whether or not there is a speed advantage to vectorizing IF statements in this manner depends in a complicated way on the complexity of the expressions and on the probability of their evaluation being required.

The second problem is possible illegal operations. A loop such as

```
DO 10 I = N,M
      IF(X(I) .NE.0)X(I) = 1./X(I)
10   IF (Y(I) .GE.0.)Y(I) = SQRT(Y(I))
```

should not be rewritten using the preceding methods. In the rewritten loop

```
DO 10 I = N,M
      X(I) = CVMGN(1./X(I),X(I),X(I))
10   Y(I) = CVMGP(SQRT(Y(I)),Y(I),Y(I))
```

both division and use of the intrinsic function SQRT would produce errors when processing unneeded cases.

The general guidelines below can be followed to promote vectorization of DO-loop operations:

- Keep subscripts simple and explicit.
- Do not use temporary variables to contain repeated subexpressions.
- Do not use IF, GO TO, or CALL statements.
- Use the CRAY-1 FORTRAN intrinsic functions where appropriate.
- Make judicious use of the CRAY-1 FORTRAN utility procedures CVMGP, CVMGM, CVMGZ, and CVMGN in lieu of IF statements.
- Rewrite large loops that contain a few unvectorizable statements as two or more loops, one or more of which will vectorize.

4.3 FORTRAN PROGRAMMING FOR CRAY-1 MEMORY OPERATIONS

The memory section of the CRAY-1 CPU can operate at a maximum transfer rate of 80-million words per second. This high speed owes in large measure to the location of consecutively addressed memory locations in separate memory banks. Since each bank operates independently, the 50 nanosecond period required to access a single memory location can overlap the 50 nanosecond period required to access the next-higher-addressed memory location, and so on. This method of accessing memory for a set of consecutively addressed memory locations is such that only the first word accessed requires 50 nanoseconds to become available with remaining words available at each 12.5 nanosecond interval thereafter.

Addresses 0 through 15 refer to memory locations in banks 0 through 15, respectively; addresses 16 through 31 to memory locations in banks 0 through 15; etc. Thus, every sixteenth address references a memory location in the same memory bank.

Data can be organized without attention to this characteristic of the CRAY-1 memory section and be fully compatible with FORTRAN processing methods and with the storage sequences established by CRAY-1 FORTRAN. Avoiding certain storage sequence patterns will, however, ensure the best possible timing of transfer rates to and from memory.

A general rule would be to avoid specifying storage sequences which, in the course of common transfer operations, would cause a given memory bank to be accessed more often than once every four clock periods.

5 THE CRAY-1 FORTRAN COMPILER

THE CRAY-1 FORTRAN COMPILER

5

5.1 GENERAL

The CRAY-1 FORTRAN Compiler (CFT) transforms a CRAY-1 FORTRAN language program into a executable program in relocatable binary that can be loaded and executed on the CRAY-1 Computer System.

The CRAY-1 Operating System (COS) calls upon the system loader (LDR) to load CFT from the Mass Storage Subsystem. Compiler operation responds to information in a COS job deck to locate and compile the FORTRAN program. Both binary and symbolic information are output from its operation.

This section describes CFT in terms of the input and output characteristics that the CRAY-1 FORTRAN programmer must understand to produce correct and effective programs.

5.2 INPUT TO CFT

CFT, when initiated, seeks two types of information; the program to be compiled and instructions on controlling the compilation.

A FORTRAN program to be compiled by CFT must be specified in punched-card or other form using the ASCII character codes in Appendix A and the format specified in Section 3. The result is a *source deck*. This source deck is not directly usable by CFT and must be converted to a CRAY-1 dataset and stored on disk before CFT can process its content. The means and manner of this conversion and storage occurs external to CFT.

Other information required by CFT for its operations is provided through COS and from compiler directives specified in (but not a part of) the program being compiled. This includes identification of the input dataset containing the source deck, identification of datasets to receive the binary and listable output from CFT during compilation, and specification of which CFT options are to be exercised. Much of this information is contained in the CFT job statement card and other cards in the user's job deck (see Section 6). Compiler directives are described in 5.4.

5.3 OUTPUT FROM CFT

The two forms of compiler output are:

- The compiled FORTRAN program in relocatable binary form, and
- A printable record of the compilation including each FORTRAN statement compiled, the assembly-language (CAL) equivalent of the machine language produced, and a cross-reference of all symbolic names and statement labels within the program being compiled.

5.3.1 BINARY OUTPUT

The relocatable binary program created by CFT compilation provides for the complete executable program including the reservation and possible initialization (via DATA statements) of all data storage sequences. Output is onto a pre-identified dataset in a format required by the system loader routine, LDR. Addresses in this output are relocatable. Additionally, LDR loads and links routines required from the system library (\$FTLIB).

5.3.2 LISTABLE OUTPUT

CFT optionally produces a dataset containing a source statement listing, a symbol table, and a listing of generated code for each program compiled. Additionally, diagnostic information may be interspersed.

Listable output is written on a dataset. The CFT control statement and compiler directives allow the user to control this output and the dataset to receive it.

Source statement listings

The source statement listing records all FORTRAN statements comprising the program as they are sequentially read and interpreted from the source input dataset. A sequence number is listed for each statement to identify its position in the program. Errors encountered while compiling a statement are flagged by lines subsequent to that statement or are recorded at the end of the source statement listing.

Symbol table and common block name list

A symbol table may follow each CFT-compiled program unit. After this table is a list of names and lengths (in octal) of all common blocks used in the subroutine.

The printing of a symbol table and/or a common block name list can be suppressed through an option on the CFT job control statement card. CFT provides for an optional cross-reference listing of statement number usage.

Information in the symbol table is presented in seven fields:

- Warning flags
- Address
- Name
- Block
- Type
- Use
- Dummy argument/equivalence flag

Fields other than the warning flags field are identified by titles.

Warning flags - Warning flags may precede the address field of variables and arrays.

<u>Flag</u>	<u>Significance</u>
N	NULL. The variable or array is neither assigned a value (defined) nor referenced in the program unit.
U	UNDEFINED. The variable or array is referenced but is not defined. Its content is unpredictable.
C	COMMON UNDEFINED. The variabe or array is in common and is referenced but not defined. (Since it is in a common block, it may be assigned a value in a different program unit.)

Address field - Addresses in this field are relative to the beginning of the program or to a named common block.

Name field - The name field contains an alphabetized list of all symbolic names specified in the program unit. If there is no PROGRAM statement in the executable program, CFT identifies the main program with the name \$MAIN.

Common block field - The block field identifies the common block containing a variable or array. If no common block name appears, the variable or array is local to the program unit.

Type field - The type field gives the type of array, variable, or program unit and can contain the following:

<u>Type</u>	<u>Significance</u>
LOG	Logical
INT	Integer
REAL	Real
CPLX	Complex
DOUB	Double precision
(BLANK)	Typeless function or subroutine

Use field - An entry in the use field describes the use of the corresponding symbolic name and may contain the following:

<u>Use</u>	<u>Significance</u>
AR n	Array with n dimensions ($1 \leq n \leq 7$)
ASF	Arithmetic statement function
ENT	Entry
EXT	External function or subroutine
INT	Intrinsic function
VAR	Simple variable

Dummy argument and equivalence flags - The characters DA appear in the field following the use field if the symbol is a dummy argument to the subroutine or function. The characters EQ appear if the symbol is an equivalenced variable or array.

Statement label list

Statement labels are listed in a separate table. A statement label may be identified as FORMAT, ***UNDEF, INACTIVE, or DO TERM. DO TERM is used if the statement is only referenced as the termination statement of a DO-loop. If a statement label is active and defined, the parcel address at which it is defined is given. Parcel addresses of instructions that refer to the statement label are optionally listed.

Internally generated statement labels are used for logical IF statements, for implied DO statements, or to mark the start and end of a DO-loop. In the first two cases, a 6-digit number in sequence from 000001 is generated with leading zeros present and significant. The beginnings and ends of DO-loops are labelled with pairs of compiler-generated statement labels. These are constructed by suffixing a DO terminator statement label with a letter. The first lettered label of a pair marks the beginning of the DO-loop; the second marks the end.

Warning messages

Warning messages are produced by CFT for statements or statement sequences that might cause error conditions during program execution. Their appearance does not affect compilation. Warning messages are listed in Appendix D. The following general form is used (seq. no. represents the number of the statement at which the problem was detected):

seq. no. WARNING message

Warning messages can be suppressed through parameters on the CFT job statement card.

Fatal error messages

When CFT determines the presence of one or more fatal errors, it continues to check subsequent statements for syntax errors but does not generate valid object code. CFT is a two-pass compiler. Errors detected during pass one are flagged immediately after the line containing the error. Errors detected during pass two are flagged at the end of each program unit. A fatal error message is listed in the following general form where *seq. no.* represents the CFT-assigned sequence number of the statement at which the error was detected and ERROR NO. *n* provides a sequential count of errors as encountered:

seq. no. ERROR NO. n message

Fatal error messages are listed in Appendix D.

5.4 COMPILER DIRECTIVES

The CRAY-1 FORTRAN Compiler (CFT) may be directed in certain of its actions by information specified in:

- The CFT control statement that appears in the job control file for the job being processed, and
- Compiler directives expressed in compiler directive lines that may be interspersed among the lines of statements and comments comprising the FORTRAN program units being compiled.

Compiler options expressed by the user in the CFT control statement (see Section 6) establish particular methods for application throughout the compilation of all related FORTRAN program units. *Compiler directives* encountered in the program units being compiled may change or reinstate this set of methods at predetermined points in the compilation. Certain other compiler actions are enabled and disabled only by compiler directives.

The CFT control statement E (enable compiler directives) option must be specified ON for compiler options to be recognized by CFT. Else, the lines containing compiler options are treated as comment lines.

The following paragraphs describe:

- Compiler directive lines,
- Listable output control directives,
- Vectorization control directives, and
- Integer control directives.

5.4.1 COMPILER DIRECTIVE LINES

A *compiler directive line* (or card) contains the characters CDIR\$ in columns 1 through 5. Generally, column 6 must be blank or contain the character zero, and columns 7 through 72 contain one or more compiler directives separated by commas. Spaces may precede, follow, or be embedded within a compiler directive. Columns 73 through 80 may be used for any purpose. Continuation of compiler directive information beyond a single line may be accomplished by:

- Entering a blank or zero character in column 6 of the initial line,
- Entering any other character in column 6 of up to 19 subsequent lines,
- Entering the characters CDIR\$ in columns 1 through 5 of all lines in the sequence, and
- Entering compiler directives in columns 7 through 72 of each line in the sequence as though a single field of up to 1,320 columns were provided.

Note that the character C in column 1 identifies these lines as comment lines to all but the CRAY-1 FORTRAN Compiler; a feature maintaining the transportability of programs using compiler directives.

Compiler directive lines are listed in the source statement listing.

5.4.2 LISTABLE OUTPUT CONTROL DIRECTIVES

The five listable output control directives (and their forms) are:

- EJECT
- LIST
- NOLIST
- CODE
- NOCODE

The CFT control statement L (listable output control directives) option must be specified ON to cause this set of compiler directives to be recognized.

EJECT directives

A compiler directive line containing an EJECT directive will be printed as the last line of the current page of source statement listing being produced. If the EJECT directive is contained in a continuation set of compiler directive lines, the last of these becomes the last line of the page. In either case, a new page is begun having as its first line the line following the last one printed on the preceding page. The EJECT directive has no effect if production of the source statement listing has been suppressed.

LIST directives

The LIST directive causes resumption of listable output production as was suppressed by a NOLIST directive. The LIST directive either results in the production of a source statement listing or is ignored if one is already being produced.

NOLIST directives

The NOLIST directive suppresses the production of all listable output. If no listable output is being produced, the NOLIST directive is ignored.

CODE directives

The CODE directive causes the production of CFT-generated code listings if suppressed by a NOCODE directive or by CFT control statement action. A complete CFT-generated code listing is produced for a program unit containing a CODE directive, and for all subsequent program units to, but not including, that containing the next NOCODE directive.

NOCODE directives

The NOCODE directive suppresses the production of CFT-generated code listings. The entire CFT-generated code listing of a program unit containing a NOCODE directive is suppressed, as are the listings for all subsequent program units to, but not including, that containing the next CODE directive. If no CFT-generated code listings are being produced, the NOCODE directive is ignored.

5.4.3 VECTORIZATION CONTROL DIRECTIVES

The three vectorization control directives (and their forms) are:

- VECTOR
- NOVECTOR
- IVDEP

VECTOR directives

The VECTOR directive causes the compiler to resume its attempts to vectorize inner DO-loops if such were suppressed by a NONVECTOR directive. Attempted vectorization is resumed for all inner DO-loops in that program unit containing the VECTOR directive. When attempted vectorization of inner DO-loops is not called for by CFT control statement option action, the VECTOR directive is ignored.

NOVECTOR directives

The NOVECTOR directive suppresses the compiler's attempts to vectorize all inner DO-loops beginning with those in the program unit containing the NOVECTOR directive. If inner DO-loop vectorization is not being attempted, the NOVECTOR directive is ignored.

IVDEP directives

The IVDEP directive is specified in advance of a DO statement to cause the compiler's attempts to vectorize the corresponding DO-loop to ignore any vector dependencies encountered. The IVDEP directive affects only the single DO-loop it precedes. Note that conditions other than vector dependencies may cause the inhibiting of vectorization whether or not an IVDEP directive is specified.

5.4.4 INTEGER CONTROL DIRECTIVE

The form of the single integer control directive, INT24, is:

INT24 v [$,v$] ...

where INT24 specifies a 24-bit integer data type and

v is the symbolic name of a variable or array.

The specification of INT24 in a program unit causes all variables and arrays named in its argument list to be identified as entities of type integer that provide 24-bit (instead of the usual 64-bit) values when referenced. The INT24 directive is not a CRAY-1 FORTRAN language statement. It must, however, be specified in a program unit according to the rules for specifying type statements. Unlike other compiler directives, INT24 must be the only compiler directive specified in the compiler directive line(s) containing it.

6 THE CRAY-1 OPERATING SYSTEM

THE CRAY-1 OPERATING SYSTEM

6

6.1 GENERAL

The CRAY-1 Operating System (COS) consists of programs that control the operation of a CRAY-1 Computer System. All jobs are processed under COS in a multiprogramming environment. In this environment, COS allocates system resources in a manner that attempts to optimize the use of these resources and that resolves conflicts among jobs requiring them.

The CRAY-1 FORTRAN programmer communicates the way in which his job is to be processed by preparing instructions to COS in the form of job control statements. These, along with the FORTRAN program and/or data to be processed, are presented to COS as a job. This section describes only those features and capabilities of COS essential to the creating and processing of FORTRAN jobs. A more detailed treatment of COS is beyond the scope of the manual and is the subject of Cray Research publication 2240011, "CRAY-OS Version 1.0 External Reference Specification".

6.2 JOB PREPARATION

In preparing his job for COS processing, the CRAY-1 FORTRAN programmer must specify the complete procedure COS is to follow; from the initial establishing of his work as a job through the disposition of all datasets supporting or resulting from its operation.

The first step in this task is typically the transcription of all program, data, and COS instruction information into punched-card or other suitable form. Instructions to COS are expressed as job control statements. Each statement is placed on a punched-card and the set of punched-cards comprise what is called the *COS job control deck*. The program is placed on punched-cards as described in Section 3 and is referred to as the *program or source deck*. Data to be referenced in the course of program execution, if any, is prepared in one or more *data decks* and in the format required by the input/output statements in the program.

Step two usually establishes these decks as files in a CRAY-1 dataset. Software at the host computer or the input/output station being used provides for creation of this dataset at the CRAY-1. The methods for accomplishing this vary according to host computer or station type and are described in other Cray Research publications. Once established in the CRAY-1, this dataset, called the *job dataset*, typically contains the COS job control deck as its first file, the program deck as its second file, and the one or more data decks, if any, as its third and following files. COS, which is in constant control of the CRAY-1, recognizes this dataset job to be processed. The job control statements comprising the first file are stored for job processing control; the remainder of the dataset (including data for the programs, if any) is placed in a dataset named \$IN.

At the conclusion of step two, COS is prepared to begin processing the job. Control statements in the job control deck govern the sequence of operations COS performs. They usually describe the following:

- Establishing datasets required for compiler input and output.
- Initiating and controlling CFT execution.
- Establishing any data decks (contained in \$IN) as datasets.
- Loading and initiating the operation of the compiled program.
- Disposing of datasets output from CFT and from the program.

When COS completes its monitoring of these operations, it terminates the job and makes available a logfile containing information on its actions. At this point, the disposition of datasets occurs, including the routing of all listable output to the host (front-end) processor or input/output station for printing.

The following paragraphs describe job control statements and resultant COS actions essential to these basic operations.

6.3 REQUIRED JOB CONTROL STATEMENTS

The first file of a job dataset contains job control statements that are sequentially read and processed.

Job control statements identify the job to the system, define operating characteristics for the job, manipulate datasets, call for the loading and execution of user programs, and call programs that perform a number of utility functions for the user. Only those job control statements essential to COS processing of FORTRAN jobs are discussed in this section.

Complete descriptions of each are provided in Cray Research publication 2240011, "CRAY-OS Version 1.0 External Reference Specification".

6.3.1. JOB IDENTIFICATION

The JOB control statement must be the first statement in a control statement file. It defines the name and characteristics of the job to the operating system.

6.3.2 ASSIGN DATASET CHARACTERISTICS

The ASSIGN control statement creates a new dataset or changes the characteristics of an existing dataset.

A FORTRAN program refers to datasets by using a unit number in its I/O statements. Allowable unit numbers range from 0 through 99, inclusive. The CRAY-1 Operating System recognizes the following three standard datasets and reserved unit numbers:

<u>Dataset name</u>	<u>Unit no.</u>	<u>Standard use</u>
\$IN	100	Job input
\$OUT	101	List output
\$PUN	102	Punch output

These three datasets are referred to implicitly by PRINT, PUNCH, and certain forms of READ statements appearing in FORTRAN programs.

A user associates a unit number with a dataset name by using the unit number prefixed by FT as either the dataset name or alias name in the ASSIGN statement. (The unit number with the FT prefix is sometimes referred to as the logical file name.)

Using the logical file name with the DN keyword (i.e., DN=FTnn) causes a user dataset named FTnn to be created. Using the logical file name with the A parameter (i.e., A=FTnn) causes FTnn to be an alias dataset name for the dataset named by the DN keyword.

If ASSIGN statements are used, they must be placed in the job control statement file prior to loading and execution of the object program; that is, they can be placed anywhere before the LDR statement.

Note that if an ASSIGN statement is not used, a dataset open for both input and output is created when the unit number is first referred to by the executing program.

The following ASSIGN statement designates the unit number 11 as an alias of the standard dataset \$OUT:

```
ASSIGN(DN=$OUT,A=FT11)
```

All references to unit number 11 in the FORTRAN program are translated to references to \$OUT, which is automatically sent to the station at job end.

6.3.3 CRAY-1 FORTRAN COMPILER EXECUTION

The CFT compiler is loaded and executed when a CFT control statement is encountered in the control statement file of the job deck on the \$IN dataset.

The format of the CFT control statement is:

CFT(I=idn,L=ldn,B=bdn,C=cdn,ON=string,OFF=string)

Options may be in any order. If a keyword and option are omitted from the statement, the compiler uses a default value. If all options are omitted, a period may be used in lieu of empty parentheses.

The compiler does not reposition datasets either before or after compilation.

I= *idn* Specifies dataset containing source input.

idn Name by which source is known at CRAY-1;
 default is \$IN.

L= *ldn* Specifies dataset to receive list output.

ldn Name by which list output is known; default is
 \$OUT. L=0 suppresses all list output except
 for fatal error messages which are written on
 \$OUT.

B= *bdn* Specifies dataset on which compiler writes binary load
 modules.

bdn Name by which binary load files are known to the
 CRAY-1; default is \$BLD. An end-of-file is not
 written.

C= *cdn* Specifies CAL dataset. This option provides for the
 generation of a text file that contains acceptable input
 to the CAL assembler, possibly with minor hand corrections.
 It is intended to be used for hand coding of inner loops
 for enhanced efficiency.

cdn Name of CAL dataset; default is no dataset.

ON=*string* Enables list or compile options; see table 6-1.

string List of characters representing options to be
 enabled.

OFF=*string* Disables list or compile options; see table 6-1.

string List of characters representing options to be
 disabled.

Table 6-1. ON and OFF options

ON and OFF options	Default with ON	Default with OFF	Compilation options
A		X	Aborts job after compilation if any of the program units contains an error.
B	X		Beginning sequence number of each vectorized code generation block (G implies B)
C	X		Common block names and lengths listed on <i>ldn</i> after each program unit
E	X		Enable recognition of compiler directive lines
G		X	Generated code for each program unit
I		X	Symbol table contains internal compiler-generated statement labels.
L	X		Enable recognition of listable output control directives
N		X	Symbol table contains null symbols (defined but not referenced)
Q	X		Aborts compilation when 100 warning and fatal error messages counted
S	X		FORTRAN source code
T	X		Symbol table after each program unit
W		X	Warning messages concerning non-standard or unusual code
X		X	Symbol table with cross references after each program unit (X overrides T)
V	X		Vectorization of inner DO-loops (for timing purposes)

(See Section 5 for a description of compiler directives.)

6.3.4 SET OR CLEAR SENSE SWITCH

The SWITCH control statement allows a user to turn on or turn off a pseudo sense switch.

In a FORTRAN job, any setting of pseudo sense switches must be performed before the system loader (LDR) is executed (i.e., SWITCH statements should precede the LDR statement). Pseudo sense switch settings are available to a FORTRAN program referencing the SSWITCH utility procedure (see Appendix C).

6.3.5 RELOCATABLE LOADER

The LDR job control statement causes the execution of LDR, the CRAY-1 relocatable loader. LDR executes within the user field and provides for loading and linking of relocatable modules from datasets on mass storage into the memory of the CPU. LDR initializes only those entities specified with DATA statements. All others remain undefined (i.e., are not default-initialized to zero).

The system relocatable loader is called through the LDR control statement. Absolute load modules can also be loaded. The design of the COS loader tables and relocatable loader allows program modules to be loaded, relocated and linked to externals in a single pass over the dataset being loaded. This minimizes the time spent in loading activities on the CRAY-1. The loader allows the immediate execution of the object module or the creation of an absolute binary image of the object module on a specified dataset. Loader features are governed by parameters of the LDR control statement. Loader input is assumed to be on a dataset named \$BLD unless otherwise specified.

6.3.6 THE JOB CONTROL STATEMENT FILE

A job control statement deck prepared by the CRAY-1 FORTRAN programmer becomes the first file of the job dataset contained in the CRAY-1. Below is a simplified example of a job control deck:

```
JOB(JN=MYTEST,M=100,T=70)
ASSIGN(DN=$IN)
ASSIGN(DN=$OUT)
CFT.
LDR.
```

6.4 ERROR MESSAGES DURING PROGRAM EXECUTION

While under COS control, the executable program calls upon numerous system routines to accomplish certain of its mathematical, input/output and utility operations. These are identified as required during compilation of the program, are loaded and linked to it by the system loader (LDR), and are contained in the system library (\$FTLIB). When in use, these routines will respond to discrepancies in programming and/or certain equipment situations with messages placed on the jobfile and on the \$OUT dataset. Conditions causing these messages also cause the job to abort.

A description of these messages and their meanings appears in Appendix E.

APPENDIXES

CHARACTER SET

A

This appendix describes the 128 control and graphic characters comprising the ASCII character set. Those numbers, letters, and special characters that form the CRAY-1 FORTRAN character set are identified by the appearance of the letter C in the fourth column. All other characters are members of the auxiliary character set. The letter A in the fourth column of the table indicates those characters belonging to the ANSI FORTRAN character set. Note that all control characters are grouped on the first page.

CONTROL CHARACTER	ASCII OCTAL CODE	ASCII PUNCHED-CARD CODE	FORTRAN (A=ANSI) (C=CRAY)	DESCRIPTION
NUL	000	12-0-9-8-1		Null
SOH	001	12-9-1		Start of heading (CC)
STX	002	12-9-2		Start of text (CC)
ETX	003	12-9-3		End of text (CC)
EOT	004	9-7		End of transmission (CC)
ENQ	005	0-9-8-5		Enquiry (CC)
ACK	006	0-9-8-6		Acknowledge (CC)
BEL	007	0-9-8-7		Bell (audible or attention signal)
BS	010	11-9-6		Backspace (FE)
HT	011	12-9-5		Horizontal tabulation (FE)
LF	012	0-9-5		Line feed (FE)
VT	013	12-9-8-3		Vertical tabulation (FE)
FF	014	12-9-8-4		Form feed (FE)
CR	015	12-9-8-5		Carriage return (FE)
SO	016	12-9-8-6		Shift out
SI	017	12-9-8-7		Shift in
DLE	020	12-11-9-8-1		Data link escape (CC)
DC1	021	11-9-1		Device control 1
DC2	022	11-9-2		Device control 2
DC3	023	11-9-3		Device control 3
DC4	024	9-8-4		Device control 4 (stop)
NAK	025	9-8-5		Negative acknowledge (CC)
SYN	026	9-2		Synchronous idle (CC)
ETB	027	0-9-6		End of transmission block (CC)
CAN	030	11-9-8		Cancel
EM	031	11-9-8-1		End of medium
SUB	032	9-8-7		Substitute
ESC	033	0-9-7		Escape
FS	034	11-9-8-4		File separator (IS)
GS	035	11-9-8-5		Group separator (IS)
RS	036	11-9-8-6		Record separator (IS)
US	037	11-9-8-7		Unit separator (IS)
DEL	177	12-9-7		Delete

Legend: CC - Communication control
 FE - Format effector
 IS - Information separator

GRAPHIC CHARACTER	ASCII OCTAL CODE	ASCII PUNCHED-CARD CODE	FORTRAN (A=ANSI) (C=CRAY)	DESCRIPTION
(Space)	040	(None)	A,C	Space (blank)
!	041	12-8-7		Exclamation point
"	042	8-7	C	Quotation marks (diaeresis)
#	043	8-3		Number sign
\$	044	11-8-3	A,C	Dollar sign (currency symbol)
%	045	0-8-4		Percent
&	046	12		Ampersand
'	047	8-5	C	Apostrophe (closing single quotation mark)
(050	12-8-5	A,C	Opening (left) parenthesis
)	051	11-8-5	A,C	Closing (right) parenthesis
*	052	11-8-4	A,C	Asterisk
+	053	12-8-6	A,C	Plus
,	054	0-8-3	A,C	Comma (cedilla)
-	055	11	A,C	Minus (hyphen)
.	056	12-8-3	A,C	Period (decimal point)
/	057	0-1	A,C	Slant (slash, virgule)
0	060	0	A,C	Zero
1	061	1	A,C	One
2	062	2	A,C	Two
3	063	3	A,C	Three
4	064	4	A,C	Four
5	065	5	A,C	Five
6	066	6	A,C	Six
7	067	7	A,C	Seven
8	070	8	A,C	Eight
9	071	9	A,C	Nine
:	072	8-2	C	Colon
;	073	11-8-6		Semicolon
<	074	12-8-4		Less than
=	075	8-6	A,C	Equal
>	076	0-8-6		Greater than
?	077	0-8-7		Question mark

GRAPHIC CHARACTER	ASCII OCTAL CODE	ASCII PUNCHED-CARD CODE	FORTRAN (A=ANSI) (C=CRAY)	DESCRIPTION
@	100	8-4		Commercial at
A	101	12-1	A,C	
B	102	12-2	A,C	
C	103	12-3	A,C	
D	104	12-4	A,C	
E	105	12-5	A,C	
F	106	12-6	A,C	
G	107	12-7	A,C	
H	110	12-8	A,C	
I	111	12-9	A,C	
J	112	11-1	A,C	
K	113	11-2	A,C	
L	114	11-3	A,C	
M	115	11-4	A,C	
N	116	11-5	A,C	Upper-case letters
O	117	11-6	A,C	
P	120	11-7	A,C	
Q	121	11-8	A,C	
R	122	11-9	A,C	
S	123	0-2	A,C	
T	124	0-3	A,C	
U	125	0-4	A,C	
V	126	0-5	A,C	
W	127	0-6	A,C	
X	130	0-7	A,C	
Y	131	0-8	A,C	
Z	132	0-9	A,C	
[133	12-8-2		Opening (left) bracket
\	134	0-8-2		Reverse slant (backslash)
]	135	11-8-2		Closing (right) bracket
^	136	11-8-7		Circumflex
-	137	0-8-5		Underline

GRAPHIC CHARACTER	ASCII OCTAL CODE	ASCII PUNCHED-CARD CODE	FORTRAN (A=ANSI) (C=CRAY)	DESCRIPTION
'	140	8-1		Grave accent (opening single quotation mark)
a	141	12-0-1		
b	142	12-0-2		
c	143	12-0-3		
d	144	12-0-4		
e	145	12-0-5		
f	146	12-0-6		
g	147	12-0-7		
h	150	12-0-8		
i	151	12-0-9		
j	152	12-11-1		
k	153	12-11-2		
l	154	12-11-3		
m	155	12-11-4		
n	156	12-11-5		
o	157	12-11-6		
p	160	12-11-7		
q	161	12-11-8		
r	162	12-11-9		
s	163	11-0-2		
t	164	11-0-3		
u	165	11-0-4		
v	166	11-0-5		
w	167	11-0-6		
x	170	11-0-7		
y	171	11-0-8		
z	172	11-0-9		
{	173	12-0		Opening (left) brace
:	174	12-11		Vertical line
}	175	11-0		Closing (right) brace
~	176	11-0-1		Overline (tilde, general accent)

CRAY-1 FORTRAN INTRINSIC FUNCTIONS

B

The CRAY-1 FORTRAN intrinsic functions described in this appendix have been grouped according to general purpose (i.e., trigonometric, exponential, etc.). This grouping is for convenience and is not provided for in the ANSI FORTRAN Standard. Note that these intrinsic functions include all "Basic External Functions" as are described separately in ANSI FORTRAN.

FUNCTION	DEFINITION	FUNCTION (y)		ARGUMENT(S) (x)			NOTES
		NAME	TYPE	NO.	TYPE	RANGE	

TRIGONOMETRIC FUNCTIONS (Angles are in radians)

Sine	$y = \sin(x)$	SIN	R	1	R	$ x < 2^{24}$	
		DSIN	D		D	$ x < 2^{48}$	
		CSIN	C		C	$ x_r < 2^{24},$ $ x_i < 2^{12} \cdot \ln 2$	$x = x_r + i \cdot x_i$
Cosine	$y = \cos(x)$	COS	R	1	R	$ x < 2^{24}$	
		DCOS	D		D	$ x < 2^{48}$	
		CCOS	C		C	$ x_r < 2^{24},$ $ x_i < 2^{12} \cdot \ln 2$	$x = x_r + i \cdot x_i$
Tangent	$y = \tan(x)$	TAN	R	1	R	$ x < 2^{24}$	Non-ANSI
Cotangent	$y = \cot(x)$	COT	R	1	R	$ x < 2^{24}$	Non-ANSI
Arcsine	$y = \arcsin(x)$	ASIN	R	1	R	$ x \leq 1$	Non-ANSI
Arccosine	$y = \arccos(x)$	ACOS	R	1	R	$ x \leq 1$	Non-ANSI
Arctangent	$y = \arctan(x)$	ATAN	R	1	R	$ x < \infty$	
		DATAN	D		D		
	$y = \arctan(x_1/x_2)$	ATAN2	R	2	R	$ x_1 < \infty,$ $0 < x_2 < \infty$	
Hyperbolic sine	$y = \sinh(x)$	SINH	R	1	R	$ x < 2^{12} \cdot \ln 2$	Non-ANSI
Hyperbolic cosine	$y = \cosh(x)$	COSH	R	1	R	$ x < 2^{12} \cdot \ln 2$	Non-ANSI
Hyperbolic tangent	$y = \tanh(x)$	TANH	R	1	R	$ x < 2^{12} \cdot \ln 2$	

LOGARITHMIC FUNCTIONS

Natural logarithm	$y = \ln(x)$	ALOG	R	1	R	$0 < x < \infty$	
		DLOG	D		D		
		CLOG	C		C	$0 < x < \infty$	$x = x_r + i \cdot x_i$
Common logarithm	$y = \log(x)$	ALOG10	R	1	R	$0 < x < \infty$	
		DLOG10	D		D		

EXPONENTIAL FUNCTIONS

Square root	$y = x^{\frac{1}{2}}$	SQRT	R	1	R	$0 \leq x < \infty$	
		DSQRT	D		D		
		CSQRT	C		C	$x_r, x_i < \infty$	$x = x_r + i \cdot x_i$
Exponent	$y = e^x$	EXP	R	1	R	$ x < 2^{12} \cdot \ln 2$	
		DEXP	D		D		
		CEXP	C		C	$ x_r < 2^{12} \cdot \ln 2,$ $ x_i < 2^{24}$	$x = x_r + i \cdot x_i$

The ANSI FORTRAN Standard does not provide for TAN, COT, ASIN, ACOS, SINH or COSH.

FUNCTION	DEFINITION	FUNCTION (y)		ARGUMENT(S) (x)			NOTES
		NAME	TYPE	NO.	TYPE	RANGE	

GENERAL ARITHMETIC FUNCTIONS

Absolute value	$y = x $	IABS	I	1	I	$ x < \infty$	
		ABS	R		R		
		DABS	D		D		
		CABS	R		C	$ x_r , x_i < \infty$	$x = x_r + i \cdot x_i$
Positive difference	$y = x_1 - x_2 \text{ if } x_1 > x_2$ $y = 0 \text{ if } x_1 \leq x_2$	IDIM	I	2	I	$ x_1 , x_2 < \infty$	$x_1 \geq x_2$
		DIM	R		R		
Truncate to integral value	$y = \lfloor x \rfloor$	INT	I	1	R	$ x < 2^{46}$	Fraction lost; no rounding
		AINT	R		D		
		IDINT	I		I		
Divide for remainder only	$y = x_1 - x_2 \cdot \lfloor x_1/x_2 \rfloor$	MOD	I	2	R	$ x_1 < \infty, 0 < x_2 < \infty$	Remainder of x_1/x_2
		AMOD	R		R		
		DMOD	D		D		
Transfer sign	$y = x_1 \text{ if } x_2 \geq 0 \text{ or}$ $y = - x_1 \text{ if } x_2 < 0$	ISIGN	I	2	I	$ x_1 , x_2 < \infty$	
		SIGN	R		R		
		DSIGN	D		D		
Obtain random number	$y = \begin{cases} \text{the first or next in} \\ \text{a series of random} \\ \text{numbers (0} < y < 1\}) \end{cases}$	RANF	R	0	-	-	Non-ANSI
Obtain random number seed	$y = \text{the currently-used random number seed}$	RANGT	R	0	-	-	Non-ANSI
Establish random number seed	$y = x$	RANSET	R	1	I	$ x < \infty$	Non-ANSI
Conjugate complex value	$y = x_r - i \cdot x_i$	CONJG	C	1	C	$ x_r , x_i < \infty$	$x = x_r + i \cdot x_i$
Real portion of complex value	$y = x_r$	REAL	R	1	C	$ x_r , x_i < \infty$	$x = x_r + i \cdot x_i$
Imaginary portion of complex value	$y = x_i$	AIMAG	R	1	C	$ x_r , x_i < \infty$	$x = x_r + i \cdot x_i$

The ANSI FORTRAN Standard does not provide for RANF, RANGT or RANSET.

FUNCTION	DEFINITION	FUNCTION (y)		ARGUMENT(S) (x)			NOTES
		NAME	TYPE	NO.	TYPE	RANGE	

MAXIMUM / MINIMUM FUNCTIONS

Select maximum value	$y = \text{the largest of all } x$	MAXØ	I	≥ 2	I	$ x < \infty$	
		AMAXØ	R				
		MAX1	I		R		
		AMAX1	R		D		
		DMAX1	D				
Select minimum value	$y = \text{the smallest of all } x$	MINØ	I	≥ 2	I	$ x < \infty$	
		AMINØ	R				
		MIN1	I		R		
		AMIN1	R		D		
		DMIN1	D				

TYPE CONVERSION FUNCTIONS

Integer to real	$y=x$	FLOAT	R	I	I	$ x < 2^{16}$	
Real to integer	$y=[x]$	IFIX	I	1	R	$ x < 2^{46}$	Fraction lost; no rounding
Real to double precision	$y=x$	DBLE	D	1	R	$ x < \infty$	Accuracy may be extended
Double precision to real	$y=x$	SNGL	R	1	D	$ x < \infty$	Accuracy may be reduced
Real to complex	$y=x_1+i\cdot x_2$	CMPLX	C	2	R	$ x_1 , x_2 < \infty$	$y=x_r+i\cdot x_i$

CRAY-1 FORTRAN UTILITY PROCEDURES

C

The CRAY-1 FORTRAN Compiler (CFT) includes a set of utility procedures which, like intrinsic functions, are predefined by name and function. Unlike intrinsic functions, however, utility procedures are not provided for by the ANSI FORTRAN Standard. They include subroutines as well as functions, may have arguments of mixed type, and may modify these arguments' contents.

The following table describes currently available utility procedures. Entity types in this table are abbreviated: integer (I), real (R), double precision (D), complex (C), logical (L), and Boolean (B).

The ANSI FORTRAN Standard does not provide for utility procedures.

SUBROUTINE NAME		DEFINITION	ARGUMENT TYPE(S)			NOTES
FUNCTION (y)	NAME		x_1	x_2	x_3	
RTC	R	y = current clock register content	-	-	-	
TIMEF	R	y = current clock register content in milliseconds	-	-	-	
SECOND		$x_1 = y$ = cumulative CPU time for job in seconds	R	-	-	
SECOND	R					
CLOCK		$x_1 = y$ = current time in ASCII code (hh:mm:ss)	I,R	-	-	
CLOCK	R					
DATE		$x_1 = y$ = current date in ASCII code (mm:dd:yy)	I,R	-	-	
DATE	R					
JDATE		$x_1 = y$ = current date (Julian) in ASCII code (yydd)	I,R	-	-	
JDATE	R					
SYSTEM	I	$y = f(x_1)$ where $0 \leq x_1 \leq 31$ to select one of 32 COS functions	I	-	-	See the "Exchange Processor" section of CRI pub. 2240012
EXIT		Job termination	-	-	-	Does not return control to job
LOC	I	y = memory address of variable or array specified for x_1	B,I,R D,C,L	-	-	The identity of x_1 (and not its content) is the argument
SSWITCH		$x_2 = 1$ if sense switch x_1 is ON $x_2 = 2$ if sense switch x_1 is OFF, if $x_1 < 1$, or if $x_1 > 6$	I	I	-	
EODW		Writes EOD and, as required, EOF and EOR record(s) on unit x_1	I	-	-	$x_1 = \begin{cases} \text{a unit number} \\ \text{or a Hollerith unit name} \end{cases}$
IEOF	I	$y = -1$ if EOD processed at unit x_1 $y = +1$ if EOF read at unit x_1 $y = 0$ otherwise	I	-	-	$x_1 = \begin{cases} \text{a unit number} \\ \text{or a Hollerith unit name} \end{cases}$
UNIT	R	$y = -2.0$ if record at unit x_1 part read $y = -1.0$ if unit x_1 transfer successful $y = 0.0$ if EOF or EOD read at unit x_1 $y = +1.0$ if disk parity error while reading unit x_1 $y = +2.0$ if unit x_1 error indicated	I	-	-	Applies only to buffered input/output operations $x_1 = \begin{cases} \text{a unit number} \\ \text{or a Hollerith unit name} \end{cases}$
LENGTH	I	y = words transferred to or from unit x_1 $y = 0$ if EOF or EOD read from unit x_1	I	-	-	Applies only to buffered input/output operations $x_1 = \begin{cases} \text{a unit number} \\ \text{or a Hollerith unit name} \end{cases}$
GETPOS	I	y = starting address of current record in dataset associated with unit x_1	I	-	-	$x_1 = \begin{cases} \text{a unit number} \\ \text{or a Hollerith unit name} \end{cases}$
SETPOS		Set starting address of current record in dataset associated with unit x_1 to beginning address of dataset if $x_2 = 0$, to ending address if $x_2 = -1$, or to word address x_2	I	I	-	$x_1 = \begin{cases} \text{a unit number} \\ \text{or a Hollerith unit name} \end{cases}$

SUBROUTINE NAME		DEFINITION	ARGUMENT TYPE(S)			NOTES	
FUNCTION (y)			x_1	x_2	x_3		
NAME	TYPE						
AND	B	$y = \text{Bit-by-bit logical AND}$ (logical product) of x_1 and x_2	B,I, R,L	B,I, R,L	-		
OR	B	$y = \text{Bit-by-bit logical OR}$ (logical sum) of x_1 and x_2	B,I, R,L	B,I, R,L	-		
XOR	B	$y = \text{Bit-by-bit exclusive OR}$ (logical difference) of x_1 and x_2	B,I, R,L	B,I, R,L	-		
EQV	B	$y = \text{Bit-by-bit equivalence}$ (XOR) of x_1 and x_2	B,I, R,L	B,I, R,L	-		
COMPL	B	$y = \text{Bit-by-bit logical complement}$ of x_1	B,I, R,L	-	-		
MASK	B	$y = x_1$ left-justified 1-bits if $0 \leq x_1 \leq 63$ $y = (128-x_1)$ right-justified 1-bits if $64 \leq x_1 \leq 128$	I	-	-		
SHIFT	B	x_1 shifted left x_2 positions with $y = \text{leftmost } x_2 \text{ positions of } x_1 \text{ replacing}$ those positions vacated	B,I, R,L	I	-	Circular or cyclic shift	
SHIFTL	B	x_1 shifted left x_2 positions with $y = \text{leftmost } x_2 \text{ positions of } x_1 \text{ lost and}$ rightmost x_2 positions set to zero	B,I, R,L	I	-	Logical shift	
SHIFTR	B	x_1 shifted right x_2 positions with $y = \text{rightmost } x_2 \text{ positions of } x_1 \text{ lost and}$ leftmost x_2 positions set to zero	B,I, R,L	I	-	Logical shift	
LEADZ	I	$y = \text{number of leading zeros in } x_1$	B,I, R,L	-	-		
POPCNT	I	$y = \text{number of ones in } x_1$	B,I, R,L	-	-		
CVMGP	R	$y = x_1 \text{ if } x_3 \geq 0$ $y = x_2 \text{ if } x_3 < 0$	R	R	R		
CVMGM	R	$y = x_1 \text{ if } x_3 < 0$ $y = x_2 \text{ if } x_3 \geq 0$	R	R	R		
CVMGZ	R	$y = x_1 \text{ if } x_3 = 0$ $y = x_2 \text{ if } x_3 \neq 0$	R	R	R		
CVMGN	R	$y = x_1 \text{ if } x_3 \neq 0$ $y = x_2 \text{ if } x_3 = 0$	R	R	R		
CSMG	B	$y = (x_2 \cap x_3) \cup (x_1 \cap \bar{x}_3)$	B,I, R,L	B,I, R,L	B,I, R,L	Bit-by-bit selective merge	

CFT ERROR MESSAGES

D

Error messages produced by CFT, the CRAY-1 FORTRAN compiler, are described in this appendix. Errors which might cause faulty execution of the compiled program issue warning messages. Errors that would definitely cause program failure or that make further compilation impossible issue fatal error messages. Both types of messages are grouped under appropriate headings in the following pages.

WARNING MESSAGES

FEWER SUBSCRIPTS THAN DECLARED

A reference to an actual array element has fewer subscript expressions in its subscript than dimension declarators in the corresponding array declarator. The missing subscript expressions are assumed rightmost in the subscript and are each assigned the value 1 by the compiler.

LAST ARRAY IN DATA LIST NOT FILLED

The last element in a DATA statement variable list is an unsubscripted array and there are not enough constants specified to completely fill the array. Remaining elements of the array are not initialized.

LOSS OF PRECISION IN DEFINITION

A constant in a PARAMETER statement must be converted to the type of the corresponding symbolic name resulting in possible loss of precision.

LOSS OF PRECISION IN TYPE CONVERSION

The type of a variable and the type of the associated constant in a DATA statement differ. The constant is converted to the type of the variable and precision is lost.

MISSING END STATEMENT

The last or only program unit being compiled lacks an END statement in its last line.

NO PATH TO THIS STATEMENT

The previous statement is an unconditional transfer and this statement has no statement number.

PUNCHES IN COL 1 THRU 5 IGNORED

Punches appear in the statement number field of a continuation card and are ignored.

STATEMENT IS NONSTANDARD

A statement is not an ANSI FORTRAN statement but is supported by CFT.

TYPE CONVERSION IN DATA STATEMENT

The type of a variable and the type of the associated constant in a DATA statement differ. The constant is converted to the type of the variable.

TYPE CONVERSION IN DEFINITION

A constant in a PARAMETER statement must be converted to the type of the corresponding symbolic name.

FATAL ERROR MESSAGES

ARRAY CANNOT BE DECLARED EXTERNAL

An array named in an array declarator is also specified in an EXTERNAL statement.

COMMA EXPECTED

A required comma has not been specified. The name of a source or destination variable, array element, or array is not present in an ENCODE or DECODE statement.

COMMON BLOCK NAME MUST BE // OR /NAME/

The specification of a common block name does not conform to the rules for constructing symbolic names.

COMPILER ERROR

A known inadequacy in the CFT compiler. (Please show your listing to a Cray systems analyst.)

COMPILING TERMINATED DUE TO EXCESSIVE ERRORS

Compilation of the current program unit is concluded due to the large number of errors encountered. This message appears in the COS logfile.

CONFLICT IN NAME USAGE

A constant name in a PARAMETER statement has been previously defined.

CONSTANT LIST EXCEEDS VARIABLE LIST

There is not a one-to-one relationship between constants and variables in a DATA statement.

CONSTANT LIST IN DATA STATEMENT BAD

A DATA statement is missing a constant list or contains an illegal separator character.

CONTROL CARD ERROR

The compiler cannot interpret information in the CFT control statement. This message appears in the COS logfile.

COUNT > 8 ONLY WITH H OR L IN DATA OR ARGUMENT LIST

An R-form Hollerith constant is specified with more than 8 characters, or an H- or L-form Hollerith constant is specified with more than 8 characters in other than a DATA statement or an actual argument list.

DATA IN BLANK COMMON IS ILLEGAL

The DATA statement cannot be used to initialize blank common.

DIMENSION EXCEEDED ON DATA STATEMENT

A subscript in a DATA statement element exceeds the corresponding array declaration.

DIVIDE BY 0

Dividing by the constant 0 is illegal.

DO TERMINATOR PRECEDES DO STATEMENT

The statement label that terminates a DO-loop precedes the corresponding DO statement.

DO VARIABLE ACTIVE OR IMPROPER

The loop control variable is either already active from a previous loop or is incorrectly specified.

DOUBLY DEFINED FUNCTION

Statement functions cannot be defined more than once in a program unit.

DOUBLY DEFINED STATEMENT NUMBER

Statement labels cannot be defined more than once in a program unit.

DUMMY ARGUMENT IN EQUIVALENCE STATEMENT

A dummy argument in a procedure subprogram cannot be named the same as a local variable or another dummy argument.

DUPLICATE COMMON DEFINITION

Common cannot be defined more than once.

DUPLICATE DIMENSION OR FUNCTION DECLARATION

Dimensions and functions cannot be declared more than once.

DUPLICATE DUMMY ARGUMENT NAMES

A symbolic name appears twice in the dummy argument list of a FUNCTION or SUBROUTINE statement.

DUPLICATE TYPE DEFINITION

Variables cannot be given more than one type.

EBCDIC NOT IMPLEMENTED

The current version of CFT allows only ASCII characters.

ERROR IN DATA CONSTANT

The elements in the constant list of a DATA statement must be constants or constant names.

ERROR IN SFN PARAMETER LIST

The formal parameter list in an arithmetic statement function definition statement contains an illegal element.

ERROR NEAR >>>cccccccc<<<

An error has been detected within or near the eight listed characters from a FORMAT statement.

EXTERNAL/VARIABLE NAME CONFLICT

A name has been used for both a variable and an external procedure.

EXTRA ITEM AFTER LEGAL END OF STATEMENT

Characters are specified after the syntactic end of a statement.

FATAL ERROR IN DECLARATIVES

A syntax error in a specification statement, an ambiguously declared entity, or other syntactic or statement sequence error.

FUNCTION MUST HAVE A PARAMETER LIST

A FUNCTION statement lacks a parenthesized list of dummy arguments, required even if the list is empty (in which case the notation "()" must be used).

FUNCTION NAME IN COMMON OR DIMENSION STATEMENT

A function name cannot be used as an array name.

FUNCTION OR CALL REFERENCES ITSELF

A reference to the function or subroutine subprogram being compiled is encountered with that subprogram.

H,L,R COUNT .LE. ZERO

In an nH , nL , or nR specification of a Hollerith value, n is less than or equal to zero.

H,L,R COUNT PAST END OF STATEMENT

In an nH , nL , or nR specification of a Hollerith value, n specifies more characters than are provided. This message also indicates that an apostrophe terminating a Hollerith string is not specified.

HOLLERITH CONSTANT > 8 CHARACTERS

A Hollerith constant of more than 8 characters is specified in other than H- or L-form and in other than an actual argument list or a DATA statement constant.

ILLEGAL CHARACTER

A nonstandard FORTRAN character, misplaced character, or syntax error has been encountered.

ILLEGAL CHARACTER IN NAME FIELD

An illegal character is found in a field that must contain a symbolic name.

ILLEGAL CHARACTER IN NUMBER FIELD

An alphabetic character appears in what should be a numeric field.

ILLEGAL COMPLEX RELATION

Relational operator for complex operands must be .EQ. or .NE. .

ILLEGAL CONDITIONAL STATEMENT

This type of statement is not allowed as the conditional statement of a direct logical IF statement.

ILLEGAL CONTINUATION

More than 19 consecutive continuation cards encountered, or the first card of a program unit is a continuation card.

ILLEGAL DATA REPETITION FIELD

The repetition count in a DATA constant list must be an integer greater than 0.

ILLEGAL DATA TYPE CONVERSION

The type of a variable and associated constant in a DATA statement differ. The type conversion required is illegal or undefined.

ILLEGAL DO TERMINATOR

DO loops must not terminate on unconditional transfer statements.

ILLEGAL FORMAT NAME

A format identifier cannot be recognized as a statement label or the name of an array.

ILLEGAL MIXED MODE OR CONVERSION

The types of two operands in an expression are incompatible, or the type of array element or variable being defined is incompatible with the type of expression being evaluated.

ILLEGAL STATEMENT SEQUENCE

An improper sequence of statement types has been encountered (i.e., a GO TO statement followed by a DIMENSION statement).

ILLEGAL STATEMENT TYPE

A statement keyword is misspell (e.g., DIMENSOIN) or is otherwise unidentifiable.

ILLEGAL SUBSCRIPT IN DATA STATEMENT

Subscripts in a DATA statement must contain only constants or constant names and must be greater than zero.

ILLEGAL SUBSCRIPT IN DECLARATIVE

A dimension declarator must be an integer constant or constant name or a dummy argument, and must be greater than zero.

ILLEGAL TERM IN CONSTANT EXPRESSION

A constant expression in a PARAMETER or DATA statement is specified with other than constants or the symbolic names of constants. A constant expression in a DATA statement is specified with other than constants, the symbolic names of constants, or the names of implied-DO variables.

ILLEGAL TYPE IN SUBSCRIPT EXPRESSION

A subscript expression is not of type integer or contains a constant that exceeds $2^{24}-1$.

ILLEGAL TYPE NAME

A type or IMPLICIT statement contains a keyword other than INTEGER, REAL, LOGICAL, COMPLEX, DOUBLE PRECISION, or DOUBLE.

ILLEGAL UNIT NAME

An input/output unit identifier cannot be recognized as an integer value, a symbolic name, or a Hollerith character string specified in H-form.

ILLEGAL USE OF ** IN CONSTANT EXPRESSION

A constant expression specifies exponentiation to a non-integer power.

ILLEGAL V ITEM

The name of a source or destination variable, array element, or array in an ENCODE or DECODE statement is not recognizable as such.

ILLEGAL VALUE IN CONSTANT EXPRESSION

The evaluation of a constant expression yields a result that is out of range.

ILLEGAL VARIABLE TYPE

A variable referenced in an ASSIGN or in a computed GO TO statement is not of type integer.

INCORRECT ARGUMENT TYPE

Actual argument is of the wrong type in a function reference.

INCORRECT NUMBER OF DO ARGUMENTS

More than three arguments have been encountered after the equal sign in a DO statement.

INPUT FILE EMPTY

An end-of-file record was encountered as the first record of the source input dataset.

LINE LENGTH EXCEEDS 152

One or more lines exceed 152 characters during FORMAT statement editing.

MISSING EQUAL SIGN

An equal sign is missing in a PARAMETER or statement function definition statement.

MISSING LEFT PARENTHESIS

Opening and closing parentheses do not match or required parenthesis not present.

MISSING PERIOD

A delimiting period is missing from a logical operator or logical constant.

MISSING RIGHT PARENTHESIS

Opening and closing parentheses do not match or required parenthesis not present.

MORE SUBSCRIPTS THAN DECLARED

An array reference has more subscripts than were declared.

MORE THAN 7 DIMENSIONS

CFT does not support arrays having more than seven dimensions.

NO CLOSING)

The closing parenthesis in a FORMAT statement is missing.

NON-NUMERIC PUNCH IN COLUMNS 1-5

A statement label specified in columns 1-15 of an initial line contains other than digits and blanks.

NOT ENOUGH MEMORY TO COMPILE

The program unit is too long to compile in the available memory.

PARAMETER DOUBLE DEFINED

A name may be given only one value in a PARAMETER statement.

RETURN NOT PERMITTED IN MAIN PROGRAM

A RETURN statement is encountered in a main program unit.

SFN REFERENCES ITSELF

A statement function definition statement cannot be recursive.

STATEMENT NUMBER MISSING

An ASSIGN statement lacks a statement label reference.

STATEMENT TEMPORARILY NOT IMPLEMENTED

A statement is recognized but is not implemented as yet in the CRAY-1 compiler.

SUBSCRIPT EXCEEDS DIMENSION BOUNDS

An implied-DO list specified in a DATA statement contains a subscript which, when evaluated, yields a result that is less than one or greater than the size of the named array.

SYNTAX ERROR

Illegal element, name where number required, extra or missing punctuation.

SYNTAX ERROR IN DATA IMPLIED DO

An implied-DO list specified in a DATA statement is of improper syntactical form, references a variable that is not an implied-DO variable, or references an array element that does not specify the implied-DO variable for this implied-DO list in its subscript.

TOO MANY ERRORS IN FORMAT, SCAN STOPPED

CFT attempts recovery from up to three errors before abandoning the FORMAT statement.

UNBALANCED PARENTHESES

Opening and closing parentheses do not match; required parenthesis not present.

UNDEFINED ASSIGNED SN

A statement label referenced in an ASSIGN statement is not defined.

UNDEFINED FORMAT

A referenced FORMAT statement is not defined.

UNDEFINED STATEMENT NUMBER

A referenced statement label is not defined.

UNDEFINED TYPE CONVERSION

A constant expression specified in a PARAMETER statement cannot be converted to the type of the corresponding constant name.

UNEXPECTED END OF STATEMENT

A statement encountered that is syntactically incomplete.

VARIABLE LIST EXCEEDS CONSTANT LIST

There must be a one-to-one correspondence between constants and variables in a DATA statement.

COS ERROR MESSAGES

E

During COS control of executable program execution, certain messages will be listed in the logfile and in the \$OUT dataset to describe certain programming and/or equipment discrepancies. These are listed alphabetically in this appendix.

ARGUMENT ERROR

An actual argument processed by a specific library routine is outside its allowable bounds. The name of the routine appears in the COS logfile.

BLOCK NUMBER ERROR

Data error encountered in using the mass storage subsystem.

BUFFER IN WORD COUNT < 1

Final transfer location in BUFFER IN statement must be greater than or equal to initial transfer location.

BUFFER OUT WORD COUNT < 0

Final transfer location in BUFFER OUT statement must be greater than or equal to initial transfer location, or one less than initial transfer location if an end-of-record is to be produced following a partial write operation

END OF *name*

An END statement was executed in FORTRAN program unit *name*.
(Processing continues with the next job control statement.)

ERROR ON OPEN

A called-for dataset cannot be opened.

EXIT CALLED BY *name*

A CALL to utility procedure EXIT was executed in FORTRAN program unit *name*. (Processing continues with the next job control statement.)

FIELD SPECIFICATION MISSING

A required value is missing after a format specification edit descriptor. The applicable format specification is listed in the \$OUT dataset.

FILE DOES NOT EXIST

The required file has not been assigned to this job.

FILE NOT OPEN

The required file cannot be written on.

FORMAT MUST BEGIN WITH (

A format specification is incorrect and/or has been destroyed during program execution.

ILLEGAL FIELD WIDTH

The field width specified for a format specification edit descriptor is in error. The applicable format specification is listed in the \$OUT dataset.

ILLEGAL FORMAT CHARACTER

An illegal character has been encountered in a format specification. The applicable format specification is listed in the \$OUT dataset.

ILLEGAL LOGICAL VALUE

A record being input does not have the letter T or the letter F in a field specified as containing a logical value. The applicable format specification is listed in the \$OUT dataset.

ILLEGAL SEQUENCE OF CHARACTERS

Characters encountered in processing a format specification are not allowed in the order used. The applicable format specification is listed in the \$OUT dataset.

IOERP UNABLE TO WRITE IN \$OUT

The Input/Output Error Processor (IOERP) is unable to write an error message on the \$OUT dataset; the \$OUT DSP is destroyed.

I/O SYSTEM ERROR - SEE ANALYST

An unidentified system error has occurred.

NUMERIC: EXPONENT OVERFLOW

An exponent value exceeds the range of legal values in the CRAY-1 during the conversion of values input under format control. The applicable format specification is listed on the \$OUT dataset.

NUMERIC: EXPONENT UNDERFLOW

An exponent value exceeds the range of legal values in the CRAY-1 during the conversion of values input under format control. The applicable format specification is listed on the \$OUT dataset.

NUMERIC: ILLEGAL CHARACTER

An illegal character is encountered in a numeric field being converted under format control. The applicable format specification is listed in the \$OUT dataset.

NUMERIC: NULL FIELD

An input field is encountered during format control processing which illegally contains all blank characters. The applicable format specification is listed in the \$OUT dataset.

NUMERIC: OVERFLOW

A value being input exceeds the range of legal values in the CRAY-1 during its conversion under format control. The applicable format specification is listed on the \$OUT dataset.

PAUSE NOT SUPPORTED, STOP SUBSTITUTED

The PAUSE statement is the same as a STOP statement in CRAY-1 FORTRAN.

READ AFTER WRITE

Reading a record immediately after writing a record on the same file is not permitted. All information following the record written is destroyed.

READ CALL OUT OF SEQUENCE

A sequence of input operations is improperly structured.

READ PAST END OF DATA

An input operation has been attempted which goes beyond the physical end of dataset.

RECORD LENGTH EXCEEDED

An input operation has attempted reading more information than contained in the record being read. The applicable format specification is listed on the \$OUT dataset.

STOP *nnnnn* IN *name*

A STOP *nnnnn* statement was executed in FORTRAN program unit *name*. Processing continues with the next job control statement.

SYSTEM ERROR - EMPTY INPUT BUFFER

An input operation attempts to read a file that has never been written or that is missing from the last sector of the mass storage subsystem.

UNCLEARED END OF FILE

An input operation has attempted reading beyond an end-of-file just read without an intervening test for its presence.

UNFORMATTED LIST EXCEEDS RECORD LENGTH

An unformatted input operation calls for more data than is contained in the current record.

UNIT NO. > 102 or < 0

An out-of-range unit number has been specified.

UNMATCHED OR TOO MANY PARENTHESIS GROUPS

Parentheses are nested more than two deep within the outer parentheses of format specification or are improperly paired. The applicable format specification is listed on the \$OUT dataset.

VALUE AND SPECIFICATION TYPES DIFFER

A format specification is encountered which has an edit descriptor that cannot be used with the corresponding input/output list item (variable, array, etc.) due to data type differences. The applicable format specification is listed on the \$OUT dataset.

WRITE AFTER UNCLEARED ERROR

A buffered output operation has been attempted on a unit that was in error during the previous transfer and was not checked for (by the UNIT function).

WRITE CALL OUT OF SEQUENCE

A sequence of output operations is improperly structured.

WRITE ON READ ONLY FILE

An output operation has been attempted on a file protected against such.

WRITE PAST END OF ALLOCATED AREA

An output operation has been attempted which would cause the allocated mass memory subsystem storage area to be exceeded.

xxxxxxxxx WAS CALLED BY yyyy/yyyy AT LOCATION aaaaaaaaaa

This message gives information that aids in locating errors causing job termination. In the first line, *xxxxxxxxx* is name of the (COS) system routine that detected the error. Similar information is printed for each previously executed procedure until the name of the main program appears as *yyyy/yyyy*. *aaaaaaaaaa* identifies the location of the applicable exit from the corresponding procedure or main program.

CREATING NON-FORTRAN PROCEDURES F

This appendix describes and exemplifies conventions for the creation of non-FORTRAN function and subroutine subprograms programmed in CAL, the CRAY-1 Assembly Language. CAL is documented in Cray Research Publication 2240000, "CAL ASSEMBLER VERSION 1 REFERENCE MANUAL".

GENERAL

CRAY-1 Assembly Language (CAL) routines that follow linkage conventions described in this appendix can be loaded and executed with executable programs compiled by CFT, the CRAY-1 FORTRAN compiler. Conventional practices for linking CAL routines with FORTRAN programs are defined in the following areas:

- Argument transmission
- B and T register use
- Error traceback

ARGUMENT TRANSMISSION

Argument transmission is by one of two methods:

- Call by address
- Call by value

CALL BY ADDRESS

In the call-by-address mode of argument transmission, memory addresses of arguments are provided to the called procedure in locations immediately preceding its entry point. The address of the initial actual argument in the FORTRAN subroutine call or function reference is stored at entry point -1 of the called procedure; the next at entry point -2; etc. A maximum of 511 argument addresses may be stored in this manner. The called procedure must allocate the memory required for this purpose. Note that these arguments (and not their addresses) may be modified by the called procedure prior to returning control to the calling FORTRAN program unit.

CALL BY VALUE

The CRAY-1 FORTRAN Compiler creates a call-by-value sequence of code upon encountering references to known library functions. This mode of argument transmission may also be used for entering one non-FORTRAN procedure from another.

In the call-by-value mode, the calling procedure enters all scalar and/or vector arguments into V, VL, and/or S registers as appropriate. Return arguments, if any, are conveyed to the calling procedure via these same registers. In the general case, the first scalar or vector argument (or first result) is passed in S1 or V1, respectively. If a double or complex value is passed, S2 or V2 is used as well. Up to seven registers can be used for passing values. These may be a combination of scalar and vector registers. However, only one of each pair of similarly numbered S and V registers is used (that is, if S1 is used for the first value and the next argument to be passed is a vector, it will be passed in V2 rather than in V1).

By convention, procedures developed for call-by-value argument transmission are assigned names ending with % (e.g., SQRT%).

B AND T REGISTER USE

Contents of the first 56 B and T registers (register addresses 00 through 67₈) must be saved upon entry to a non-FORTRAN procedure and restored before control is returned to the calling procedure. By convention, the last eight B and T registers (register addresses 70₈ through 77₈) are work registers that need not be saved and restored even if used by the called procedure.

ERROR TRACEBACK

Called routines are entered with a return jump. Register B00, therefore, contains the exit address for the called routine. Upon being entered, the called routine should load register B01 with the address of a 3-word error recovery linkage table. Word 1 of this table contains the called routine name (up to 8 left-adjusted ASCII characters). Word 2 contains the entry value contained in B00 (the exit address for the called routine). Word 3 contains the entry value contained in B01 (the linkage table address of the calling routine).

Example:

The following is an example of a FORTRAN call to a CAL routine:

1	7
CALL TEST (A,B)	

The following CAL code corresponds to this call:

LOCATION	RESULT	OPERAND	COMMENT
1	10	20	35
	A1	A	ADDRESS OF FIRST PARAMETER
	W.TEST-1,0	A1	STORE ADDRESS AT TEST-1
	A1	B	ADDRESS OF SECOND PARAMETER
	W.TEST-2,0	A1	STORE ADDRESS AT TEST-2
	R	P.TEST	RETURN JUMP TO TEST

Subroutine TEST could be coded in CAL follows:

LOCATION	RESULT	OPERAND	COMMENT
1	10	20	35
	IDENT	EXAMPLE	
	ENTRY	TEST	
BCOUNT	SET	5	ROUTINE USES B2 through B6
TCOUNT	SET	1	ROUTINE USES REGISTER TO
TABLE	CON	'TEST'L	
BSAVE	BSS	2+BCOUNT	SAVE B0, B1, ..., B6
TSAVE	BSS	TCOUNT	SAVE TO
	BSS	1	2ND ARGUMENT ADDRESS
	BSS	1	1ST ARGUMENT ADDRESS
TEST	A0	BSAVE	
	A1	2+BCOUNT	
	,A0	B0,A1	SAVE B REGISTERS
	S0	TO	
	TSAVE,0	S0	SAVE TO
	A1	TABLE	CURRENT LINKAGE TABLE ADDRESS
	B1	A1	
	:	:	:
	A1	W.TEST-1,0	ADDRESS OF FIRST ARGUMENT
	S1	A1,0	VALUE OF FIRST ARGUMENT
	:	:	:
	A1	W.TEST-2,0	ADDRESS OF SECOND ARGUMENT
	A1,0	S3	RETURN NEW VALUE TO SECOND ARGUMENT
	:	:	:
	A0	BSAVE	
	A2	2+BCOUNT	
	B0,A2	,AO	RESTORE B REGISTERS
	S3	TSAVE,0	
	T0	S3	RESTORE TO
	J	B0	RETURN
	END		

CRAY-1 CENTRAL PROCESSING UNIT

G

This appendix highlights those features of the central processing unit (CPU) of significance to the CRAY-1 FORTRAN programmer in program unit developing and testing. A complete description of the CPU is documented in Cray Research publication 2240004, "CRAY-1 Computer System Reference Manual.

THE MEMORY SECTION

Depending on the model of CRAY-1 Computer System, the memory section of the CPU contains 262,144 or 524,288 or 1,048,576 memory locations for retaining as many 64-bit CRAY-1 computer words. These locations are evenly divided among sixteen independently operating memory banks. Each word is associated with a unique address and can be accessed in serially-addressed blocks at a rate of 80-million per second. Computer words contained in memory can be received from or transmitted to a single section of the CPU at any given moment.

Other features of the memory section include:

- Memory protection, which guards specified areas of memory against inadvertent modification,
- Error detection and correction, which detects and, in most cases, corrects errors encountered in reading from memory locations, and
- Conflict detection and resolution, which enables simultaneous memory bank operations and prevents the loss of information when memory bank access conflicts occur.

THE INSTRUCTION CONTROL SECTION

The instruction control section performs the interrelated CPU functions of program and interrupt control. Program control functions obtain and execute instructions. Interrupt control functions coordinate the orderly processing of unpredictable error and other conditions occurring during CPU operation.

Program control

Instruction control registers supporting program control functions include:

- The program register,
- The instruction buffer registers,
- The instruction issue registers, and
- The real-time clock register.

The 22-bit program (P) register specifies the address of that memory location containing the next instruction required for program execution. A CRAY-1 instruction is either 16 or 32 bits and is contained in one or two of the four 16-bit *parcels* at that address. (A 32-bit instruction can have its first 16 bits in the last parcel at that address and its last 16 bits in the first parcel at the next higher address.) The P-register also indicates which instruction in the word executes.

Each of four instruction buffer registers store 64 16-bit instruction parcels within the instruction control section. If P-register addresses indicate instructions already in the buffer registers, no reference to memory occurs. Otherwise, the memory location addressed has its content and the contents of fifteen adjacently-addressed memory locations read into an instruction buffer register. Instructions to be executed are transferred from an instruction buffer register into instruction issue registers.

The instruction issue registers hold an instruction during its decoding. The first seven bit positions of each instruction identify it among 128 possible CRAY-1 instructions. Instruction execution is complete when the CPU actions called for have been produced.

All sections of the CPU operate on a 12.5 nanosecond time base. The instruction control section permits access to a 64-bit real-time clock register that is incremented every 12.5 nanoseconds.

Interrupt control

The interrupt control functions of the instruction control section provide for immediate interruption of program execution upon the occurrence of certain programming and equipment conditions. The CRAY-1 Operating System (COS) is given instant control of the system by these functions. It is provided with information enabling analysis of the situation, selection of appropriate actions, and restoration of control to the interrupted computer program, if feasible. The mode (M) and flag (F) registers serve inter-related functions in accounting for the type of interrupt and the corresponding action to be taken. An exchange address (XA) register locates the sixteen memory locations that contain information required for interrupt processing and designated to receive the contents of key operating and control registers from throughout the CPU. The procedure through which this is accomplished is called an exchange sequence. The content of these sixteen registers is referred to as an exchange package.

THE COMPUTATION SECTION

The computation section of the CRAY-1 (Figure G-1) consists of operating registers and functional units that perform three types of processing: vector, scalar, and address. Certain registers and functional units are associated with each of these types. For vector processing, there are a set of 64-bit multi-element registers, three functional units dedicated solely to vector operations, and three floating-point functional units shared by vector and scalar operations. For scalar processing, there are two levels of 64-bit scalar registers, four functional units dedicated solely to scalar processing, and the three floating-point units shared with vector operations. For address processing, there are two levels of 24-bit registers and two integer arithmetic functional units.

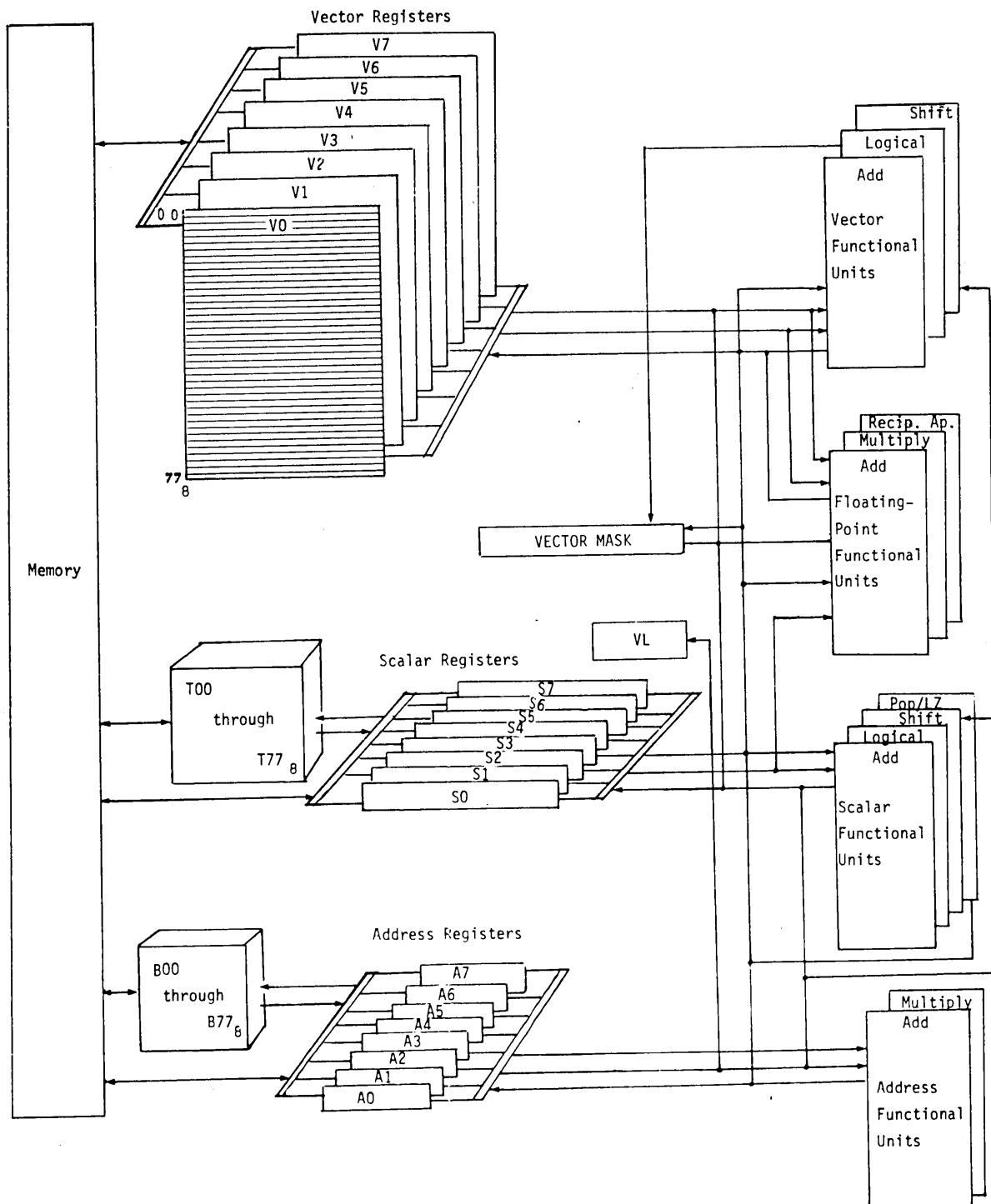


Figure G-1. The computation section

Registers

The five types of operating registers in the computation section are:

- Address (A) registers,
- Intermediate address (B) registers,
- Scalar (S) registers,
- Intermediate scalar (T) registers, and
- Vector (V) registers.

Supporting registers include the vector length (VL) and vector mask (VM) registers.

A registers - The eight 24-bit A registers are primarily used as address and index registers for memory references. They are individually designated A0, A1, ..., A7. Data may be directly transferred between A registers and memory.

B registers - The sixty-four 24-bit B registers provide rapid-access temporary storage for the A registers. They are individually designated B0, B1, ..., B77. Data is transferred between B registers and memory via high-speed block transfers.

S registers - The eight 64-bit S registers are the principal scalar registers for the CPU. They are individually designated S0, S1, ..., S7. These registers serve as source and destination registers during scalar arithmetic and logical operations. They may also furnish one operand during vector operations. Data flows between these registers and the A, T, V, and VM registers and between the S registers and memory.

T registers - The sixty-four 64-bit T registers provide rapid access temporary storage for the S registers. They are individually designated by the symbols T0, T1, ..., T77. Data is transferred between T registers and memory via high-speed block transfers.

V registers - The eight 64-element V registers are the operating registers for vector computations. Each element is 64 bits. The V registers are individually designated V0, V1, ..., V7. These registers serve as a source and destination registers for vector arithmetic and logical operations. Data flows between V registers and S registers, and between V registers and memory.

The VL register - The 7-bit VL register holds a value in the range of 1-64. Vector computations are performed on vectors of the length specified by the content of VL.

The VM register - The 64-bit VM register contains a vector mask to control register selection during the execution of a vector merge instruction. Bits of the VM register can also be set during the execution of a vector mask instruction. Each bit of the VM register corresponds to a vector element in a selected V register.

Comparison of scalar and vector processing

Scalar instructions apply a function to one or two operands contained in registers and enter the result into a register. An example would be adding two integers in S1 and S2, then entering their sum into S3. Vector instructions apply a function to one or two sets of operands called vectors. The addend vector is placed into one V register and the augend vector into another V register. Execution of a single vector addition instruction would then produce the desired vector sum. In most applications, vector processing provides significantly higher result rates than are possible using scalar processing. The CRAY-1 FORTRAN compiler includes vector instructions at appropriate points in the programs it compiles.

Functional units

Arithmetic, logical, and related operations are performed by twelve functional units, any number of which may be in operation at once.

A functional unit receives operands from operating registers, operates on them and delivers the result to a register. Functional units generally operate in a three-address mode where two source and one destination operating register addresses are specified.

Each functional unit performs its operations in a fixed amount of time. The time required between delivery of the operands to a unit and its completion of operations is termed "functional unit time" and is measured in 12.5 nanosecond clock periods.

Functional units are fully segmented such that a new set of operands for the next computation may enter a functional unit each clock period even though the functional unit time may be more than one clock period.

The twelve functional units are described in four groups: address, scalar, vector, and floating-point. The address, scalar, and vector groups relate to the A, S, and V registers, respectively. The floating-point group supports both scalar and vector operations, and accepts operands from and delivers results to S and V registers.

Address functional units - Each address functional unit performs 24-bit two's complement integer arithmetic on operands obtained from A registers and delivers a result to an A register.

Address add unit

The address add unit performs a 24-bit integer addition or subtraction in two clock periods.

Address multiply unit

The address multiply unit forms a 24-bit integer product from two 24-bit operands in six clock periods. Rounding is not performed.

Scalar functional units - Each of four scalar functional units operates on 64-bit operands obtained from S registers and, in most cases, delivers a 64-bit result to an S register. Arithmetic is two's complement integer.

Three floating-point functional units are associated with both scalar and vector operations. When a floating-point functional unit is used for a scalar operation, the general description of scalar functional units applies.

Scalar add unit

The scalar add unit performs a 64-bit integer addition or subtraction in three clock periods.

Scalar shift unit

The scalar shift unit shifts the 64-bit content of an S register or the 128-bit content of two concatenated S registers. A shift count is obtained from an A register or directly from the instruction. A single-register shift is executed in two clock periods; a double-register shift in three.

Scalar logical unit

The scalar logical unit performs bit-by-bit manipulation of 64-bit quantities obtained from S registers. This operation requires one clock period.

Population/leading zero count unit

This functional unit counts the number of one bits in an operand in four clock periods. It also counts the number of leading zero bits in an operand in three clock periods. In either case, a 64-bit operand is obtained from an S register and a 7-bit result is delivered to an A register.

Vector functional units - Vector functional units operate on operands obtained from one or two V registers, or from a V register and S register. Results are generally delivered to a V register.

One or a pair of register elements is transmitted to a functional unit each clock period. The corresponding result is available n clock periods later, where n is constant for a given functional unit. The content of the vector length (VL) register determines the number of register element pairs to be processed by a functional unit.

The three vector functional units are exclusively associated with vector operations. Three floating-point functional units are associated with both vector and scalar operations. When a floating-point unit is used for a vector operation, the general description of vector functional units applies.

Vector add unit

The vector add unit performs a 64-bit integer addition or subtraction for a vector operation and delivers the result to a V register. The functional unit time for the vector add unit is three clock periods per element-pair operation.

Vector shift unit

The vector shift unit shifts the 64-bit content of a V register element or the 128-bit content of two concatenated V register elements. The shift count is obtained from an A register. Shifts are end-off with zero fill. The functional unit time is four clock periods.

Vector logical unit

The vector logical unit performs bit-by-bit manipulation of 64-bit quantities contained in V registers. The functional unit time is two clock periods.

Floating-point functional units - Three floating-point functional units perform floating-point arithmetic for both scalar and vector operations. When executing a scalar instruction, an operand pair is obtained from S registers and the result is delivered to an S register. Operands for most vector instructions are obtained from two V registers or from a V register and an S register; normalized results are delivered to a V register.

Floating-point add unit

The floating-point add unit adds or subtracts two 64-bit floating-point operands in six clock periods.

Floating-point multiply unit

The floating-point multiply unit performs full- and half-precision multiplication of two 64-bit floating-point operands and computes two minus a floating-point product for reciprocal operations. The floating-point multiply unit also handles integer multiplication as a special case. The functional unit time is seven clock periods.

Reciprocal approximation unit

The reciprocal approximation unit approximates the inverse of a 64-bit floating-point operand. This operation requires fourteen clock periods.

Functional unit relationships

Table G-1 summarizes the relationships between functional units, the three computation modes, the three types of arithmetic, and the operations performed in the computation section of the CPU. Each of the twelve functional units is named in the body of this table.

Table G-1. Functional unit relationships

COMPUTATION MODE	VECTOR		SCALAR	ADDRESS
ARITHMETIC TYPE	INTEGER (64-BIT)	FLOATING POINT (64-BIT)	INTEGER (64-BIT)	INTEGER (24-BIT)
OPERATION				
ADDITION	VECTOR ADD	FLOATING-POINT ADD	SCALAR ADD	ADDRESS ADD
MULTIPLICATION	FLOATING-POINT MULTIPLY			ADDRESS MULTIPLY
RECIPROCAL APPROXIMATION	---	FLOATING-POINT RECIPROCAL	---	---
LOGICAL	VECTOR LOGICAL	---	SCALAR LOGICAL	---
SHIFT	VECTOR SHIFT	---	SCALAR SHIFT	---
POPULATION AND LEADING ZERO COUNT	---	---	SCALAR POPULATION AND LEADING ZERO COUNT	---

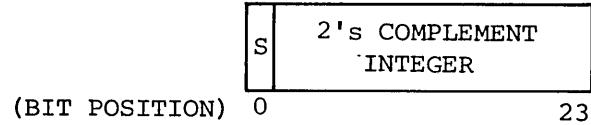
Data representations

All numeric data in the CPU are represented in one of three forms:

- 24-bit integer
- 64-bit integer
- 64-bit floating point

These are illustrated in figure G-2. The n bit positions within each numeric data word are denoted 0 through $n-1$ from left to right .

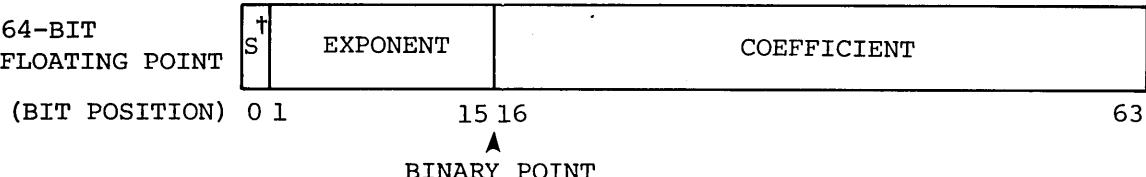
24-BIT
INTEGER



64-BIT
INTEGER



64-BIT
FLOATING POINT



S = Sign bit

S[†] = Sign bit of coefficient

Figure G-2. Numeric data representation

24-bit integer data - A 24-bit integer datum is represented as a sign bit followed by 23 bits. Positive integer values up to 8,388,607 (that is, $2^{23}-1$) may be represented and are indicated by a zero in the leftmost (sign) bit position. Negative integer values with magnitudes as large as 8,388,608 (that is, 2^{23}) are indicated by a one in the leftmost (sign) bit position and are represented in two's complement form. Zero is represented by zeros in all 24 bit positions.

Examples:

<u>Decimal value</u>	<u>24-bit (octal) representation</u>
14	0 0 0 0 0 1 6
-14	7 7 7 7 7 7 6 2
1,624,718	0 6 1 4 5 2 1 6
-1,624,718	7 1 6 3 2 5 6 2
0	0 0 0 0 0 0 0 0
8,388,607	3 7 7 7 7 7 7 7
-8,388,607	4 0 0 0 0 0 0 1
-8,388,608	4 0 0 0 0 0 0 0

64-bit integer data - A 64-bit integer datum is represented as a sign bit followed by 63 bits. Positive integer values up to $2^{63}-1$ may be represented and are indicated by a zero in the leftmost (sign) bit position. Negative integer values with magnitudes as large as 2^{63} are indicated by a one in the leftmost (sign) bit position and are represented in two's complement form. Zero is represented by zeros in all 64-bit positions.

Examples:

<u>Decimal value</u>	<u>64-bit (octal) representation</u>
14	0 1 6
-14	1 7 6 2
7,291,343,192,400,000,615	0 6 2 4 6 0 0 3 4 0 0 2 2 6 3 5 5 7 3 1 4 7
-7,291,343,192,400,000,615	1 1 5 3 1 7 7 4 3 7 7 5 5 1 4 2 2 0 4 6 3 1
0	0 0
9,223,372,036,854,775,807	0 7
-9,223,372,036,854,775,807	1 0 1
-9,223,372,036,854,775,808	1 0

64-bit floating-point data - A 64-bit floating-point datum is represented by a sign bit followed by 15 exponent bits followed by 48 coefficient bits. As illustrated in figure G-2, the implied binary point for the coefficient precedes its first bit (position 16) such that the coefficient is always less than one.

The sign of the coefficient is in the leftmost bit of the word (that is, position zero), is zero to indicate a positive coefficient and is one to indicate a negative coefficient. The value of the coefficient is contained in the rightmost 48 bit positions of the word (positions 16 through 63). Unlike integer value representations which are expressed in two's complement form when negative, a floating-point coefficient is the same for both positive and negative representations of the same value; only its sign bit changes. This is spoken of as signed magnitude representation. The largest fractional value the coefficient may accurately contain is $1-2^{-48}$.

The exponent portion of a floating-point data representation occupies bit positions 1 through 15 of the word. It contains a value representing that positive or negative integral power of two by which the coefficient is to be multiplied. A biased exponent value of 40000_8 represents 2^0 (or a multiplier of 1). This value may be increased to a maximum of 57777_8 which represents 2^{8191} .

Correspondingly, it can be decreased to a minimum of 20000_8 which represents 2^{-8192} . Exponent values of 60000_8 or higher, or of 17777_8 or lower, are invalid and represent out-of-range situations. An exponent values of zero is an exception; it is invalid except when used with a coefficient of zero to represent the quantity zero. Table G-2 summarizes the interpretations of significant exponent value ranges.

Table G-2. Significant exponent value ranges

Exponent	Interpretation
0	Valid only for the value zero
$1 - 17777_8$	Invalid
$20000_8 - 37777_8$	$2^{-8192} - 2^{-1}$
40000_8	2^0
$40001_8 - 57777_8$	$2^1 - 2^{8191}$
$60000_8 - 77777_8$	Invalid

The appearance of leading zeros in the coefficient portion of a floating-point value characterizes that value as being *unnormalized*. The *normalized* form has these leading zeros eliminated and the exponent adjusted accordingly. The CRAY-1 FORTRAN compiler assumes only normalized floating-point values.

Examples:

Decimal value	Floating-point representation(octal)			Notes
	Sign (0)	Exponent (1-15)	Coefficient (16-63)	
.5	0	40000	4000000000000000	Normalized
-.5	1	40057	0000000000000001	Unnormalized
.75	0	40033	0000000006000000	Unnormalized
-1.75	1	40001	7000000000000000	Normalized
1.00	0	40001	4000000000000000	Normalized
-2.00	1	40002	4000000000000000	Normalized
281,474,976,710,655	0	40060	7777777777777777	Normalized
-140,737,488,355,327.5	1	40057	7777777777777777	Normalized
562,949,953,421,310	0	40061	7777777777777777	Normalized
-.25	1	37777	4000000000000000	Normalized
.000000000001	0	37731	4313631402267520	Normalized
-1,000,000,000,000	1	40050	7215224504000000	Normalized
$(1-2^{-48})$	0	40000	7777777777777777	Normalized
0	0	00000	0000000000000000	Zero

Floating-point operations deal chiefly with normalized values to provide for the most accurate representation possible in the 48 coefficient bit positions. (The CRAY-1 FORTRAN programmer requires an understanding of unnormalized floating-point values only for certain error analysis situations.)

The only representation of zero permitted is a word containing zeros in all 64 bit positions.

Arithmetic operations

The three types of arithmetic operation in the CRAY-1 CPU are:

- 24-bit integer,
- 64-bit integer, and
- 64-bit floating point.

Integer arithmetic operations - All integer arithmetic, whether 24-bit or 64-bit, is performed in two's complement. Address add and address multiply functional units perform 24-bit integer arithmetic. Scalar add and vector add functional units perform 64-bit integer arithmetic.

Multiplication of two integer operands is accomplished by the floating-point multiply unit which recognizes zero exponents in both operands as a special case. The upper 48-bits of the result are returned as an un-normalized value preceded by fifteen zeros. Division of integers requires that they first be converted to floating-point format, then processed by the floating-point functional units.

Floating-point arithmetic operations - Floating-point numbers are represented as a signed binary coefficient and an exponent or power of two.

In terms of decimal values, the floating-point format of the CRAY-1 allows the expression of numbers accurate to about 15 decimal digits and in the approximate decimal range of 10^{-2500} through 10^{+2500} .

Normalized floating point

A non-zero floating-point number is normalized if the most significant bit of the coefficient (position 16) is one. This condition implies that the coefficient has been shifted to the left as far as required to eliminate all leading zeros.

Floating-point range errors

Overflow of the floating-point range is indicated by an exponent value of 60000_8 or greater; underflow by an exponent value of 17777_8 or less. Detection of the overflow condition can cause an interrupt depending upon the mode (M) register content.

Floating-point addition

A floating-point add range error condition is generated when the larger incoming exponent is greater than or equal to 60000_8 . The floating-point error flag is set and an exponent of 60000_8 is sent to the result register along with the computed coefficient, as in the following example:

$$\begin{array}{r} 60000.4 \text{ (Range error)} \\ + 57777.4 \\ \hline 60000.6 \text{ (Range error)} \end{array}$$

Floating-point multiplication

In the floating-point multiply unit, if the exponent of either operand is greater than or equal to 60000_8 or if the sum of the two exponents is greater than or equal to 60000_8 , the floating-point error flag is set and an exponent of 60000_8 is sent to the result register along with the computed coefficient.

An underflow condition is detected when the result exponent is less than or equal to 17777_8 and causes a zero exponent and coefficient to be returned to the result register. Underflow is also generated when either, but not both, operand exponent is zero. Both exponents equal to zero is treated as an integer multiply.

The floating-point multiply unit approximates a normalized 48-bit fractional product. This results in a product variation between 1.16×10^{-16} and 6.66×10^{-16} .

Floating-point reciprocal approximation

For the floating-point reciprocal approximation unit, an incoming operand with an exponent less than or equal to 20001_8 or greater than or equal to 60000_8 causes a floating-point range error. The error flag is set and an exponent of 60000_8 is associated with the computed coefficient.

Floating-point division is performed by the reciprocal approximation method.

The division algorithm that computes S1/S2 to full precision requires four operations:

1. $S_3 = 1/S_2$ Reciprocal approximation
2. $S_4 = (2 - S_3 \cdot S_2)$ Reciprocal iteration
3. $S_5 = S_1 \cdot S_3$ Numerator approximation
4. $S_6 = S_4 \cdot S_5$ Half-precision quotient correction factor

This approximation is based on Newton's method and, at step 1, is correct to 30 bits. The additional Newton iteration at step 2 increase this accuracy to 47 bits and is applied as a correction factor at step 4 in a full-precision multiply operation.

Where 31 bits of accuracy is sufficient, the reciprocal approximation instruction may be used with half-precision multiplication to produce a half-precision quotient.

The 18 lowest-order bits of half-precision results are returned as zeros. Rounding is applied to the 30-bit result.

Logical operations

The scalar and vector logical units perform bit-by-bit manipulation of 64-bit quantities. Operations provide for forming logical products, logical differences, and logical sums.

A logical product is the AND function:

operand one	1 0 1 0
operand two	<u>1 1 0 0</u>
result	1 0 0 0

A logical difference is the exclusive OR function:

operand one	1 0 1 0
operand two	<u>1 1 0 0</u>
result	0 1 1 0

A logical sum is the inclusive OR function:

operand one	1 0 1 0
operand two	<u>1 1 0 0</u>
result	1 1 1 0

THE INPUT/OUTPUT SECTION

The input/output section of the CPU contains twelve input and twelve output channels that are divided into four groups of six channels each.

Each input channel consists of a 16-bit data channel, a 64-bit assembly register, a current address register, and a limit address register. An input channel can cause a CPU interrupt condition when the current address equals the limit address or when the input device sends a disconnect signal.

Each output channel consists of a 16-bit data channel, a 64-bit disassembly register, a current address register, and a limit address register. An output channel can cause a CPU interrupt condition when the current address equals the limit address. A disconnect is sent to the output device after the last word of a transmission has been sent and acknowledged as received.

It is possible to have an I/O memory request every clock period.

Maximum request rates in clock pulses (CPs) are:

- 1 channel 8 CPs (10 million words per second)
- 1 channel group 4 CPs (20 million words per second)
- All channel groups 1 CP (80 million words per second)

RANDOM INPUT/OUTPUT OPERATIONS

H

FORTRAN input/output statements provide for the transfer of information directly to and from input/output devices. CRAY-1 FORTRAN input/output statement specification conforms with this provision but is also able to exploit the presence of these data in memory. This enables random input/output operations to be performed whereby records can be accessed directly instead of serially. This appendix describes methods for performing random input/output operations.

The CRAY-1 FORTRAN Compiler (CFT) associates unit identifiers specified in input/output statements with datasets assigned to a job. This association permits input/output statements to perform operations on datasets as though they were actual input/output devices. Thus, the forms of these statements remain standard. Within a dataset, each of a sequence of files contains a sequence of records. Input/output statements normally operate on these files and records in their order of appearance within the dataset (i.e., perform sequential input/output operations).

CFT provides for non-sequential or random input/output operations on datasets through the use of the utility procedures GETPOS and SETPOS (see Appendix C). Additionally, any dataset to be processed in this manner must be so identified on the ASSIGN control statement for that dataset (as described in Cray Research publication 2240011, "CRAY-OS Version 1.0, External Reference Specification").

A dataset can only be created using sequential output operations. It may be accessed and modified through random input/output operations but only if the lengths and starting locations of each record in the dataset are determined initially and kept current.

Using random input/output operations, any record in the datafile can be replaced by a record of equal length without rewriting the entire datafile.

The following is an example of random input/output programming.

Example:

In the main program below, up to one hundred records containing from zero to ten words each are written into a dataset associated with input/output unit 1. A final record of up to 201 words is added and contains length and location information for each preceding record plus a count of their number. The dataset is rewound. At a later point in the program, a subroutine is called which causes all records to be read in reverse order and all but the last stored into ten word vectors of a one hundred vector array. Information in the last record directs this process. The subroutine then returns control to the main program.

The following are assumed preset:

- NRECS to the number of records to be processed ($1 \leq NRECS \leq 100$)
- RLENGTH(i) to the number of words in the i th record written ($0 \leq RLENGTH(i) \leq 10$)
- RECORD(j) to the j th word to be written in each non-empty record ($1 \leq j \leq RLENGTH(i)$)

```

PROGRAM RANDOMIO
INTEGER RLENGTH(100),ADDRESS(100),RECORD(10),NRECS,LRA,RESULT
COMMON RESULT(100,10)
:
DO 20 I=1,NRECS
  ADDRESS(I)=GETPOS(1)
20 WRITE(1) (RECORD(J),J=1,RLENGTH(I))
  LRA=GETPOS(1)
  WRITE(1)NRECS,(RLENGTH(I),ADDRESS(I),I=1,NRECS)
  REWIND(1)
:
CALL READIN(LRA,1)
:
END

```

```

SUBROUTINE READIN(LRA,UNIT)
INTEGER RCOUNT,RADDRESS(100),LNGTH(100),RESULT,UNIT,LRA
COMMON RESULT(100,10)
:
CALL SETPOS(UNIT,LRA)
READ(UNIT)RCOUNT,(LNGTH(I),RADDRESS(I),I=1,RCOUNT)
DO 10 I=1,RCOUNT
  J=RCOUNT-I+1
  CALL SETPOS(UNIT,RADDRESS(J))
10 READ(UNIT)(RESULT(I,K),K=1,LNGTH(J))
:
END

```

INDEX

Actual argument	3-131	Buffered I/O operations	3-111
Actual array declarator	3-29	CAL procedures	F-1
Addition operator	3-36, 3-39	CALL statement	3-126
Adjustable array declarator	3-28	CFT (see Compiler)	
Alphanumeric characters	3-3	CFT control statement	6-4
ANSI FORTRAN		Character, alphanumeric	3-3
character set	A-1	Character set	
conformance to standard	3-1	ANSI	A-1
Argument		ASCII	A-1
actual	3-53, 67, 131	auxiliary	3-3
association of	3-131	FORTRAN	3-3
dummy	3-130	Characters, control	3-90
storage	3-137	CODE directive	5-7
transmission	F-2	Coding form	3-7
Arithmetic assignment statement	3-70	Collation sequence	3-3
Arithmetic constants	3-20	Columns on coding form	3-7
Arithmetic expression	3-35	Comment lines	3-9, 18
Arithmetic IF statement	3-75	Common block	3-59, 60
Arithmetic operations, CRAY-1	G-9	blank	3-61
Array		COMMON association	3-60
actual	3-31	EQUIVALENCE association	3-59
adjustable	3-34	list	5-2
data initialization	3-68	named	3-61
data type	3-29	size	3-61
declarator	3-28, 58	storage sequence	3-61, 137
defined entities	3-139	storage unit association	3-61
definition	3-28	symbolic name	3-141
dimensionality	3-29	COMMON statement	3-60, 62
dummy	3-31	Compiler	
dummy arguments	3-132	input to	5-1
element name	3-19	directive lines	3-9; 5-6
element order	3-30	directives	5-5
name	3-19	output from	5-1
properties	3-29	Complex constants	3-24
size	3-29	Complex data	3-14
storage	3-30, 137	COMPLEX FUNCTION statement	3-123
symbolic name	3-143	COMPLEX statement	3-64
undefined entities	3-138	Computation section	2-3
ASCII character set	A-1	arithmetic operations	G-13
ASSIGN control statement	6-3	data representation	G-9
ASSIGN TO statement	3-71	functional units	G-6
Assigned GO TO statement	3-74	logical operations	G-15
Asterisks in output	3-101	registers	G-5
Auxiliary character set	3-3	scalar and vector processing	G-6
BACKSPACE statement	3-91	Computed GO TO statement	3-73
Blank common	3-61	Constants	
Blank characters in constants	3-20	array declarator	3-28
Blanks in input	3-101	description	3-4, 20
BLOCK DATA statement	3-135	Hollerith	3-44
Block data subprograms	3-10, 135	increment integer	4-2
Boolean constants	3-25	name	3-29
Boolean data	3-14	symbolic names	3-143
Brackets, square	3-2	Continuation lines	3-9
BUFFER IN statement	3-112	CONTINUE statement	3-81
BUFFER OUT statement	3-113	Control characters	3-90

Conversion		Disk storage unit	2-2
complex	3-44	Dividing by zero	3-53
double precision	3-43	Division operator	3-36
integer	3-43	DO-list, implied	3-68, 87
maximum value	3-43	DO-loops	
real	3-43	active	3-78
type	3-43	dependencies	4-3
zero value	3-43	description	3-77
COS (see operating system)		DO-variable	3-79
CPU		execution	3-78
computation section	2-3;G-3	inactive	3-78
general description	2-2	iteration count	3-79
input/output section	2-3;G-16	loop control processing	3-79
instruction section	2-3;G-2	range	3-78
memory section	2-3;G-2	terminal statement	3-79
CRAY-1 Computer System		transfer into	3-81
configuration	2-1	transfer of control	3-78
general description	G-1	vectorizable	4-1, 9
overview	2-1	DO statements	3-77
Data		Double-precision constants	3-23
arithmetic assignment	3-70	Double-precision data	3-14
ASSIGN statement	3-71	DOUBLE PRECISION statement	3-64
association	3-15	DOUBLE PRECISION FUNCTION	
deck	6-1	statement	3-123
declaration	3-58, 66	DOUBLE statement	3-64
definition	3-15	Dummy arguments	3-130, 132
form	3-10	Dummy array declarator	3-29
formatted assignment	3-115		
formatted transfer	3-90	Edit descriptor	
identifiers	3-19	A	3-109
initialization	3-68	apostrophe	3-97
logical assignment	3-71	asterisk	3-97
name	3-10	colon	3-99
reference	3-15	D	3-103
representation	3-14	description	3-93
specification	3-14, 19	E	3-104
storage	3-15	F	3-103
transfer direction	3-88	G	3-106
type	3-10, 41	H	3-98
unformatted transfer	3-89	I	3-101
DATA statement	3-68, 69	L	3-108
Dataset	3-84	O	3-107
DD-19 drives	2-2	P	3-99
Decimal point in input	3-101	quotation mark	3-97
Decimal range of constants	3-21	R	3-110
Declarator		slash	3-99
actual array	3-31	T	3-98
adjustable array	3-34	X	3-98
array	3-28, 58	Z	3-108
dimension	3-28		
dummy array	3-31	Editing of data	
DECODE statement	3-115	complex	3-103
Dimension declarator	3-28	description	3-83
Dimension limit	3-14	double-precision	3-103
DIMENSION statement	3-58	integer	3-101
Direct logical IF statement	3-76	numeric	3-101
Disjuncts, logical	3-49	positional	3-98
		real	3-103

EJECT directive	5-6	Functions	
elements of arrays	3-14	description	3-14
Ellipses in this publication	3-2	dummy arguments	3-133
ENCODE statement	3-115	external	3-123
END statement	3-82	intrinsic	3-119
Endfile record	3-83	name	3-19
ENDFILE statement	3-91	order of evaluation	3-53
End-of-file record	3-83	reference execution	3-53
ENTRY statement	3-128	referencing	3-118
EQUIVALENCE statement	3-58,62	subprograms	3-12
Error messages		FUNCTION statement	3-123
CFT fatal error	5-5;D-3	Functional units in CPU	G-6
CFT warning	5-4;D-2		
Operating system	E-1	Global entities	3-140
Program execution	6-7	GO TO statements	
Error traceback	F-3	assigned	3-74
Execution		computed	3-73
end	3-81	unconditional	3-72
main program	3-10	Hollerith constant	3-26,44
sequence	3-17	Hollerith data	3-14
Exponentiation	3-36	Hollerith expressions	3-44
Expressions		Host processor	2-1
arithmetic	3-35	Identity operator	3-36,39
arithmetic relational	3-45,54	IF statement	
Boolean	3-51,55	direct logical	3-76
complex	3-41	indirect logical	3-77
definition	3-35	three-branch	3-75
double-precision	3-41	two-branch	3-75
evaluation	3-52	IMPLICIT statement	3-65
equivalent	3-54	Implicit type	3-19
Hollerith	3-44,55	Implied DO-list	3-87
Hollerith relational	3-46	Initial line	3-9
integer	3-41	Input/output	
interpretation	3-41	buffered	3-111
logical	3-47,55	lists	3-86
real	3-41	operations	3-83,111
relational	3-45,55	random	3-84;H-1
subscript	3-31	section of CPU	2-3
type	3-41	sequential	3-84
External functions	3-13,123	statement execution	3-88
External procedures	3-13	Instruction	
EXTERNAL statement	3-67	in CPU	2-3
Factors	3-38	interrupt control	G-3
Fatal error messages	5-5	program control	G-2
Field width	3-97	Integer constants	3-21
Files	3-84	Integer data	3-14
Floating point values	3-23;G-9	Integer division	3-54
Format identifier	3-92	INTEGER FUNCTION statement	3-123
Format specification	3-92	INTEGER statement	3-65
FORMAT statement	3-92	Interpretation rules	3-52
Formats, input/output	3-85	Interrupt control	G-3
Formatted data assignment	3-115	Intrinsic functions	
Formatted record	3-83	CRAY-1 FORTRAN	B-1
FORTRAN call to CAL routine	F-1	description	3-12,119
FORTRAN character set	3-3	examples	3-120
FORTRAN language	3-1	EXTERNAL statement	3-67
FORTRAN programming	4-1	referencing	3-119
		restrictions	3-120
		symbolic name of	3-144

Invariant values	3-20	
Italics in this publication	3-2	
IVDEP directive	5-8	
Job control deck	6-1	
JOB control statement	6-2	
Job control statement file	6-6	
Job control statements	6-2	
Job dataset	6-1	
Keywords	3-6	
Labels, statement	3-5	
LDR control statement	6-6	
LENGTH function	3-114	
Less than operator	3-45	
Lines	3-9, 16	
List	3-6	
List control information	3-86	
LIST directive	5-7	
List items	3-6	
Lists, input/output	3-86	
Local entities	3-141	
Logical assignment statement	3-71	
Logical constants	3-25	
Logical data	3-15	
Logical expression	3-48	
LOGICAL FUNCTION statement	3-123	
LOGICAL statement	3-64	
Main program	3-10, 116	
Mass storage subsystem	2-2	
MCU	2-1	
Memory	G-2	
Memory operations	4-9	
Multiplication operator	3-36	
Named common	3-71	
Names		
array element	3-31	
array use	3-34	
main program	3-140	
symbolic	3-5	
Negation operator	3-36, 39	
NOCODE directive	5-7	
NOLIST directive	5-7	
Notation in FORTRAN	3-3	
NOVECTOR directive	5-7	
Operating system		
error messages	E-1	
general description	6-1	
Operators		
addition	3-36, 39	
arithmetic	3-36	
consecutive	3-39	
description	3-6	
division	3-36	
equal to	3-45	
exponentiation	3-36	
greater than	3-45	
greater than or equal to	3-45	
identity	3-39	
less than	3-45	
less than or equal to	3-45	
logical conjunction	3-47	
logical equivalence	3-47	
logical exclusive disjunction	3-47	
logical inclusive disjunction	3-47	
logical negation	3-47	
logical non-equivalence	3-47	
multiplication	3-36	
negation	3-36, 39	
not equal to	3-45	
precedence among types	3-51	
precedence of arithmetic	3-36	
relational	3-45	
subtraction	3-36, 39	
Operands		
arithmetic	3-37	
logical	3-48	
relational	3-45	
Output		
common block name list	5-2	
fatal error messages	5-5	
source statement listing	5-2	
statement label list	5-4	
symbolic table	5-2	
warning messages	5-4	
PARAMETER statement	3-20, 66	
Parentheses	3-54	
PAUSE statement	3-82	
Peripherals	2-1	
Position, record on file	3-84	
Positional editing	3-98	
Positioning by format control	3-96	
Powers of negative values	3-53	
Powers of zero	3-53	
Precision of real constants	3-23, 24	
Primaries	3-37, 48, 52	
PRINT statement	3-85	
Printer control	3-90	
Printing	3-90	

Procedure	3-12	Subtraction operator	3-36,39
external	3-67	SWITCH control statement	6-6
utility	3-67	Symbol table	5-2
Procedure subprograms	3-12	Symbolic names	3-5,19,140
Program control	3-72	Syntactic items	3-4
Program, executable	3-10	Syntax of statements	3-2
Program specification	3-1		
PROGRAM statement	3-117	Temporary variables, use of	4-2
Program unit	3-10	Terminal lines	3-9
PUNCH statement	3-85	Terms	3-38
Quotients, integer	3-43	Terms, logical	3-48
Random I/O operations	H-1	Type, change of	3-65
READ statement	3-85	Type statements	3-19,64
Real constants	3-21		
Real data	3-14	Unconditional GO TO statement	3-72
Real exponent	3-23	Unformatted record	3-83
REAL FUNCTION statement	3-123	UNIT function	3-113
REAL statement	3-64	Unit identifier	3-85
Record	3-83	Units, input/output	3-85
Registers, CPU	G-5	Unsigned constant	3-21
Relocatable binary program	5-2	Utility procedures	3-12,135;C-1
Relocatable loader	6-6		
RETURN statement	3-129	Value	
REWIND statement	3-91	arithmetic expression	3-54
Rows on coding form	3-7	arithmetic relational	
Scale factor	3-99,104	expression	3-45
Sequence	3-3	Boolean expression	3-51
Source deck	6-1	false	3-45,46
Specification subprogram	3-10	Hollerith relational	
Statement		expression	3-46
classification of	3-56	logical disjunct	3-49
data specification	3-56	logical expression	3-49
description	3-7	logical factor	3-49
executable	3-7,56	logical term	3-49
functions	3-12,120	true	3-45,46
functions, names of	3-144	Variables	
label	3-5	defined entities	3-138
label list	5-4	description	3-14
list output	5-2	dummy arguments	3-132
nonexecutable	3-7,57	initialization of data	3-68
order of	3-16	name	3-19
size	3-7	storage sequences	3-137
syntax	3-2	symbolic names	3-143
STOP statement	3-81	undefined entities	3-139
Storage sequences	3-15,137	Vector array reference	4-2
Storage units	3-15	VECTOR directive	5-7
Subprogram	3-10	Vector operations, programming	4-1
SUBROUTINE statement	3-126	Vectorization of DO-loops	4-2,9
Subroutine subprograms	3-12	Vertical spacing on print	
Subroutines	3-13,126	output	3-90
Subscript values	3-31	Warning messages	5-4
		WRITE statement	3-85



TECHNICAL COMMUNICATIONS

7850 Metro Parkway, Suite 213, Minneapolis, MN 55420 • (612) 854-7472

PUBLICATION CHANGE NOTICE

April 15, 1978

TITLE: CRAY-1 FORTRAN (CFT) Reference Manual

PUBLICATION NO. 2240009 REV. C

Revision C obsoletes all previous printings of this publication. This printing brings the manual into full agreement with the April 1978 release of the CRAY-1 FORTRAN Compiler (CFT) Version 1.01.

Comment Sheet

Publication Number: 2240009 C

Title: CRAY-1 FORTRAN (CFT) Reference Manual

Please feel free to share with us your comments, criticisms, or compliments regarding this publication. We value your feedback. Thank you.

Comments:

Mail to: Publications
CRAY RESEARCH, INC.
7850 Metro Parkway
Suite 213
Minneapolis, MN 55420



CFT CONTROL STATEMENT FORMAT

The form of the CFT control statement is:

CFT (I=idn,L=lbn,B=bdn,C=cdn,ON=string,OFF=string)

I, L, B, C, ON, and OFF may appear in any order or may be omitted to cause the default options described below to apply. If all are omitted, a period may be used in lieu of empty parentheses.

idn specifies a source input dataset; default is \$IN.

lbn specifies a listable output dataset; default is \$OUT.

bdn specifies a binary output dataset; default is \$BLD.

cdn specifies a listable output dataset in a form suitable for input to the CRAY-1 Assembly Language assembler, CAL; default is no dataset.

string specifies a string of letters each of which enables or disables a particular compiler option when specified ON or OFF, respectively. The default is described with each option in the chart below.

Letter in string	ON (enables option)	OFF (supresses option)	Compilation option († indicates default)
A		†	Aborts job after compilation if any of the program units contains an error.
B	†		List beginning sequence number of each block of vectorizable code (G implies B).
C	†		List common block names and lengths after each program unit.
E	†		Enable recognition of compiler directive lines.
G		†	List code generated for each program unit.
I		†	List CFT-generated statement labels in symbol table.
L	†		Enable recognition of listable output control directives.
N		†	List null statement labels in the symbol table.
Q	†		Abort CFT when 100 errors have been encountered.
S	†		List FORTRAN statement.
T	†		List symbol table for each program unit.
V	†		Vectorize inner DO loops (for timing analyses).
W		†	List warning messages.
X		†	List cross-references within each program unit's symbol table (X overrides T).

Refer to section 6 for additional information on COS and the CFT control statement. Compiler directives are described in section 5.



HEADQUARTERS • 7850 Metro Parkway, Suite 213, Minneapolis, MN 55420 • (612) 854-7472
DEVELOPMENT LABORATORY • P.O. Box 169, Chippewa Falls, WI 54729 • (715) 723-0266