TEXAS INSTRUMENTS

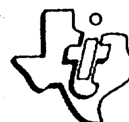
Improving Man's Effectiveness Through Electronics

Model 990 Computer FORTRAN

Programmer's Reference Manual

MANUAL NO. 946260-9701
ORIGINAL ISSUE 15 AUGUST 1977
REVISED 1 MAY 1979

Digital Systems Division



© Texas Instruments Incorporated 1977, 1978, 1979

All Rights Reserved

The information and/or drawings set forth in this document and all rights in and to inventions disclosed herein and patents which might be granted thereon disclosing or employing the materials, methods, techniques or apparatus described herein are the exclusive property of Texas Instruments Incorporated.

LIST OF EFFECTIVE PAGES

INSERT LATEST CHANGED PAGES DESTROY SUPERSEDED PAGES

Note: The portion of the text affected by the changes is indicated by a vertical bar in the outer margins of the page.

Model 990 Computer FORTRAN Programmer's Reference Manual (946260-9701)

Original Issue . . . . .15 August 1977
Change 1 . . . . .15 October 1977 (ECN 419803)
Change 2 . . . . .15 December 1977 (ECN 419841)
Change 3 . . . . .1 December 1978 (ECN 419617)
Change 4 . . . . .1 January 1979 (ECN 004415)
Revised . . . . .1 May 1979 (ECN 009760)

Total number of pages in this publication is 304 consisting of the following:

PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.	PAGE NO.	CHANGE NO.
Cover	0	C-1 - C-8	0		
Effective Pages	0	D-1 - D-20	0		
iii - xiv	0	E-1 - E-20	0		
1-1 - 1-8	0	F-1 - F-2	0		
2-1 - 2-22	0	G-1 - G-12	0		
3-1 - 3-12	0	H-1 - H-20	0		
4-1 - 4-14	0	I-1 - I-4	0		
5-1 - 5-30	0	J-1 - J-6	0		
6-1 - 6-12	0	Index-1 - Index-14	0		
7-1 - 7-48	0	User's Response	0		
8-1 - 8-2	0	Business Reply	0		
9-1 - 9-12	0	Cover Blank	0		
A-1 - A-2	0	Cover	0		
B-1 - B-16	0				

PREFACE

This manual describes FORTRAN language as implemented for the 990 Computer Family. It is a reference source for FORTRAN and assembly language programmers. It includes specifications for the available types of data, descriptions of arithmetic and logical operations, rules for constructing expressions and discussions of statements for program control and Input/Output. Also included are some techniques for careful efficient programming, a discussion of the compiler output, instructions for running FORTRAN, and discussions of function and subroutine subprograms. This manual lists the functional subroutines in the FORTRAN library.

The description of 990 FORTRAN contained in this manual includes in-line comments discussing variations in the implementation of FORTRAN DX10 2.X, DX10 3.X and TXDS. This manual also includes operating instructions for DX10 3.X and TXDS. Operating instructions for 990 FORTRAN under DX10 Release 2 are contained in a separate document entitled *Model 990 Computer, DX10 Operating System Release 2, FORTRAN Operating Instructions,* part number 949619-9701.

Parts of Section I and Sections II through VI of this manual describe the features of 990 FORTRAN that are common to most FORTRAN compilers. These portions may be skimmed by an experienced FORTRAN programmer.

The 990 FORTRAN language discussion contained in this manual is divided into the following sections and appendixes.

I Section I contains an introduction to the FORTRAN language and compiler as applied to the 990 computer.

II Section II discusses the handling of data in FORTRAN. It also specifies the data conventions that may or may not be allowed in a program.

III Section III discusses the data, operators, and expressions used in FORTRAN computation. It also contains a discussion of assignment statements and type conversion.

IV Section IV describes the use of control statements which allows the programmer to manipulate and control execution of the FORTRAN program.

V Section V describes Input/Output and those statements designed to control the transmission of information between the computer and the peripheral Input/Output devices or units.

VI Section VI includes an indepth discussion of the functions and subprograms required to specify relatively complicated operations, such as the termination of a trigonometric function or the printing of specialized header information, etc.

VII Section VII describes the FORTRAN library of standard external and intrinsic functions which may be referenced from any program.

VIII Section VIII describes recommended programming techniques designed to improve program structuring by imposing a set of coding standards.

IX Section IX discusses the compiler; its output listing elements, error diagnostics, object code, and runtime error traceback information. Section IX also includes a complete FORTRAN compiler statement error diagnostic message table.

A Appendix A discusses the FORTRAN source program character set.

B Appendix B describes the calling sequences between FORTRAN and assembly language program.

C Appendix C discusses the floating point arithmetic package.

D Appendix D contains character string manipulation package.

E Appendix E contains the FORTRAN runtime description.

F Appendix F describes FORTRAN installation on TXDS.

G Appendix G discusses FORTRAN execution on TXDS.

H Appendix H discusses FORTRAN execution on a DX10 3.X system.

I Appendix I discusses FORTRAN compiler and runtime organization.

J Appendix J discusses usage of fixed point numbers.

Information related to the material discussed in this manual is found in the following publications:

Title	Part Number
Model 990 Computer DX10 Operating System Reference Manual	946250-9701 946250-9702 946250-9703 946250-9704 946250-9705 946250-9706
Model 990 Computer DX10 Operating System Programmers Guide	945257-9701
Model 990/10 Computer Program Development System Operator's Guide	945256-9701
Model 990 Computer TMS 990 Assembly Language Programmer's Guide	943441-9701
Model 990 Computer Models 306 and 588 Line Printers Installation and Operation	945261-9701
Model 990 Computer PROM Programming Module Installation and Operation	945258-9701

Title	Part Number
990 Computer Family Systems Handbook	945250-9701
Model 990 Computer Communications System Installation and Operation	945409-9701
Model 990 Computer Communication System Software	946236-9701
Model 990 Computer Terminal Executive Development System (TXDS) Programmer's Guide	946258-9701
Model 990 Computer TX990 Operating System Programmer's Guide	946259-9701
Model 990 Computer Model FD800 Floppy Disc System Installation and Operation	945253-9701
Model 990 Computer Model 913 CRT Display Terminal Installation and Operation	943457-9701
Model 990 Computer Model 733 ASR/KSR Data Terminal Installation and Operation	945259-9701
Model 990 Computer Model 804 Card Reader Installation and Operation	945262-9701
Model 990 Computer 5MT/6MT Serial Interface Module Installation and Operation	946269-9701
Model 990 Computer 32 Input/Transition Definition 32-Bit Output DAta Module and Digital Input/Output Termination Panel Installation and Operation	946267-9701
990 Link Editor Manual	949617-9701

Fundamentals of the FORTRAN language can be found in *A Guide to FORTRAN IV Programming* by Daniel D. McCracken.

TABLE OF CONTENTS

Paragraph	Title	Page
1.1	General	1-1
1.2	Program Preparation	1-2
1.2.1	Fields in a Source Record	1-2
1.2.2	FORTRAN Statements	1-3
1.2.3	Comments	1-4
1.2.4	Source Statement Ordering	1-4
1.2.5	Program Restrictions	1-4
1.3	COPY Statement	1-5
1.4	Character String Package	1-6
1.5	FORTRAN Character Set	1-6
1.5.1	Letters	1-6
1.5.2	Digits	1-6
1.5.3	Alphanumeric Characters	1-6

SECTION II. DATA SPECIFICATIONS

Paragraph	Title	Page
2.1	General	2-1
2.2	Definitions of Terms	2-1
2.2.1	Identifiers	2-1
2.2.2	Variables	2-1
2.2.3	Constants	2-2
2.3	Type Statements	2-2
2.3.1	INTEGER Statement	2-3
2.3.2	REAL Statement	2-3
2.3.3	DOUBLE PRECISION Statement	2-3
2.3.4	COMPLEX Statement	2-4
2.3.5	LOGICAL Statement	2-4
2.3.6	FIXED Statement	2-4
2.3.7	IMPLICIT Statement	2-5
2.4	Types of Data	2-5
2.4.1	Integers	2-6
2.4.2	Real Numbers	2-6
2.4.2.1	Examples of Real Numbers	2-7
2.4.2.2	Memory Representation	2-7
2.4.3	Double Precision Numbers	2-8
2.4.3.1	Examples of Double Precision Numbers	2-9
2.4.3.2	Memory Representation	2-9
2.4.4	Complex Numbers	2-9
2.4.5	Logical Constants	2-10
2.4.6	Fixed Point Integers	2-10
2.4.6.1	External Representation	2-10
2.4.6.2	Internal Representation	2-11
2.4.6.3	Examples	2-11
2.5	Hollerith Field	2-12
2.6	Dimensioned Variables	2-12
2.6.1	Arrays and Subscripts	2-12
2.6.2	Storage of Array Elements	2-14
2.6.3	DIMENSION Statements	2-14

TABLE OF CONTENTS (Continued)

Paragraph	Title	Page
2.7	COMMON and EQUIVALENCE Statements	2-16
2.7.1	COMMON Statement	2-16
2.7.2	EQUIVALENCE Statement	2-19
2.7.3	EQUIVALENCE and COMMON Interaction	2-20
2.8	Function Reference	2-20
	SECTION III. OPERATORS AND EXPRESSIONS	
3.1	General	3-1
3.2	Arithmetic Expressions	3-1
3.2.1	Arithmetic Operations	3-2
3.2.2	Formation of Arithmetic Expressions	3-2
3.2.3	Evaluation of Arithmetic Expressions	3-4
3.3	Logical Expressions	3-6
3.3.1	Relational Operations	3-6
3.3.2	Logical Operations	3-7
3.3.3	Formation of Logical Expressions	3-8
3.3.4	Evaluation of Logical Expressions	3-8
3.4	Summary of Rules for Expressions	3-9
3.4.1	Operator Precedence	3-9
3.4.2	Assignment Statements	3-9
	SECTION IV. CONTROL	
4.1	General Statements	4-1
4.2	GO TO Statement	4-1
4.2.1	Unconditional GO TO Statement	4-1
4.2.2	Computed GO TO Statement	4-2
4.2.3	ASSIGN and Assigned GO TO Statements	4-2
4.3	IF Statement	4-4
4.3.1	Arithmetic IF Statement	4-4
4.3.2	Logical IF Statement	4-4
4.4	DO and CONTINUE Statements	4-5
4.4.1	DO Statement	4-5
4.4.2	CONTINUE Statement	4-9
4.5	Transfer of Control to Subroutines	4-10
4.5.1	CALL Statement	4-10
4.5.2	RETURN Statement	4-11
4.6	PAUSE, STOP and END Statements	4-11
4.6.1	PAUSE Statement	4-12
4.6.1.1	DX10 PAUSE Statement 2.X Releases	4-12
4.6.1.2	DX10 PAUSE Statement 3.X Releases	4-12
4.6.1.3	TXDS PAUSE Statement	4-12
4.6.2	STOP Statement	4-12
4.6.2.1	DX10 Statement 2.X Releases	4-12
4.6.2.2	DX10 STOP Statement 3.X Releases	4-13
4.6.2.3	TXDS STOP Statement	4-13
4.6.3	END Statement	4-13

TABLE OF CONTENTS (Continued)

Paragraph	Title	Page

SECTION V. INPUT/OUTPUT STATEMENTS

5.1	General	5-1
5.2	FORMAT Statement	5-1
5.2.1	Numerical Data Specifications	5-1
5.2.1.1	Field Specifications	5-2
5.2.1.2	F Field Specifications	5-3
5.2.1.3	E Field Specifications	5-4
5.2.1.4	G Field Specifications	5-5
5.2.1.5	D Field Specifications	5-5
5.2.1.6	Z Field Specifications	5-5
5.2.1.7	Scale Factors	5-6
5.2.2	Logical Data Field Specifications	5-7
5.2.3	Alphanumeric and Literal Data Specifications	5-7
5.2.3.1	Carriage Control	5-7
5.2.3.2	A Field Specification	5-8
5.2.3.3	H Field Specification	5-8
5.2.3.4	X Field Specification	5-9
5.2.3.5	T Field Specification	5-9
5.2.4	Complex Quantities	5-10
5.2.5	Repeated Group and Field Specifications	5-10
5.2.5.1	Repetition of Field Specifications	5-10
5.2.5.2	Repetition of Groups	5-10
5.2.6	Record Separation Indicator (Slash)	5-11
5.2.7	Formats Stored as Data	5-12
5.2.8	Combinations of Format	5-12
5.2.9	Free Field Format	5-12
5.3	READ and WRITE Statements	5-13
5.3.1	READ and WRITE Sequential File Characteristics	5-13
5.3.2	READ and WRITE Record Characteristics	5-14
5.3.3	READ and WRITE Input and Output Lists	5-14
5.3.4	Formatted READ Statement	5-15
5.3.5	Formatted WRITE Statement	5-16
5.3.6	Unformatted READ and WRITE Statements	5-16
5.4	Internal Transmission	5-17
5.4.1	ENCODE Statement	5-17
5.4.2	DECODE Statement	5-18
5.5	Mass Storage File Input/Output Statements	5-18
5.5.1	REWIND Statement	5-19
5.5.2	BACKSPACE Statement	5-19
5.5.3	END FILE Statement	5-19
5.6	Direct Access Input/Output	5-19
5.6.1	DEFINE FILE Statement	5-20
5.6.2	READ Statement	5-23
5.6.3	WRITE Statement	5-24
5.6.4	FIND Statement	5-24
5.7	Console Display Input/Output	5-24
5.7.1	ACCEPT Statement	5-25

TABLE OF CONTENTS (Continued)

Paragraph	Title	Page
5.7.2	DISPLAY Statement	5-25
5.8	Logical and Physical Record Characteristics	5-27
5.9	FORTRAN Unit Numbers	5-28
5.10	Valid FORTRAN I/O Operations	5-28
	SECTION VI. FUNCTIONS AND SUBPROGRAMS	
6.1	General	6-1
6.2	Subprograms	6-1
6.2.1	Definitions	6-1
6.2.2	Dummy Identifiers	6-1
6.3	Statement Functions Definitions	6-2
6.4	FUNCTION Subprogram	6-3
6.5	SUBROUTINE Subprogram	6-5
6.6	Specification Subprograms for Data Initialization	6-6
6.6.1	DATA Statement	6-7
6.6.2	BLOCK DATA Statement	6-9
6.7	EXTERNAL Statement	6-9
6.8	REENTRANT Statement	6-10
	SECTION VII. FORTRAN LIBRARY	
7.1	General	7-1
7.2	Library Subroutines	7-1
7.2.1	BUFIN Subroutine	7-1
7.2.2	BUFOUT Subroutine	7-5
7.3	ISA Extensions	7-6
7.3.1	Start a Program	7-6
7.3.2	Start a Program at a Specified Time	7-7
7.3.3	Delay Continuation of a Program	7-8
7.3.4	Digital Input	7-8
7.3.5	Latched Digital Output	7-9
7.3.6	Momentary Digital Output	7-10
7.3.7	Obtain Date	7-10
7.3.8	Obtain Time	7-10
7.3.9	Analog Data Handling	7-11
7.3.10	CFILW Subroutine	7-13
7.3.11	DFILW Subroutine	7-14
7.3.12	OPENW/CLOSEW Subroutine	7-15
7.3.13	RDRW/WRTRW Subroutine	7-16
7.3.14	SVCFUT Subroutine	7-17
7.3.15	MODAPW Subroutine	7-18
7.4	990 FORTRAN Callable Subroutines	7-19
7.4.1	Pseudo-Random Number	7-19
7.4.2	Preset Random Number Generator	7-19
7.4.3	Generate Supervisor Call	7-19
7.4.4	Absolute Address	7-19
7.4.5	Bid Task	7-20
7.4.6	Delayed Bit Task	7-20

TABLE OF CONTENTS (Continued)

Paragraph	Title	Page
7.4.7	CRU Input	7-20
7.4.8	CRU Output	7-20
7.4.9	Obtain Date and Time	7-21
7.4.10	Obtain ASCII Date	7-21
7.4.11	Obtain ASCII Time	7-21
7.4.12	Obtain Military Date	7-21
7.5	FORTRAN-PROM Program Subroutines	7-22
7.5.1	Limitations	7-22
7.5.2	Subroutine Package Modules	7-22
7.5.2.1	IMGBLD Module	7-22
7.5.2.2	PRGROM Module	7-22
7.5.2.3	RDPROM Module	7-22
7.5.3	User FORTRAN Routine	7-22
7.5.3.1	Inputting the PROM Program from Mass Storage	7-22
7.5.3.2	Writing the PROM	7-24
7.5.3.3	Reading the PROM	7-26
7.5.4	Sample Program Descriptions	7-27
7.6	5MT/6MT Serial Interface Module Subroutines	7-31
7.6.1	OPENMT	7-32
7.6.2	CLOSMT	7-32
7.6.3	RDSTS	7-33
7.6.4	RDMTR	7-33
7.6.5	RDMTS	7-34
7.6.6	WRMTR	7-35
7.6.7	WRMTS	7-36
7.7	32 Input/Transition Detection Module Subroutines	7-37
7.7.1	OPN32I	7-38
7.7.2	CLS32I	7-38
7.7.3	WRTMSK	7-38
7.7.4	READ32	7-39
7.7.5	RDIBIT	7-40
7.8	Multi-key Index File Handler	7-41

SECTION VIII. PROGRAMMING TECHNIQUE

SECTION IX. FORTRAN COMPILER OUTPUT

Paragraph	Title	Page
9.1	General	9-1
9.2	Output Listing Elements	9-1
9.3	Error Diagnostics and Messages	9-1
9.3.1	Statement Error Diagnostics	9-1
9.3.2	Program Error Diagnostics	9-2
9.3.3	Runtime Error Diagnostics	9-2
9.4	Object Code	9-2
9.5	Runtime Error Traceback Information	9-9

APPENDIXES

Appendix	Title	Page
A	FORTRAN Source Program Character Set	A-1
B	Calling Sequences Between FORTRAN and Assembly Language Programs	B-1
C	Floating Point Arithmetic Package	C-1
D	Character String Manipulation	D-1
E	FORTRAN Runtime Description	E-1
F	FORTRAN Installation on TXDS	F-1
G	FORTRAN Execution on TXDS	G-1
H	FORTRAN Execution on a DX10 3.X System	H-1
I	FORTRAN Compiler and Runtime Organization	I-1
J	Usage of Fixed Point Numbers	J-1

LIST OF ILLUSTRATIONS

Figure	Title	Page
2-1	Memory Representation of Real Numbers	2-8
2-2	Memory Representation of Double Precision Numbers	2-9
5-1	Define File Example Output	5-22
6-1	Example of Subroutine	6-6
7-1	IMGBLD Data Flow	7-23
7-2	Example: Read a 2708 EPROM	7-27
7-3	Example: Write Zeros to a 2708 EPROM	7-28
7-4	Example: Write a Bipolar S287 PROM to Ones	7-29
7-5	Example: Read and Relocate an Object Module	7-30
7-6	RDIBIT Calling Program Buffer	7-40

LIST OF TABLES

Table	Title	Page
1-1	Fields in a FORTRAN Program Line	1-3
1-2	Ordering of FORTRAN Source Statements	1-5
2-1	Example of an Array: Precipitation in Selected United States Cities	2-13
3-1	Precedence of Operators	3-9
3-2	Assignment Statement Type Conversion	3-10
5-1	Validate DX FORTRAN I/O Operations	5-29
5-2	Validate TX FORTRAN I/O Operations	5-30

LIST OF TABLES (Continued)

Table	Title	Page
7-1	FORTRAN Library Basic External Functions	7-2
7-2	FORTRAN Library Intrinsic Functions	7-4
7-3	Error Table	7-24
7-4	5MT/6MT ISA Subroutines	7-31
7-5	32 IT Module ISA Subroutines	7-37
9-1	FORTRAN Compiler Statement Error Diagnostic Messages	9-3
9-2	FORTRAN Compiler Program Error Diagnostic Messages	9-6
9-3	FORTRAN Runtime Package Diagnostic Messages	9-8
9-4	FORTRAN Runtime Error Messages	9-10

SECTION I

INTRODUCTION

1.1 GENERAL

This manual describes the FORTRAN language and compiler for the Model 990 Computer. The compiler meets the specifications set forth in American National Standards Institute publication USAS X3.9-1966; such a compiler is referred to as ANSI standard FORTRAN or FORTRAN IV.

A FORTRAN program is a series of FORTRAN statements that accomplish a problem-solving task involving mathematical and logical computations. FORTRAN statements may be one of the following four types:

- Arithmetic operations
- Modifications of the flow of control within the program
- Declarations containing information about procedures and data
- Input/output operations

FORTRAN statements must be transformed, or translated, into a form that may be handled conveniently within the computer. This form is machine code, which is handled within the computer system hardware as a series of binary digits. FORTRAN is a high level language whose form is much more like mathematical statements and English language words than machine code. The FORTRAN statements are the *source code* and the machine code is the *object code*.

The version of FORTRAN implemented on the 990 computer includes extensions to ANSI standard FORTRAN that provide increased flexibility. These extensions include:

- Internal data manipulation statements
- Variable names of any length
- General integer expressions in subscripts
- Video Data Terminal data handling statements
- Direct Access I/O
- Mixed mode expressions
- Hollerith and hexadecimal constants and assignments
- Extended Integers
- 16-bit fixed-point arithmetic
- Implicit variable typing

In addition, 990 FORTRAN incorporates the extensions to FORTRAN language recommended by the Instrument Society of America (ISA Extensions S61.1-1975). These features include:

- Control of program starting (immediate, execution after a delay, or execution at a time of day).
- Delay of program continuation
- Control of analog and digital I/O (analog input in sequential order or in any order, analog output, and digital input and output).
- Logical operations (OR, AND, NOT, Exclusive OR)
- Bit string shifts
- Bit testing and setting
- Time and date information

990 FORTRAN under the DX10 3.X operating systems, provides the ISA FORTRAN (S61.2-1976) procedures for file access and the control of file contention. These external procedures provide means for accessing files, and for resolving problems of file access contention in a multiprogramming environment. These procedures include:

- Create a file
- Delete a file
- Open a file
- Close a file
- Modify access privileges
- Input/output to unformatted direct access files

NOTE

> These procedures are not supported under release 2.X of the DX10 operating system.

1.2 PROGRAM PREPARATION

Each statement in a FORTRAN program appears in a single line; however, continuation lines for individual statements may be used.

If cards are used as the source medium, each line is punched on a separate card, using the Hollerith character code. If disk or cassette is the source medium, each line is recorded in ASCII code.

1.2.1 FIELDS IN A SOURCE RECORD. Each line of the program, corresponding to a record, consists of 80 characters or less that are divided into four fields, as shown in table 1-1.

Statement numbers that are not needed may decrease efficiency during compilation and should, therefore, be avoided. The last statement of each program must be an END statement. In addition, each FORTRAN source file must be terminated with the end-of-file record.

NOTE

When the source is on cards, the end-of-file record consisting of a slash (/) in the first character position and an asterisk (*) in the second position is the last card.

1.2.2 FORTRAN STATEMENTS. A statement consists of an initial line and any number of continuation lines. An initial line is a line that is neither a comment line nor an END line and contains either a blank or the digit 0 in Column 6. A continuation line is not a comment line and contains any character other than blank or 0 (zero) in Column 6. The character in Column 6 is not recognized as a statement character and only serves to indicate continuation. An END line is a line containing an END statement, which cannot be continued.

Example:

Card Column	1	2	3	4	5	6	7	8	9	10	11 12 . . .
			2	0	0		A	=	B	+	
						X	C				
						X	+	D			

Table 1-1. Fields in a FORTRAN Program Line

Columns	Field
1-5	*Statement Number.* Serves as statement identifier for cross-references. Number optional as identifier for other statements. The statement number consists of one to five digits of any value from 1 to 99999. Blanks and leading zeros are ignored; however, zero may not be a statement number. Statement numbers may appear anywhere within the field but must not contain nonnumeric characters. They may be assigned in any order. The sequence of operations depends on the order of the statements in the program, not on the statement number.
	Two alphabetic characters (C and D) may be placed in column 1. The character C in column 1 indicates the program appears in columns 2-72. The character D in column 1 treats columns 2-72 as comments if conditional compilation is not specified; if specified, these columns are treated as though column 1 contained a blank.
6	*Continuation Indicator.* A nonzero, nonblank character indicates that the line is a continuation line. Any number of continuation lines are allowed, up to the memory capacity limit.
7-72	*Statement Field.* The FORTRAN statement appears in these columns. Both the initial and continuation lines occur within these field limits. Blanks in these columns are ignored except in certain alphanumeric fields, and may be used to improve readability.
73-80	*Identification Field.* These columns are not used by the FORTRAN compiler. They may be used as an identifier for the program, for sequencing, or any other similar purpose.

The first line of this example is an initial line. The second and third lines are continuation lines. The statement label is 200. The statement is equivalent to:

```
A = B + C + D
```

One kind of FORTRAN statement is used to compute a new value of a variable. This is done with an assignment statement of the form:

```
variable = expression
```

where a variable name appears to the left of the equal sign and an expression to the right. The value of the expression is calculated and that value is given to the variable on the left. Variables and expressions are discussed in more detail in Section II.

Another example of a valid FORTRAN statement is

```
J = J + 1
```

This statement calculates the sum of the current value of J and one, and assigns the sum as the new value of J. Note that this is not a valid algebraic statement from a mathematical viewpoint. The value of J is replaced with a new value.

1.2.3 COMMENTS. Comments are for the convenience of the programmer to describe the program. A comment line cannot be followed by a continuation line. It can only be followed either by another comment line or by the initial line of a statement.

Comment lines have the character C in Column 1. Columns 2 through 72 may be used in any desired format.

1.2.4 SOURCE STATEMENT ORDERING. Table 1-2 shows the order in which source statements of each program must be written. Within each group the statements may be written in any sequence. DATA statements may appear anywhere after Group 2 and before Group 7, but must appear after any declarations (COMMON, DIMENSION, or type) affecting the variables to be initialized. FORMAT statements may appear anywhere after Group 1 and before Group 7. COPY statements may appear anywhere before Group 7.

1.2.5 PROGRAM RESTRICTIONS. The programmer must observe certain program size restrictions when developing a FORTRAN program (due to the nature of the compiler constructions). The number of each of the following terms within each main program, subroutine, or function must be less than 1024:

- Scalar Variables
- Array Variables
- Common Variables
- Equivalenced Variable Names
- Statement Numbers
- Names in Explicit Type Statements

- Unique Extended INTEGER, REAL, DOUBLE PRECISION and COMPLEX Constants
- Unique INTEGER Constants
- Unique LOGICAL Constants
- Unique Subprograms Called
- Arithmetic Statement Function Definitions

1.3 COPY STATEMENT

The COPY statement enables parts of the program to be stored in more than one file and allows many FORTRAN programs to use a single common source. This statement appears as follows:

COPY name

where name is a full-qualified file access name or synonym (under DX10). The name must be completely contained on one line (not extended across continuation lines) and may not contain blanks. A DX10 program may contain any number of COPY statements, but they should not be nested deeper than five. The file is rewound before the COPY operation begins.

NOTE

COPY statements may NOT be nested at all in TXDS.

Table 1-2. Ordering of FORTRAN Source Statements

Group	Source Statement	Group	Source Statement	Group	Source Statement
1	BLOCK DATA COPY FUNCTION SUBROUTINE	4 (Cont.)	EXTERNAL FORMAT REENTRANT	6 (Cont.)	DISPLAY DO ENCODE ENDFILE FIND FORMAT GO TO IF PAUSE READ RETURN REWIND STOP WRITE
2	COPY FORMAT IMPLICIT	5	COPY DATA* EQUIVALENCE FORMAT Statement Function		
3	COMPLEX FIXED INTEGER LOGICAL REAL DOUBLE PRECISION	6	ACCEPT Assignment ASSIGN BACKSPACE CALL CONTINUE COPY DATA* DECODE DEFINE FILE		
4	COMMON COPY DATA* DIMENSION			7	END

*Must follow declarations for the initialized variables.

The contents of the named file are inserted into the source program such that the first record of the file will be the next line after the COPY statement. Thus, COPY statements may be labeled and referenced the same as CONTINUE statements.

For example, if the DX10 file, .COPY.DATA contains FORTRAN source, this source can be included in another FORTRAN source program with a COPY statement as follows:

```
      .
      .
      .
      COPY  .COPY.DATA
      .
      .
      .
      END
```

1.4 CHARACTER STRING PACKAGE

This package will enable the user to:

1. Move characters into and out of arrays
2. Compare characters
3. Check for illegal characters
4. Verify characters
5. Find the length of character strings
6. Translate characters

The character string package was written so that the user need not concern himself with word boundaries or data type when manipulating character strings.

1.5 FORTRAN CHARACTER SET

The FORTRAN character set consists of twenty-six letters, ten digits, and twelve special characters.

1.5.1 LETTERS.

A letter is one of the twenty-six characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

1.5.2 DIGITS.

A digit is one of the ten characters:

0 1 2 3 4 5 6 7 8 9

1.5.3 ALPHANUMERIC CHARACTERS.

An alphanumeric character is a letter or a digit.

1.5.4 SPECIAL CHARACTERS. A special character is one of the twelve characters:

Character	Name of Character
	Blank
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Period
'	Apostrophe
>	Greater Than Sign

Other characters may appear in a FORTRAN statement only as part of a Hollerith constant or alphanumeric literal. Any printable character may appear in a comment. Refer to Appendix A for additional characters.

SECTION II

DATA SPECIFICATIONS

2.1 GENERAL

This section discusses the handling of data in FORTRAN, and specifies the data conventions that are or are not allowed in a program. The topics discussed include:

- How data may be represented in FORTRAN statements.
- Types of data.
- FORTRAN statements that specify or manipulate data.
- Mathematical functions.

Data used with 990 FORTRAN is classified as integers, real numbers, double precision numbers, complex numbers, logical, and fixed. Data of these classes may be handled as individual data elements or grouped in multidimensional arrays. FORTRAN statements define the data characteristics and data arrangements for each program. If declarations are not stated, the compiler defaults to predefined characteristics.

2.2 DEFINITIONS OF TERMS

The terms identifier, variable and constant which are used to specify and define data must be understood before any data characteristics are discussed.

2.2.1 IDENTIFIERS. An identifier is a character string that represents a variable, function or subprogram in a program. The identifier may be any alphanumeric combination as long as the first character is a letter. The compiler does not restrict the number of characters that may be used in an identifier. When the identifier is used in the program, however, the compiler recognizes only the first six characters of the identifier. Therefore, no two identifiers can contain the same characters in the first six character positions. If MODIFICATION is used as an identifier, for example, MODIFIER2 cannot be used. The first six characters in both is MODIFI. Since blanks are not significant within identifiers, the following example is valid:

```
ROOT OF EQUATION = SQRT (FIRST COEFFICIENT)
```

2.2.2. VARIABLES. Variables are data whose values can change during the execution of a program. Becuase the value is dynamic, variables are assigned unique, fixed identifiers for reference within the program. In a program, the value of the variable is the most recent value stored in that variable's storage area. For example, the following three statements define the relationship of the variable identifiers, A and B:

```
A = 8.0

B = A + 2.0

A = A + 1.0
```

The first statement assigns a value to A. The second statement defines the relationship of B to A. The value of A is used to calculate the value of B. The third statement assigns a new value to A by adding 1.0 to the old value and storing the new value in the same storage location. A variable identifier may refer to data in any of the six categories of data listed in paragraph 2.1.

2.2.3 CONSTANTS. Constants are data that do not change in value during the execution of a program. They are not assigned an identifier and are referred to by their explicit values. For example, the integer seven is represented in a program as the number "7".

2.3 TYPE STATEMENTS

Data may be one of six types named in paragraph 2.1, two of which (integers and real numbers) may have variable names implicitly defined:

- If the variable name starts with the letter I, J, K, L, M or N, it is an integer variable.
- If the variable name starts with any other letter, it is a real variable.

The rules for integers and real numbers may be altered through the use of an IMPLICIT statement. The other four data types must have variable names that are explicitly declared in the program to be that particular data type. (Data types are described in more detail in paragraph 2.4.) In addition, if the programmer wishes to use a variable name for integer or real number data that does not conform to the rules above, for example a real variable name beginning with the letter I, he may do so. Assignment of variable names to data types in this manner is accomplished through a *type statement*.

The type statement is one example of the FORTRAN statements known as a *declaration*. FORTRAN declarations are used to supply descriptive information about the program rather than specify computation or other action. Such descriptive information is concerned primarily with the interpretation of source program identifiers and object program storage allocation.

Six type statements are used to explicitly specify identifier types appearing in the program. These include: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, and FIXED. The following rules apply to these type statements:

- The name of the statement type is followed by identifiers (used for variable names) that are separated by commas. The FIXED statement includes a scale factor in parenthesis before the identifier.
- Type declaration statements must precede any nondeclarative statements, arithmetic function definition statements or DATA statements in the program. Arithmetic function definition statements and DATA statements are discussed in Section VI.
- Each type statement must appear in the program before the first use of any identifiers named in the statement.
- An identifier may appear in only one type statement.
- Type statements may be used to declare arrays that are *not* dimensioned in either the DIMENSION or the COMMON statement.

A statement similar to type statements is available for declaring a function name to be an argument in a subprogram call. This statement, the EXTERNAL statement, is described in Section VI.

2.3.1 INTEGER STATEMENT.

Form:

```
INTEGER identifier, identifier, . . .
     or
INTEGER*2 identifier, identifier, . . .
     and
INTEGER*4 identifier, identifier, . . .
```

The first two forms are equivalent and are used to declare the listed identifiers to be integer type with each datum occupying one word (two bytes). The third form, INTEGER*4, declares extended integer type data occupying two words (four bytes) each.

Examples:

```
INTEGER ALPHA,P

INTEGER*4 LAMBDA
```

2.3.2 REAL STATEMENT.

Form:

```
REAL identifier, identifier, . . .
     or
REAL*4 identifier, identifier, . . .
     and
REAL*8 identifier, identifier, . . .
```

The first two forms are equivalent and are used to declare the listed identifiers to be real type with each datum occupying two words (four bytes) in floating point format. The last form, REAL*8, declares four words (eight bytes) of floating point data. REAL*8 is equivalent to DOUBLE PRECISION.

Examples:

```
REAL LOGX, MASS(10,4)

REAL*8 A
```

2.3.3 DOUBLE PRECISION STATEMENT.

Form:

```
DOUBLE PRECISION identifier, identifier, . . .
```

This statement declares the listed identifiers to be of double precision type. Double precision data occupies four words in floating point format.

Example:

DOUBLE PRECISION RATE,Y,FLOW

2.3.4 COMPLEX STATEMENT.

Form:

COMPLEX identifier, identifier, . . .

This statement declares the identifiers to be of complex type. Each datum occupies four words, two floating point numbers representing the real and imaginary parts.

Example:

COMPLEX ZETA,W,ROOT

2.3.5 LOGICAL STATEMENT.

Form:

LOGICAL identifier, identifier, . . .

This statement declares the listed identifiers to be of logical type. Each datum occupies one word where zero represents .FALSE. and nonzero represents .TRUE..

Example:

LOGICAL BOOL,P,Q,ANSWER

2.3.6 FIXED STATEMENT.

Form:

FIXED (scale) identifier, identifier, . . .

Scale is a signed or unsigned integer constant within the range of $-31 \leq scale \leq 31$. The FIXED statement declares the listed identifiers to be of fixed type with the stated binary scale factor. Each datum occupies one word.

Example:

FIXED(3) WEEKLY,NET

This statement declares the two variables WEEKLY and NET are of fixed type with three bits to the right of the binary point (the value is multiplied by 2^{-3}).

Example:

FIXED(-21)HUGE(10)

This statement declares an array with the binary point of each element assumed to be 21 bits to the right of the word; i.e., the value is multiplied by 2^{21}.

Examples of fixed integers are given in paragraph 2.4.6.

2.3.7 IMPLICIT STATEMENT. The IMPLICIT statement establishes the implicit type of an identifier if the programmer wishes it to be different from the default type wherein identifiers beginning with letters I, J, K, L, M, N are integer, and others are real.

Format:

IMPLICIT $type_1(A_1,A_2,\ldots),type_2(A_3,A_4,\ldots),\ldots$

where $type_i$ represents one of the following: INTEGER, INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, DOUBLE PRECISION, COMPLEX, LOGICAL, FIXED (scale); and A_1, $A_2,\ldots$ represent single alphabetic characters or a range of characters (in alphabetic sequence) denoted by the first and last character of the range separated by a minus sign (e.g., A-D).

This statement applies only to variables not mentioned in an explicit type statement. Variables, whose first character is listed following a type identifier assume the type appearing before the list. Unlisted variables assume their default types.

Example:

```
IMPLICIT INTEGER (A-C,X),DOUBLE PRECISION(D),LOGICAL(L)
```

This statement would cause the following implicit declarations to be in effect:

1. Identifiers beginning with A, B, C, I, J, K, M, N, X are integer.
2. Identifiers beginning with D are double precision.
3. Identifiers beginning with L are logical.
4. Identifiers beginning with E, F, G, H, O, P, Q, R, S, T, U, V, W, Y, Z are real.

2.4 TYPES OF DATA

The compiler recognizes six types of data as input for processing. These types and the number of words of machine storage they occupy are as follows:

Data Type	No. of Storage Words
Integer	1 or 2
Real number	2
Double precision number	4
Complex number	4
Logical quantity	1
Fixed	1

The following paragraphs describe each of these data types. Data type declarations (described in paragraph 2.3) are used to specify a particular data type for use in a program.

There are two other types of data for which data type declarations do not exist. They are Hollerith data, discussed in paragraph 2.5, and functions, discussed in paragraph 2.8. Declarations for Hollerith data and functions are handled in a different manner.

2.4.1 INTEGERS. Integers are whole numbers whose least significant digit represents the one's position. Integers may be positive, negative or zero. If unsigned, they are assumed to be positive. If not explicitly declared, an integer variable name must start with the letter I, J, K, L, M or N.

Integers cannot contain a decimal point. The compiler assumes that the decimal point is located to the right of the rightmost digit position. A comma may not be used in the number (to separate the thousands digit from the hundreds digit, for example). Only numbers are valid characters in the integer.

Simple integers occupy one 16-bit word; extended integers occupy two 16-bit words. In either case the high-order bit is reserved for the sign. Therefore, the range of integers in 990 FORTRAN is -32768 to +32767 (-2^{15} to $+2^{15}-1$) for simple integers and -2147483648 to +2147483647 (-2^{31} to $+2^{31}-1$) for extended integers.

For example, the following numbers are valid integers for input to the compiler:

3298

-3298

1234567890

0

0001

The following numbers are not valid integers for input to the compiler:

3,287	Commas cannot be used in integer data.
42.	Decimal points cannot be used in integer data.
3234567890	This number exceeds the upper bound of 2147483647 for positive integer values.

Hexadecimal constants may be written in two forms:

$>D_iD_2\ldots$

$nZD_iD_2\ldots Dn$

where n is an unsigned integer constant, Z or > indicates hexadecimal, and D_i represents a hexadecimal digit (0-9, A-F). Hexadecimal constants are stored internally right justified in one word, if there are four or less hexadecimal digits, or in two words. They are treated as integers or extended integers within expressions.

2.4.2 REAL NUMBERS. Real numbers contain up to seven significant decimal digits and must have a decimal point. Real numbers may have an exponent in the input and can represent any values within the approximate range 10^{-78} to 10^{75}, including zero. A real number may be represented in either of two forms:

- Digits and decimal point, which may be preceded by a plus or minus sign: a number in the range -32768. to +32767.
- Power-of-ten form: a number times 10 raised to an integer power.

The power-of-ten form is as follows:

zzzEn

where the letters "zzz" represent a numerical value of any number of digits (only the first seven are significant) that may or may not contain a decimal point. The letter E indicates that the number following it, represented by "n", is a power of ten. The power may be an integer; positive, negative or zero. The number must be within the specified range for real numbers.

An unsigned number is assumed to be positive. Commas cannot appear in the data input. If not explicitly declared, a real variable name must start with a letter of the alphabet other than I, J, K, L, M or N.

2.4.2.1 Examples of Real Numbers. The following numbers are correct representations of real numbers:

1387.	
1.23456	
1.37E2	(1.37×10^2)
137E-2	(137×10^{-2} or 1.37)
-4.53716E2	(-4.53716×10^2)
3E3	(3×10^3)

The following numbers are not correctly formulated real numbers for input to the compiler:

387	Numbers not in exponential form require a decimal point.
3.E	The exponent value has been omitted after the letter E.
3,871.E2	Commas cannot be used in real number data input.
5E2.5	Nonintegral exponent of ten is not allowed.
4.72E76	Number larger than upper bound of permissible range.

2.4.2.2 Memory Representation. The real numbers described in paragraph 2.4.2 are single precision. Single precision real numbers are stored in memory in two 16-bit words as illustrated in figure 2-1. Before being stored in memory, however, the number is transformed to a normalized hexadecimal fraction, a corresponding hexadecimal exponent that is a power of 16 and a sign bit.

The fraction portion of the number is normalized; that is, it is shifted to the left to eliminate leading zeros between the radix point and the first significant bit of the fraction. Normalization is by hexadecimal digits; therefore, the number is shifted 4 bits at a time until the left most 4 bits are not all zeros. Each bit position shift in the normalization process produces a corresponding change in the exponent portion of the number to maintain the correct magnitude of the number. When completely normalized, the hexadecimal fraction is stored in bits 8 through 15 of the first memory word and in the entire second memory word. The radix point for the fraction is assumed to be positioned between bits 7 and 8 of the first memory word (at the start of the hexadecimal fraction).

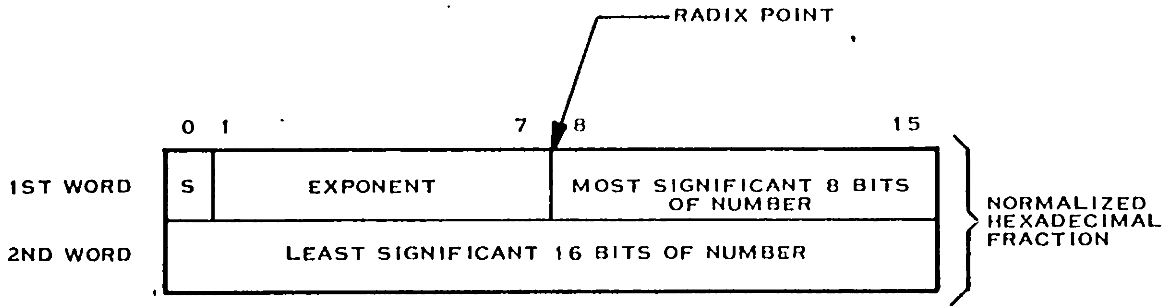


Figure 2-1. Memory Representation of Real Numbers

The exponent portion of the number is biased by 40_{16} (excess 64 notation), so that an exponent for the number 16^0 is represented in memory by 40_{16}. Positive exponents, therefore, are represented by numbers greater than 40_{16}, and negative exponents are represented by numbers less than 40_{16}. For example, 16^{-8} is represented in the exponent field by a value of 38_{16}. The exponent may be any value from 00_{16} to $7F_{16}$. Using the 40_{16} bias value, these numbers represent exponent values from -40_{16} to $+3F_{16}$ (16^{-64} to 16^{63}). The seven exponent bits are stored in bits 1 through 7 of the first memory word.

Bit 0 of the first memory word is used for a sign bit. When this bit is a zero, the number is positive; when this bit is one, the number is negative.

Examples:

Base Ten Number	Hexadecimal Contents of Memory Words Word 1	Word 2
1.0	4110	0000
0.5	4080	0000
100.0	4264	0000
.03125 (1/32)	3F80	0000
-1.0	C110	0000

2.4.3 DOUBLE PRECISION NUMBERS. Double precision numbers are similar to real numbers, except that they occupy two more memory words and provide up to sixteen significant digits (56 bits) instead of the seven available with real numbers. Double precision numbers, like real numbers, are restricted to the range of values from 10^{-78} to 10^{75}, including zero. The numbers may be positive, negative or zero; if unsigned, they are assumed to be positive. The numbers must be written in an exponential form similar to real numbers:

zzzDn

where zzz represents any number of digits (only the first sixteen are significant) that may or may not include a decimal point. The letter D indicates that the number following it is a power of ten; it must be included in the data format. The power, represented by "n", may be positive, negative or zero, and must be within the specified range for double precision numbers. Commas cannot appear in the data input.

2.4.3.1 **Examples of Double Precision Numbers.** The following numbers are correct representations of double precision numbers:

3D-16	(3×10^{-16})
487.318264D0	(487.318264)

The following numbers are not correctly formulated double precision numbers for input to the compiler:

3.8D+327	Exponent exceeds the upper bound of 75.
4,134,904D5	Commas cannot be used for double precision data input.
8.333333D-2.7	Nonintegral exponent of ten is not allowed.
8.33333333333	Letter D not included.

2.4.3.2 **Memory Representation.** Double precision real numbers are stored in memory in four 16-bit words as illustrated in figure 2-2. The most significant bit of the first word is a sign bit for the data value field: 0 if the number is positive and 1 if it is negative. Bits 1 through 7 of the first word are the exponent. The exponent follows the same form as for real number exponents. The remaining bits of the first word of the other three words contain the double precision number.

2.4.4 COMPLEX NUMBERS. A complex number has two components, a real part and an imaginary part. The imaginary part is a real number multiplied by $i = \sqrt{-1}$. Such a number may be represented graphically on a two-dimensional plane with perpendicular axes, one axis for the real part and one for the imaginary part.

In 990 FORTRAN, complex numbers are written as an ordered pair of real numbers in this form:

(c_1,c_2)

where c_1 represents the real part of the complex number and c_2 represents the coefficient of i for the imaginary part of the complex number. Since c_1 and c_2 are both real numbers, they must conform to the format for real numbers described in paragraph 2.4.2. Each of these

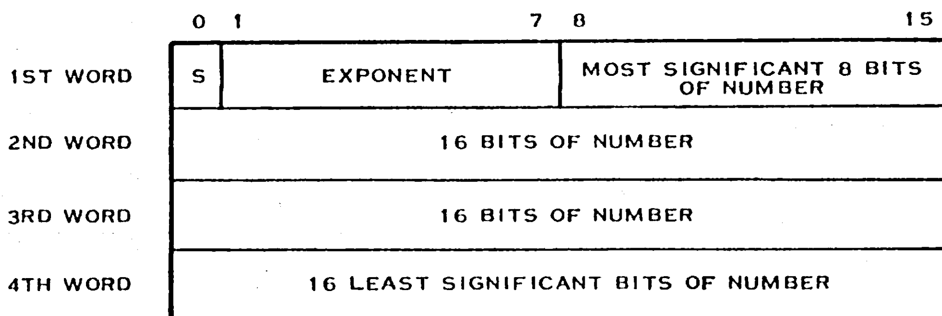


Figure 2-2. Memory Representation of Double Precision Numbers

numbers may be signed or unsigned. The parentheses and comma are required in the complex number representation. The following are correctly formatted complex numbers for use with the compiler:

(1.25,-8.6)	Represents the expression 1.25 - 8.6i
(0.,0.)	Represents the expression 0 + 0i
(2.5E-7,-3.)	Represents the expression $2.5 \times 10^{-7} - 3i$

2.4.5 **LOGICAL CONSTANTS.** A logical constant may assume one of two logical states, true or false. These states are expressed as follows for use with the compiler:

.TRUE.

.FALSE.

The enclosing periods are part of the constant and must be used regardless of context. These values can be assigned to variables just as numerical values can, provided the variables have been previously declared to be logical variables (refer to the declarations of data types in paragraph 2.3). A logical constant occupies one word (16 bits) in memory. That word is either all zeros (.FALSE.) or any nonzero value (.TRUE.). The following expression is valid if the variable A has been declared to be a logical variable:

```
A = .TRUE.
```

The use of the relational operators and logical operators in logical expressions is described in Section III.

2.4.6 **FIXED POINT INTEGERS.** A fixed point integer is an integer data type having a binary scale factor associated with it. The motivation behind this scaled integer feature is to provide a mechanism for performing simple floating-point-like operations using integer arithmetic which is much faster than floating point arithmetic. The feature is also useful in scaling and normalization for accuracy optimization.

2.4.6.1 **External Representation.** Fixed point integers are represented in source text by variable names declared in a FIXED type statement or by constants of the form:

±iQ±s

where:

i is the integer portion and consists of a string of decimal digits with no periods allowed.

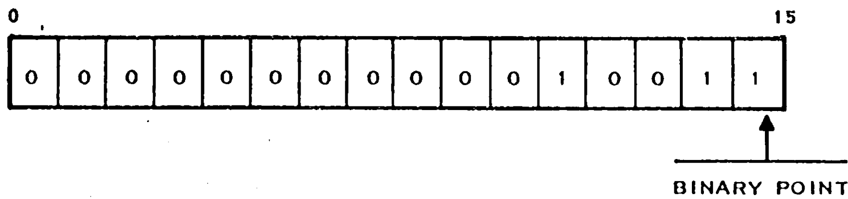
i must be in the range -32768 to 32767.

s is the scale factor portion and consists of one or two decimal digits.

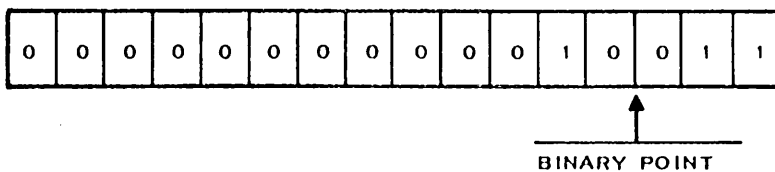
s must be in the range -31 to 31.

2.4.6.2 Internal Representation. The integer portion is represented internally as a 16-bit two's complement binary number. The scale factor then represents the position of the binary point much like the integer portion. A scale factor of zero indicates the binary point is just right of the least signficant bit of the integer portion. This orientation of the binary point is identical to that of the usual integer data type. A positive scale factor s indicates the binary point is s bits to the left of the least significant bit; a negative scale factor, s bits to the right. For example:

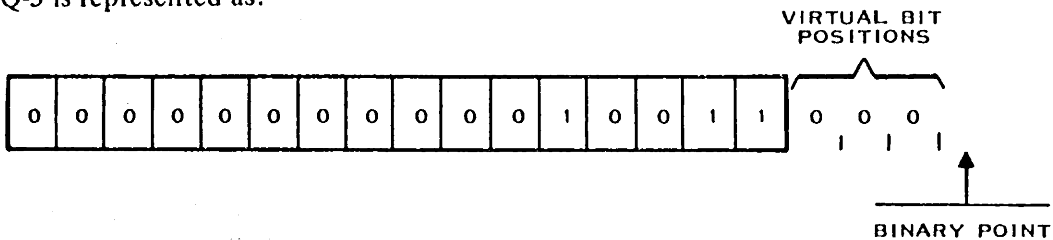
19Q0 is represented as:



19Q3 is represented as:



and 19Q-3 is represented as:



NOTE

The binary point is not stored internally as a part of the integer portion. Its position is maintained separately by the FORTRAN compiler.

2.4.6.3 Examples.

512Q0
256Q-1
4096Q+3
} All of these have a decimal value of 512.

13Q0 is equivalent to a decimal value of 13.
13Q4 is equivalent to a decimal value of 0.8125.
13Q-2 is equivalent to a decimal value of 52.

See Appendix J for a more detailed discussion of fixed point integers use.

2.5 HOLLERITH FIELD

A Hollerith field is defined as a string of characters. The characters are represented internally in standard ASCII code. Hollerith fields may contain any of the characters represented in ASCII code and are not restricted to the character set defined for FORTRAN programming. The blank character is significant within the Hollerith field. Hollerith fields do not have an associated type declaraction in the sense of paragraph 2.3. Hollerith data can be assigned to a variable by using a DATA statement if such an assignment is required. However, since Hollerith field codes occupy eight bits per character, only two characters can be assigned to an integer variable and only four characters can be assigned to a real number variable.

A Hollerith constant may be written in an assignment statement (IA = 'BC'), in the argument list of a CALL statement (Section IV) or in a DATA (Section VI) or a FORMAT (Section V) statement. The constant is written in one of two forms. The first form appears as:

$nHc_1c_2c_3...c_n$

In this format, the letter n specifies the number of constants to be used, the letter H designates that the constants are character data, and the letters c_1 through c_n represent the input characters.

The second form encloses the desired character field in apostrophes to indicate that the data represents character constants rather than numerical data. For this form, an apostrophe in the character field must be represented by two adjacent apostrophes. Therefore, the statement:

```
10 FORMAT ('DON''T')
```

represents the literal statement

```
DON'T
```

As an illustration of the two forms, note that these Hollerith constants are equivalent:

```
28HOUTPUT NO. 3 ISN'T MORE THAN

'OUTPUT NO. 3 ISN''T MORE THAN'
```

2.6 DIMENSIONED VARIABLES

A programmer must often perform calculations using large sets of related data. It is generally convenient to handle such data as a unit, and FORTRAN allows a single variable name to be assigned to such a set of data. To be used in a program, the set of data consists of a specified number of individual data items. The discussion of arrays, subscripts, and the DIMENSION statement in the following paragraphs describes the handling of grouped and related data in FORTRAN.

2.6.1 ARRAYS AND SUBSCRIPTS. A set of related data is called an *array*, reflecting the fact that the data can be arranged into lists, or tables of rows and columns. Table 2-1 shows a group of data presented in a two-dimensional array. Each number in the table represents the average precipitation in inches of a particular city during a particular month, and is an element of the array. If the array is given the variable name PRECIP, any element in the array can be identified by a row number and a column number following the name, for example PRECIP (3,5) = 4.8, the average precipitation in Dallas in May. The numbers 3 and 5 are called *subscripts*. The number of subscripts following the variable name is the same as the number of dimensions in the array. Commas separate the subscripts.

Table 2-1. Example of an Array: Precipitation in Selected United States Cities

City	Month											
	(1) Jan.	(2) Feb.	(3) Mar.	(4) Apr.	(5) May	(6) June	(7) July	(8) Aug.	(9) Sept.	(10) Oct.	(11) Nov.	(12) Dec.
(1) Atlanta	4.4	4.5	5.4	4.5	3.2	3.8	4.7	3.6	3.3	2.4	3.0	4.4
(2) Chicago	1.9	1.6	2.7	3.0	3.7	4.1	3.4	3.2	2.7	2.8	2.2	1.9
(3) Dallas	2.3	2.6	2.9	4.0	4.8	3.2	1.9	1.9	2.8	2.7	2.7	2.7
(4) Los Angeles	3.1	3.3	2.3	1.2	0.2	0.1	0.0	0.0	0.2	0.4	1.1	2.9
(5) New York	3.3	2.8	4.0	3.4	3.7	3.3	3.7	4.4	3.9	3.1	3.4	3.3
(6) Seattle	5.7	4.2	3.8	2.4	1.7	1.6	0.8	1.0	2.1	4.0	5.4	6.3

Any element in the array in table 2-1 could be denoted as PRECIP (I,J), where I equals an integer value from 1 to 6 for the cities and J equals an integer value from 1 to 12 for the months. Arrays are often used as described above in programs where a computation is to be performed on each element in turn.

An example of a one-dimensional array is the data for the average precipitation in Atlanta in each month (refer to table 2-1). If this array is named P1, it has as its elements P1(1) = 4.4, P1(2) = 4.5, and so forth through P1(12) = 4.4.

The number of subscripts permissible in a 990 FORTRAN array is limited to three. A sufficiently large area of storage must be available to accommodate the array.

All elements in an array must be the same data type, where the data type is one of the six described in paragraph 2.4.

2.6.2 STORAGE OF ARRAY ELEMENTS. Arrays are stored in storage locations in the following manner. The first subscript is incremented through its possible values; then the second subscript is incremented by one and the first subscript is again incremented through its possible values. When the second subscript has been incremented through its possible values in this manner, the third subscript is incremented by one and the first and second subscripts are again incremented through their ranges of values.

For example, the 3 by 3 by 3 array variable A(I,J,K) would be stored in this order:

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

The identifiers for arrays follow the same rules as identifiers for undimensioned variables. For example, these are valid array names:

ARRAY
E
ST64A

These are not valid array names:

2X2	Identifier may not begin with a numeral.
RUN#	Characters other than letters and numerals may not appear in an identifier.

2.6.3 DIMENSION STATEMENT. The DIMENSION statement is used to declare identifiers to be array identifiers and to specify the number and bounds of the array subscripts. The information supplied in a DIMENSION statement is required for the allocation of memory for arrays. Any number of arrays may be declared in a single DIMENSION statement.

Form:

DIMENSION $S_1,S_2,\ldots,S_k$

where S is an array specification.

Each array variable appearing in the program must represent an element of an array declared in a DIMENSION statement, unless the dimension information is given in another statement. When the dimension information is provided in a COMMON or type declaration statement, it may not appear in a DIMENSION statement.

Each array specification gives the array identifier and the maximum values that each of its subscripts may assume, thus:

identifier(max_1[,max_2[,max_3]])

The maxima must be integers. An array may have one, two or three dimensions. The dimension information for an array must be given before the occurrence of that array variable in the program. Subscripts must not exceed their maximum values declared in the DIMENSION statement.

NOTE

> Check that the program does not reference an array beyond its dimension to avoid affecting adjoining variables.

For example, the statement

```
DIMENSION EDGE(10,8)
```

specifies EDGE to be a two-dimensional array, the first subscript of which may vary from 1 to 10 inclusive, and the second from 1 to 8 inclusive. Zero and negative subscripts are not permitted.

Example:

```
DIMENSION PLACE(3,3,3),HI(2,4),K(256)
```

Arrays may also be declared in the COMMON or type declaration statements in the same way:

```
COMMON X(10,4),Y,Z
INTEGER A(7,32),B
DOUBLE PRECISION K(6,10)
```

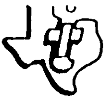
Note that each element of an integer or logical array requires one word of storage. Each element of a real array requires two words of storage, double-precision require four words of storage, and complex require four words of storage.

Within a subprogram, array specifications may use integer variables instead of constants, provided that the array name and variable dimensions are dummy arguments of the subprogram. The actual array name and values of the dummy variables are given by the calling program when the subprogram is called. Quantities needed for reference to the array are evaluated when the subprogram is entered. These quantities are not changed by any subsequent modification of the dimension variables.

Example:

```
DIMENSION BETA(L,M,G),B(20)
```

The identifiers BETA,L,M, and G must all be dummy arguments.



2.7 COMMON AND EQUIVALENCE STATEMENTS

The COMMON and EQUIVALENCE statements provide flexibility in the naming of variables and the assignment of storage locations to them.

The COMMON statement is used to make an identifier in a program and in separate subprograms refer to the same variable. Generally, variables that have the same name but are in different programs or subprograms are distinct variables. The COMMON statement assigns these distinct variables to the same storage location.

The EQUIVALENCE statement causes two or more variables in a single program or subprogram to be placed in the same storage location. This statement can be used for these purposes:

- Two or more identifiers may be defined to have identical meanings.
- A storage location can be used to contain two or more variables that are never needed at the same time.

2.7.1 COMMON STATEMENT.

A COMMON statement has the following form:

```
COMMON block-list
```

The COMMON statement specifies that certain variables or arrays are to be stored in an area also available to other programs. By means of COMMON statements, a program and its subprograms may share a common storage area.

Example:

```
COMMON A,B(2,3),C,D(4)
```

This statement causes A to be stored in the first location of the common area, the array B to be stored in the next six locations (in the order specified in paragraph 2.6.2), then C in the following location, followed by the array D in four locations.

One COMMON statement by itself does not cause multiple assignments of an area of storage. One or more additional COMMON statements, each assigning storage in a specified order to variables, accomplishes this. For example, if one program contains the above example and a subprogram contains this statement:

```
COMMON G(6),F,E(2,2),H
```

then the common area shared by the two programs or subprograms will contain the following data. Each rectangle represents a storage location.

A	G(1)
B(1,1)	G(2)
B(2,1)	G(3)
B(1,2)	G(4)
B(2,2)	G(5)
B(1,3)	G(6)
B(2,3)	F
C	E(1,1)
D(1)	E(2,1)
D(2)	E(1,2)
D(3)	E(2,2)
D(4)	H

The preceding illustration is an example of *blank common*, where all COMMON statements refer to a single unnamed common area of storage. It is possible to have a number of distinct common areas or blocks, each named with an identifier. A named common block is known as *labeled common*.

The common area may be divided into separate blocks identified by block names. A block is specified in this way:

/identifier/$identifier_1$,$identifier_2$, . . . ,$identifier_n$

The identifier enclosed in slashes is the block name. The subscripted identifiers which follow are the names of the variables or arrays assigned to the block. These elements are placed in the block in the order in which they appear in the block specification.

Quantities placed in a common block by means of equivalences (refer to paragraph 2.7.2) may cause the end of the common block to be extended. For example, the statements

```
COMMON/R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE(A,Y)
```

cause the common block R to extend from X to A(4), arranged as follows:

X	
Y	A(1)
Z	A(2)
	A(3)
	A(4)

Equivalence which causes extension of the start of a common block is not allowed. For example, the sequence

```
COMMON/R/X,Y,Z
DIMENSION A(4)
EQUIVALENCE(X,A(3))
```

is not permitted since it requires block R to be arranged

	A(1)
	A(2)
X	A(3)
Y	A(4)
Z	

A(1) and A(2) extend the start of block R.

Redundant COMMON entries are not allowed. For example, the following is invalid:

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

A variable written with subscripting information in a COMMON or type statement must not be mentioned in a DIMENSION statement. A variable in a DIMENSION statement, however, may be mentioned without subscript dimensions in a COMMON or type statement.

The block-list of the COMMON statement consists of a sequence of one or more block specifications. For example, the statement:

```
COMMON/R/X,Y,T/C/U,V,W,Z
```

indicates that the elements X, Y, and T, in that order, are to be placed in block R and that U, V, W, and Z are to be placed in block C.

Block entries are concatenated throughout the program unit, beginning with the first COMMON statement. For example, the statements:

```
COMMON/D/ALPHA/R/A,B/C/S
COMMON/C/X,Y/R/U,V,W
```

have the same effect as the statement

```
COMMON/D/ALPHA/R/A,B,U,V,W/C/S,X,Y
```

One block of common storage may be left unlabeled, in effect making it blank common. Blank common is indicated by two consecutive slashes. For instance,

```
COMMON/R/X,Y//B,C,D
```

indicates that B, C, and D are placed in blank common.

The slashes may be omitted when blank common is the first block of the statement.

```
COMMON B,C,D
```

Array names appearing in COMMON statements may have dimension information appended, as in a DIMENSION statement. For example:

```
COMMON ALPHA,T(15,10,5),GAMMA
```

specifies the dimensions of the array T while entering T in blank common.

NOTE

Dummy arguments for SUBROUTINE or FUNCTION statements cannot appear in COMMON statements.

A single COMMON statement may contain variable names, array names, and dimensioned array names (but not array elements). For example, the following are valid:

```
DIMENSION B(5,15)
COMMON A,B,C(9,9,9)
```

Variables or arrays that appear in the main program or a subprogram may be made to share the same storage locations with variables or arrays in other subprograms, by use of the COMMON statement. For example, if one program contains the statement:

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

and a second program contains the statement:

```
COMMON LIST
```

The variable names TABLE and LIST refer to the same storage locations.

If the main program contains the statement:

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

and a subprogram contains the statement:

```
COMMON X,Y,Z,XX,YY,ZZ
```

and A, B, and C are equal in length to X, Y, and Z, respectively, then A and X refer to the same storage locations, as do B and Y, and C and Z. XX, YY and ZZ are unique; that is, they do not share a storage location with any other variables.

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

A dummy variable can be used in a COMMON statement to establish shared locations for variables that would otherwise occupy different locations. For example, the variable Z of the previous example will share storage with S if the following statement is used:

```
COMMON Q,R,S
```

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

2.7.2 EQUIVALENCE STATEMENT. The EQUIVALENCE statement allows more than one identifier to represent the same quantity.

Form:

EQUIVALENCE $(r_1,r_2,\ldots,r_n),(s_1,s_2,\ldots,s_m),\ldots,(t_1,t_2,\ldots,t_k)$

where r, s, and t represent the equivalenced variables, and n, m, and k represent the number of variables equivalenced. Variables r_1 through r_n are equivalent; that is, they represent the same quantity. Variables s_1 through s_m all represent a second quantity.

The references of an EQUIVALENCE statement may be variables or array identifiers or array element references. The subscripts of an array element must be integer constants. The number of subscripts must be equal to the array dimension or must be one.

NOTE

Conversion to single subscripts is permitted only in EQUIVALENCE statements.

Since entire arrays are shifted to satisfy the equivalence, only the relative positions of the references are important. In the following example

```
EQUIVALENCE (BETA(1),ALPHA(7))
```

or

```
EQUIVALENCE (BETA(2),ALPHA(8))
```

accomplish the same result.

Note that the relation of equivalence is transitive, e.g., the two statements

```
EQUIVALENCE (A,B),(B,C)
EQUIVALENCE (A,B,C)
```

have the same effect. Redundant equivalence statements are allowed:

```
EQUIVALENCE (X,Y)
EQUIVALENCE (Y,X)
```

2.7.3 EQUIVALENCE AND COMMON INTERACTION. Identifiers may appear in both COMMON and EQUIVALENCE statements, provided no two quantities in common are set equivalent to one another.

2.8 FUNCTION REFERENCE

A *mathematical function* is a quantity or group of quantities that are defined in terms of relations between or operations upon some other quantity or quantities over a specified range of values. The value of the function depends on one or more independent variables. These independent variables are called *arguments.*

An example of a function is:

$$f(x) = x^2 + 2x + 1$$

The function value f(x) is defined in terms of the single argument x. For a given argument value such as x = 3, there is an associated function value, in this case f(x) = 16.

In a program, an arithmetic function is a subprogram which acts upon a number of arguments and produces a single numerical quantity as the function value. Function references are denoted by the identifier which names the function, followed by an argument list enclosed in parentheses:

identifier (argument$_1$, argument$_2$, . . . , argument$_n$)

In this case, there are n arguments. There must be at least one. An argument may be an expression, an array identifier, or a subprogram identifier.

A function reference represents a quantity, namely the function value, and acts as a basic element. The type of the function value is given by the type of the identifier which names the function. The type of the function is independent of the types of its arguments.

Examples:

```
X = COS(THETA)
I  = IZETA(S+SQRT(S))
```

NOTE

> In these examples, a real value X is returned for the function COS, and an integer value I is returned for IZETA, in accordance with the rules for implicit declarations of variables.

References to logical functions are written in the same way as references to arithmetic functions. The identifier used to name the function must be a logical identifier. The function arguments may be expressions (logical or arithmetic), array identifiers, or subprogram identifiers.

SECTION III

OPERATORS AND EXPRESSIONS

3.1 GENERAL

Data (mathematical elements consisting of constants, variables and functions) used in FORTRAN computation may be combined or altered using mathematical operators. The operators indicate the manner in which a group of values is to be changed to obtain a new value.

A sequence of mathematical or logical elements, separated by operators and possibly by parentheses according to specific rules, is called an *expression*. The mathematical elements are constants, variables and functions. The parentheses indicate how elements are grouped within the expression.

FORTRAN expressions may be considered to belong to either of two general classes: arithmetic or logical. The mathematical data values in the expression and the types of operators that relate them determine the class of a given expression.

The six data types described in Section II may be divided into those which have numeric values and those which have logical values. Integers, real numbers, double precision numbers, fixed and complex numbers are all numeric data. The sixth type, logical data, is distinct from the others because its value is either *true* or *false* rather than a numeric quantity.

Data values may be combined or transformed by means of operators, yielding a result which is a new data value. The FORTRAN operators may be any of three types:

- Arithmetic operators specify operations on numeric quantities.
- Relational operators express a logical relation between numeric quantities. The result of the operation is a logical value.
- Logical operators specify operations on logical quantities.

The class of an expression is determined by the results of the operations within it. If the results are numeric values, the expression is *arithmetic*; if they are logical values, the expression is *logical.* Arithmetic expressions contain arithmetic operators, while relational and logical operators appear in logical expressions.

3.2 ARITHMETIC EXPRESSIONS

The following paragraphs describe the arithmetic operators and the rules for forming and evaluating arithmetic expressions.

3.2.1 ARITHMETIC OPERATIONS. There are five arithmetic operations, represented in FORTRAN by the following symbols:

+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

Two of these symbols are also used as algebraic signs to indicate positive or negative values:

+	Positive sign
-	Negative sign

These algebraic signs are handled as operators by the FORTRAN compiler. They are read as positive and negative signs rather than addition and subtraction operators if they are not immediately preceded by an arithmetic quantity.

If the precedence of operations is not given explicitly by parentheses, the order of precedence is taken to be the following:

Operator	Operation
**	Exponentiation (highest precedence)
-	Negative sign
* and /	Multiplication and division
+ and -	Addition, positive sign and subtraction (lowest precedence)

When the operators are of equal precedence, they are understood to be grouped from the left, except for exponentiation which is performed from right to left. Thus, the expression A + B + C + D is calculated as ((A + B) + C) + D. This convention takes care of such ambiguous expressions as:

```
A**B**C
X/Y/Z
```

In 990 FORTRAN, these expressions are calculated as

```
A**(B**C)
(X/Y)/Z
```

3.2.2 FORMATION OF ARITHMETIC EXPRESSIONS. An arithmetic expression consists of numerical elements, arithmetic operators and parentheses. It has a single value which is the result of the calculations specified by the quantities and operators in the expression.

The rules for forming arithmetic expressions are the following:

1. An expression may consist of a single element (constant, variable, or function reference):

   ```
   2.71828
   Z(N)
   TAN(THETA)
   ```

2. Compound expressions may be formed by using operators to combine basic elements:

```
X+3
TOTAL/POINTS
TAN(PI*M)
```

3. Any expression may be enclosed in parentheses and considered to be a single element:

```
(X+Y)/2
(ZETA)
COS (SIN(PI*M)+X)
```

4. Expressions may be preceded by a positive sign or negative sign:

```
+X
-(ALPHA*BETA)
-SQRT (-GAMMA)
```

5. No two operators may appear in sequence. For instance:

```
X*-Y
```

is improper. Use of parentheses yields the correct form:

```
X*(-Y)
```

6. The precedence of operations, if not given explicitly with parentheses, is according to the rules in paragraph 3.2.1. For example, the expression

```
A*B+C/D**E
```

is taken to be

```
(A*B)+(C/(D**E))
```

7. Sequences of operations are evaluated according to the rules in paragraphs 3.2.1 and 3.2.3.

NOTE

Parentheses may not be used to imply multiplication. The asterisk arithmetic operator must always be used for this purpose. Therefore, the algebraic expression:

$(a\ b)\ (-c^{d})$

must be written as:

```
(A*B) * (-C**D)
```

NOTE

Hexadecimal constants may be used in the algebraic expressions. For example:

```
I = >73C + 4
```

3.2.3 EVALUATION OF ARITHMETIC EXPRESSIONS. The value of an arithmetic expression may be of the integer, fixed, real, double-precision or complex type. If more than one of these types appears in an expression, the expression's data type is that of the highest rank of any element in the expression.

The arithmetic data types are ranked as follows:

Complex (highest rank)

Double precision

Real

Fixed

Extended Integer

Integer (lowest rank)

The type of a subscripted variable, however, is determined by the type of the variable identifier.

Each operation within an expression is evaluated in the type of the operation's highest ranking operand. Thus, the evaluation of an expression is not changed to a higher rank until necessary. For example, if I and J are integers, R is REAL*4, DP is a REAL*8 number and C is a complex variable, then the expression:

(I/J + R)*DP*C

is evaluated as:

CMPLX((DBLE(FLOAT(I/J)+R)*DP),0)*C

where:

COMPLX	function that converts real → complex
DBLE	function that converts real → double
FLOAT	function that converts integer → real

Integer and fixed expressions are evaluated using binary integer arithmetic throughout, giving an integer value as the result. In integer arithmetic, fractional parts arising in division are truncated, not rounded. For example:

7/3 yields 2
6/7 yields 0

All other calculations use binary floating point arithmetic.

Conversions to higher rank are performed as follows:

1. An integer quantity becomes the integer part of a fixed or real quantity. The fractional part is zero.

2. A fixed quantity is converted to real such that the part to the left of the binary point is the integer part and the part to the right of the binary point is the fractional part.

3. A real quantity becomes the most significant part of a double precision real quantity. The least significant part is zero.

4. A real quantity becomes the real part of a complex quantity. The imaginary part is zero.

5. A double precision quantity is converted to single precision and becomes the real part of a complex quantity. The imaginary part is zero.

In exponentiation (**), the types of the base and exponent are restricted as follows:

1. Complex exponents are not allowed.

2. Only integer exponents may be used with complex bases.

3. Fixed bases are not allowed.

In exponentiation (**), the evaluation is as follows:

1. If the base is exponentiated by an integer expression, the result is the same type as the base.

2. An INTEGER*2 or INTEGER*4 expression may be exponentiated by an INTEGER*2 or INTEGER*4 expression. The result of two INTEGER*2 expressions is INTEGER*2; otherwise, the result is INTEGER*4.

3. A real or double-precision base may have a real or double-precision exponent; only if both arithmetic expressions are real, will the result be real.

4. With a complex base and integer exponent, the result is complex.

Operations are performed by evaluating exponentiation first, then negative signs, then multiplication and division, then addition, subtraction and positive signs. Operations of the same priority are evaluated left to right, except for exponentiation. The following example illustrates how this process is carried out for the five arithmetic operations:

```
2**3/4+5*6-7
  8 /4+5*6-7

    2 +5*6-7
    2 + 30-7

      32  -7

        25
```

Expressions within parentheses are evaluated first. Nested groups of parentheses are evaluated by starting with the innermost sets of parentheses and working outward. Nesting of parentheses to any level is permitted in FORTRAN. The following example illustrates how this type of expression is evaluated:

```
4*(5-((17+37)/3**(1+2)))+((21-9)/(10-6)+13)
4*(5-( 54   /3**(1+2)))+((21-9)/(10-6)+13)
4*(5-( 54   /3**  3   ))+((21-9)/(10-6)+13)
4*(5-( 54   /   27    ))+((21-9)/(10-6)+13)
4*(5-      2           )+((21-9)/(10-6)+13)
4*        3             +((21-9)/(10-6)+13)
          12            +((21-9)/(10-6)+13)
          12            +(  12  /(10-6)+13)
          12            +(  12  /  4   +13)
          12            +(       3     +13)
          12            +        16
                        28
```

When several variables or numbers in sequence are to be multiplied and/or divided, they are calculated as if grouped from the left. For example, these two are equivalent:

```
A*B/C*D
((A*B)/C)*D
```

Similarly, a string of variables or numbers to be added and/or subtracted are calculated as if grouped from the left. These two are also equivalent:

```
A-B+C-D
((A-B)+C)-D
```

3.3 LOGICAL EXPRESSIONS

Four basic elements are used in FORTRAN logical computations: constants, variables, functional references, and relations. All of these basic elements represent logical quantities. Logical identifiers are written in the same way as numerical identifiers; however, they must always be explicitly declared to be of the logical type.

A logical quantity may have either of two values: *true* or *false*. Logical quantities occupy one word.

The following paragraphs discuss relational operations, logical operations, and the formation and evaluation of logical expressions.

3.3.1 RELATIONAL OPERATIONS. Relations are constructed from arithmetic expressions of the integer, fixed, real or double-precision type. These relations make use of the relational operators:

.GT.	Greater than
.GE.	Greater than or equal to
.LT.	Less than

.LE. Less than or equal to

.EQ. Equal to

.NE. Not equal to

The enclosing periods are part of the operator and must be present.

Complex operands are allowed for .EQ. and .NE. only. Relational operators have lower precedence than arithmetic operators. Two expressions of integer, fixed, real, or double-precision type, separated by a relational operator, form a relation. For example:

```
X+2.LE.3*Y
```

is a relation. The entire relation constitutes a basic logical element.

The value of such an element is .TRUE. if the relation expressed is .TRUE.; otherwise, the value is .FALSE.. In the example above, the element has the value .TRUE. if X is 2 and Y is 2, and the value .FALSE. if X is 2 and Y is 1.

Examples of valid relational expressions are:

```
Q1 .GT. Q2
Q3**3 .EQ. (6*Q4+2)
0.4*FLN .LE. 1.83D4
```

Q1, Q2, Q3, Q4, and FLN are arithmetic quantities. In the following examples, A is an arithmetic variable and S, T1 and T2 are logical variables. Examples of illegal relational expressions are:

A .GT. (4.345, 7.614)	Complex numbers are illegal in logical expressions except .EQ. and .NE..
S .GE. (T1 + T2)	Logical quantities may not be joined by relational operators.

3.3.2 LOGICAL OPERATIONS. There are three logical operators, which are arranged according to precedence in the evaluation of an expression:

.NOT. Logical *not* (highest precedence)

.AND. Logical *and*

.OR. Logical *or* (lowest precedence)

The enclosing periods are part of the operators and must be present.

The logical operations have specific meanings, as follows. The expression .NOT.P is true if P is false and false if P is true. The expression P.AND.Q is true if P and Q are both true; otherwise, it is false. The expression P.OR.Q is false if P and Q are both false; otherwise, it is true. In a FORTRAN program, P and Q would have been explicitly defined as logical variables.

3.3.3 **FORMATION OF LOGICAL EXPRESSIONS.** Logical expressions are formed according to the following rules:

1. A logical expression may consist of a single logical element. For example:

```
.TRUE.
BOOL(N)
X.GE.3.14159
```

2. Single elements may be combined through use of the logical operators .AND. and .OR. to form compound expressions, such as:

```
TVAL.AND.INDEX
BOOL(M).OR.K.EQ.LIMIT
```

3. Any logical expression may be enclosed in parentheses and regarded as an element:

```
(T.OR.S).AND.(R.OR.Q)
(BOOL(M))
PARITY.OR.((2.GT.Y.OR.X.GE.Y).AND.NEVER)
```

4. Any logical expression may be preceded by the operator .NOT. as in:

```
.NOT.T
.NOT.X+7.GT.Y+Z
BOOL(K).AND..NOT.(TVAL.OR.R)
```

The only situation in which two logical operators may occur in sequence is when the second operator is .NOT.. An example is:

```
P.AND..NOT.Q
```

5. Expressions whose values are either *true* or *false* may be combined using logical operators.

3.3.4 **EVALUATION OF LOGICAL EXPRESSIONS.** When logical expressions are evaluated, the relational operations have precedence over the logical operations; in other words, the relational operations are evaluated before the logical operations.

Among the logical operators, .NOT. is evaluated first; then .AND.; .OR. is evaluated last. Therefore, the expression

```
T.AND..NOT.S.OR..NOT.P.AND.R
```

is interpreted

```
(T.AND.(.NOT.S)).OR.((.NOT.P).AND.R)
```

In this case, .NOT.S and .NOT.P are evaluated first and second, respectively, followed by the two expressions with .AND., and finally .OR..

3.4 SUMMARY OF RULES FOR EXPRESSIONS

The following paragraphs summarize the hierarchy of operator precedence and provide some general comments about permissible arithmetic and logical statements.

3.4.1 OPERATOR PRECEDENCE. The hierarchy of precedence of arithmetic and logical operators in FORTRAN is summarized in table 3-1. The first item in the table has the highest precedence if functions and expressions within parentheses have been evaluated. Note that arithmetic operators are evaluated before relational and logical operators. All operators of the same precedence are evaluated from left to right.

For example, the logical expression

```
.NOT.ZETA**2+Y*MASS.GT.K-2.OR.PARITY.AND.X.EQ.Y
```

is interpreted

```
(.NOT.(((ZETA**2)+(Y*MASS)).GT.(K-2))).OR.(PARITY.AND.(X.EQ.Y))
```

Table 3-1. Precedence of Operators

Operator	Operation
**	Exponentiation
-	Negative sign
* and /	Multiplication and division
+ and -	Addition, positive sign and subtraction
.GT.	
.GE.	
.LT.	Relational operations
.LE.	
.EQ.	
.NE.	
.NOT.	Logical not
.AND.	Logical and
.OR.	Logical or

3.4.2 ASSIGNMENT STATEMENTS. Arithmetic and logical FORTRAN assignment statements have the form:

variable = expression

where *expression* may be any arithmetic or logical expression.

The statement defines the value of the variable by evaluating the expression to the right of the equal sign.

Note that an assignment statement is not a mathematical equation, for the "=" sign is not used in its usual mathematical sense. In a FORTRAN statement, the "=" means *is replaced by the value of* rather than *is equal to the value of*. For example, the statement

```
N = N + 1
```

is interpreted to mean "N is replaced by the value N+1"; i.e., one is added to the value of N. N is *assigned* a new value which is greater than its previous value by 1.

Examples:

```
Y = 2*Y
P = .TRUE.
X(N) = N*ZETA(ALPHA*M/PI)
```

If the type of the expression on the right of the assignment statement differs from the type of the variable on the left, the expression on the right takes on the type of the variable on the left. Table 3-2 summarizes the conversion for all combinations of expressions and variables.

Assignment statements of the form:

variable = Hollerith constant

are special cases. The character string represented by the Hollerith constant is transferred to the variable without any type conversion. The string is left-justified in the variable with blanks added on the right as necessary to fill the variable. The number of characters cannot be greater than the number of bytes in the variable.

Example:

```
X = 'AB'
```

The example X = 'AB' implies that the four bytes represented by X (REAL*4) appear as

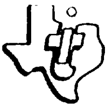
A B b̶ b̶

Table 3-2. Assignment Statement Type Conversion

Type of Variable (Left Side of Statement)	Type of Expression (Right side of Statement)	Conversion
INTEGER*2	INTEGER*2	None.
INTEGER*2	INTEGER*4	Take least significant word.
INTEGER*2	FIXED	Take least significant 16 bits of the integer part.
INTEGER*2	REAL	Truncate fractional part; convert to integer form.
INTEGER*2	DOUBLE PRECISION	Truncate fractional part; convert to integer form.
INTEGER*2	COMPLEX	Truncate fraction part of real part; convert to integer form.
INTEGER*4	INTEGER*2	Extend sign into most significant of two words.
INTEGER*4	INTEGER*4	None.
INTEGER*4	FIXED	Take least significant 16 bits of the integer part.
INTEGER*4	REAL	Truncate fractional part; convert to integer form.
INTEGER*4	DOUBLE PRECISION	Truncate fractional part; convert to integer form.
INTEGER*4	COMPLEX	Truncate fractional part of real part; convert to integer form.
FIXED	INTEGER*2	Integer becomes integral part; shift according to scale.

Table 3-2. Assignment Statement Type Conversion (Continued)

Type of Variable (Left Side of Statement)	Type of Expression (Right Side of Statement)	Conversion
FIXED	INTEGER*4	Integer becomes integral part; shift according to scale.
FIXED	FIXED	None, except possible shift.
FIXED	REAL	Convert both integral and fractional parts to integer form; shift according to scale.
FIXED	DOUBLE PRECISION	Convert both integral and fractional parts to integer form; shift according to scale.
FIXED	COMPLEX	Convert both integral and fractional parts of real part to integer form; shift according to scale.
REAL	INTEGER*2	Convert to floating-point form.
REAL	INTEGER*4	Convert to floating-point form.
REAL	FIXED	Convert to floating-point form.
REAL	REAL	None.
REAL	DOUBLE PRECISION	Take first two words of extended floating point.
REAL	COMPLEX	Take real part.
DOUBLE PRECISION	INTEGER*2	Convert to extended floating-point form.
DOUBLE PRECISION	INTEGER*4	Convert to extended floating-point form.
DOUBLE PRECISION	FIXED	Convert to extended floating-point form.
DOUBLE PRECISION	REAL	Extend single floating point to extended form.
DOUBLE PRECISION	DOUBLE PRECISION	None.
DOUBLE PRECISION	COMPLEX	Extend real part to extended floating-point form.
COMPLEX	INTEGER*2	Convert to real form; assign to real part; assign zero imaginary.
COMPLEX	INTEGER*4	Convert to real form; assign to real part; assign zero imaginary.
COMPLEX	FIXED	Convert to real form; assign to real part; assign zero imaginary.
COMPLEX	REAL	Assign to real part; assign zero imaginary.
COMPLEX	DOUBLE PRECISION	Take first two words of extended floating point; assign to real part; assign zero imaginary.
COMPLEX	COMPLEX	None.



SECTION IV

CONTROL

4.1 GENERAL STATEMENTS

Statements in FORTRAN programs are normally executed sequentially. This sequence may be altered by the use of control statements which allow the programmer to make decisions during the course of his program and control the order of execution of the program.

FORTRAN statements may be given numbers to be referenced by control statements. A statement number is written as an unsigned, nonzero integer of five digits or less. Leading zeros are ignored. Although statement numbers are written as integers, they represent labels rather than numerical quantities. Statement numbers are used for program control, not numerical calculation. Statement numbers must be unique within any separately compiled program unit. For example, no two statements may have the same number in a program which will be called ANY in this example; however, ANY may contain a statement numbered j and may call a subprogram which contains a statement numbered j.

4.2 GO TO STATEMENT

The following paragraphs discuss the various forms of the GO TO statement and a statement used in conjunction with it, the ASSIGN statement. The statements included are:

- Unconditional GO TO statement
- Computed GO TO statement
- ASSIGN statement
- Assigned GO TO statement

4.2.1 UNCONDITIONAL GO TO STATEMENT.

Form:

```
GO TO j
```

This statement transfers control from one point of a program to another by indicating the number of the statement to be executed next, j. Each time the GO TO statement is reached, the program transfers control to statement number j and proceeds sequentially from there until another control statement is reached.

Example:

```
     GO TO 10
 8   A=B+C
10   B=B+1
```

Statement 10 is executed immediately after the GO TO statement.

4.2.2 COMPUTED GO TO STATEMENT.

Form:

GO TO $(j_1,j_2,j_3,\ldots,j_n)$[,]i

This statement allows control to be transferred to any one of n different statements, where $j_1, j_2,\ldots,j_n$ are the statement numbers, and i is a nonsubscripted integer variable which has a value in the range 1 through n. The computed GO TO causes control to be transferred to the statement whose label is j_k, where k is the value of the integer variable i. If i is outside the range 1 through n, then the next sequential statement following the GO TO statement is executed. The comma separating i from the statement list is optional.

Example:

GO TO (89,14,92,21),ICE

Statement 89 is executed next if the value of ICE is 1. If the value of ICE is 3, statement 92 is executed.

4.2.3 ASSIGN AND ASSIGNED GO TO STATEMENTS.

Forms:

1. ASSIGN i TO variable

   .
   .
   .

   GO TO variable

2. GO TO variable, $(n_1,n_2,n_3,\ldots,n_k)$

Variable is a nonsubscripted integer variable, i is an executable statement number, and $n_1,n_2,\ldots,n_k$ are executable statement numbers.

The second form of the assigned GO TO is allowed for coding compatibility with other versions of FORTRAN only. It is important to note that the list of statement numbers is not used to limit transfer of control, i.e., the statement number does not have to be included in the list for the program to work. However, the list may be used as a handy reference of all possible transfer points, and if a listed statement number is not used as a label somewhere in the program, an error message is printed by the compiler.

This statement transfers control to the statement whose number was assigned to the variable. The assignment must take place in a previously executed ASSIGN statement.

The variable is a control variable, having a label as a value, not a numerical quantity. At the time of execution of an assigned GO TO statement, the current value of *variable* must have been defined by the previous execution of an ASSIGN statement. The value of the integer variable is not the integer statement number; ASSIGN 10 TO I is not the same as I =10.

Example 1:

```
      ASSIGN 40 TO NERROR
         .
         .
         .
      GO TO NERROR
         .
         .
         .
   40 A=A+B
```

In example 1, control is transferred to the statement numbered 40.

Example 2:

```
      GO TO N, (10,25,8)
```

If the current assignment of the integer variable N is statement number 8, then the statement numbered 8 is executed. If the current assignment of N is statement number 10, then the statement numbered 10 is executed next. If N is assigned statement number 25, statement 25 is executed next. If N is assigned statement number 12, statement 12 is executed, even though it is not included in the list within parentheses.

Example 3:

```
         .
         .
         .
      ASSIGN 10 TO ITEM
         .
         .
         .
   13 GO TO ITEM,(8,12,25,50,10)
         .
         .
         .
    8 A=B+C
         .
         .
         .
   10 B=C+D
      ASSIGN 25 TO ITEM
      GO TO 13
         .
         .
         .
   25 C=E**2
```

In example 3, the first time statement 13 is executed, control is transferred to statement 10. On the second execution of statement 13, control is transferred to statement 25.

4.3 IF STATEMENT

The following paragraphs describe the two forms of the IF statement: the arithmetic IF statement and the logical IF statement.

4.3.1 ARITHMETIC IF STATEMENT.

Form:

```
IF (e) n1,n2,n3
```

The quantity e is an arithmetic expression and n_1, n_2, and n_3 are statement numbers. The simple GO TO statement causes an unconditional transfer of control to a specified statement. The transfer does not depend on any condition of the data, status of the machine, or any other changeable hardware or software condition. The unconditional GO TO by itself would permit little work to be done; it is also necessary to be able to transfer if some condition is met during program execution. The arithmetic IF statement accomplishes this. If the value of the expression within parentheses is negative, the statement having statement number n_1 is executed next; if the value of the expression is zero, statement n_2 is executed next; if the expression is positive, n_3 is executed next.

Example:

```
      IF (A(J,K)**3-B) 10,4,30
         .
         .
         .
  4   D=B+C
         .
         .
         .
 30   C=(D+C)**2
         .
         .
         .
 10   E=(F*B)/(D+1)
```

In this example, if the value of the expression (A(J,K)**3-B) is negative, statement number 10 is executed next. If the value of the expression is zero, statement number 4 is executed next. If the value of the expression is positive, statement number 30 is executed next.

4.3.2 LOGICAL IF STATEMENT.

Form:

```
IF(e)s
```

Another tool for transfer of control is the logical IF statement, where e is a logical expression and s is any other statement except another logical IF or a DO (discussed in paragraph 4.4). The most common form of logical expression in this context is one that asks a question about two arithmetic expressions. Such relational expressions are written by using the six relational operators: .GT.; .GE.; .LT.; .LE.; .EQ.; .NE.. (These operators are discussed in Section III.)

The action of the logical IF is as follows: if the logical expression is true, statement s is executed; if the logical expression is false, statement s is not executed. In either case, the next statement executed is the one following the logical IF, unless s is a GO TO or an arithmetic IF, and the expression is true.

Example 1:

```
          .
          .
          .
      IF (A.LE.0) GO TO 25
      C=D+E
          .
          .
          .
   25 W=X**Z
```

If the value of the expression (A.LE.0) is *true*, indicating that A is less than or equal to 0, the statement GO TO 25 is executed next and control is passed to statement number 25. If the value of (A.LE.0) is *false*, indicating that A is greater than 0, the statement GO TO 25 is ignored and control is passed to the next sequential instruction.

Example 2:

Assume that P and Q are logical variables.

```
          .
          .
          .
      IF (P.OR..NOT.Q) A=B
      C=B**2
```

In the first statement, if the value of (P.OR..NOT.Q) is *true*, the value of A is replaced by the value of B and the second statement is executed next. If the value of (P.OR..NOT.Q) is *false*, the statement A=B is skipped and the second statement is executed.

4.4 DO AND CONTINUE STATEMENTS

The following paragraphs describe the DO statement and a statement used in conjunction with it, the CONTINUE statement.

4.4.1 DO STATEMENT.

Forms:

DO n i=k_1,k_2
DO n i=k_1,k_2,k_3

Most of the computation and information processing involved in a program is essentially repetitive. The DO statement serves as a powerful tool in FORTRAN by facilitating the definition and control of repetitive processes. For example, the following statements:

```
    I=1
10  SQ(I)=A(I)*A(I)
    I=I+1
    IF(I.LE.50)GO TO 10
```

which calculate the squares of 50 numbers can be replaced by:

```
    DO 10 I=1,50
10  SQ(I)=A(I)*A(I)
```

where the DO statement simplifies the iterative process controlled by the first, third, and fourth statements.

In the general form of the DO statement, n is a statement number, i is a nonsubscripted integer variable, k_1, k_2, and k_3 are unsigned integer constants or nonsubscripted integer variables. All integer variables must be INTEGER*2 type. If k_3 is not stated, it is assumed to be 1. The value k_1 is called the *initial value*, k_2 is called the *test value*, and k_3 is called the *increment*.

NOTE

> Specifying a K_2 (DO loop test value) of -1 results in an infinite DO loop.

The n which follows DO is called the *range limit* and indicates that all the instructions following the DO statement, up to the including statement n, should be executed until the value of i exceeds k_2.

The variable i is referred to as the *index* and may be used as a subscript or ordinary integer variable within the DO loop. It may also be changed within the DO loop. Once outside the DO loop, the index may be redefined and used again.

The DO statement first sets the index to the initial value k_1. After the statements in the range have been executed, the value of i is incremented by the value k_3 (or 1 if k_3 is not specified), and i is then tested to see whether it is greater than the test value k_2. If it is not, all the statements in the range are executed again. If i does exceed the test value, control passes to the next statement immediately following the one labeled n. This is referred to as a normal exit from the DO statement and the value of the index i is undefined. An exit can also occur due to a transfer within the range, in which case the value of index i is still defined.

Because the test value is compared with the value of the index following the execution of the range, a DO loop is always performed at least once, even if the initial value is greater than the test value when the DO is entered.

Example 1:

```
      DO 5 I=1,20
   1       .
   2       .
   3       .
   4       .
   5  C=A*B
   6       .
```

In this example, statements 1-5 will be executed 20 times with I ranging in value from 1-20. After completion of the DO loop, statement 6 is executed.

Example 2:

```
           .
           .
           .
      DO 25 I=1,1000
  25  STOCK(I)=STOCK(I)-OUT(I)
      A=B+C
           .
           .
           .
```

In example 2, the index, I, is set to the initial value of 1. Before the second execution of statement 25, I is increased by the increment, 1, and statement 25 is again executed. After 1000 executions of the DO loop, I equals 1000. Since I is now equal to the highest value that does not exceed the test value, 1000, control passes out of the DO loop and the third statement is executed next. Note that the DO variable I is now undefined; its value is not necessarily 1000 or 1001.

Example 3:

```
           .
           .
           .
      DO 25 I=1,10,2
      J=I+K
  25  ARRAY(J)=BRAY(J)
      A=B+C
           .
           .
           .
```

In example 3, statement 25 is the end of the range of the DO loop. The DO variable, I, is set to the initial value of 1. Before the second execution of the DO loop, I is increased by the increment, 2, and the second and third statements are executed a second time. After the fifth execution of the DO loop, I equals 9. Since I is now equal to the highest value that does not exceed the test value, 10, control passes out of the DO loop and the fourth statement is executed next. Note that the DO variable I is now undefined; its value is not necessarily 9 or 11.

Programming considerations in using a DO loop are as follows:

1. The indexing parameters of a DO statement (i, k_1, k_2, k_3) may be changed (provided they are not constants) by a statement within the range of the DO loop. This is not allowed in most other versions of FORTRAN.

2. There may be other DO statements within the range of a DO statement. All statements in the range of the inner DO must be in the range of the outer DO. A set of DO statements satisfying this rule is called a nest of DOs.

Example 1:

```
      DO 50 I = 1,4                                   ] Range of
      A(I) = B(I)**2                                  ] Outer DO
      DO 50 J = 1,5              ] Range of           ]
50    C(J+1) = A(I)              ] Inner DO           ]
```

Example 2:

```
      DO 10 INDEX = L,M                               ] Range of
      N = INDEX + K                                   ] Outer DO
      DO 15 J = 1,100,2          ] Range of           ]
15    TABLE(J) = SUM(J,N)-1      ] Inner DO           ]
10    B(N) = A(N)
```

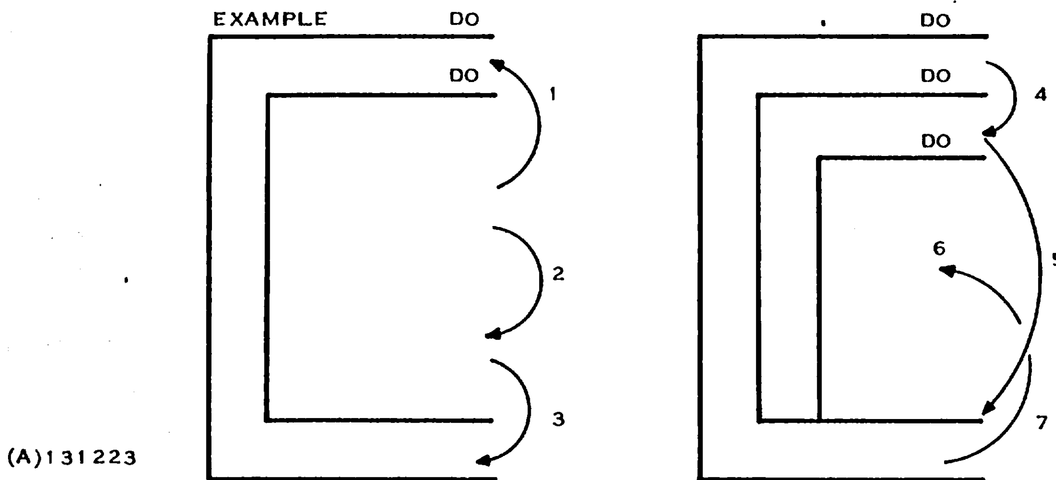
3. A transfer out of the range of any DO loop is permissible at any time. A normal exit from a DO loop makes the value of index i undefined. A transfer out of the range of the DO loop reserves the current value of the index i.

4. The extended range of a DO is defined as those statements that are executed between the transfer out of the innermost DO of a nest of DOs and the transfer back into the range of this innermost DO. The extended range must be within the program unit containing the DO statement. The following restrictions apply:

   a. Transfer into the range of a DO permitted only if such a transfer is from the extended range of the DO.

   b. The extended range of a DO statement must not contain another DO statement having an extended range when the contained (second) DO statement is within the same program unit as the first.

   Note that a statement that is the end of the range of more than one DO statement is within the innermost DO. The statement label of such a terminal statement may not be used in any GO TO or arithmetic IF statement that occurs anywhere but in the range of the most deeply contained DO with that terminal statement.



(A)131223

In the preceding example, the transfers specified by the numbers 1, 2, and 3 are permissible, whereas those specified by 4, 5, 6, and 7 are not. Transfers 1 and 3 are from a point in an inner DO to a point in a DO containing the inner DO. Transfer 2 is within the range of the innermost DO. Transfers 4 and 6, from a point in a DO to a point in a second DO contained within the first, are not allowed. Transfer 5 is from a point in a DO to a terminal statement shared with a contained DO; this transfer is not allowed. Transfer 7 is into a DO from outside its range; it also is not allowed.

5. The last statement in the range of a DO loop must be an executable statement. The statement cannot be a GO TO statement of any form, or a PAUSE, STOP, RETURN, arithmetic IF statement, another DO statement, or a logical IF statement containing the forms GO TO, PAUSE, STOP, RETURN, or arithmetic IF. When the last statement in the range of a DO loop is a logical IF, control is transferred to the next statement when the conditions satisfying that DO loop are met.

Example:

```
   DO 5 K = 1,4
5  IF (X(K).GT.Y(K)) Y(K) = X(K)
6  CONTINUE
```

Statement 5 is executed four times regardless of the outcome of the IF statement. Statement 6 is executed following the fourth execution of statement 5 regardless of the outcome of the IF statement.

6. The use of, and return from, a subprogram from within any DO loop in a nest of DOs is permitted.

7. The indexing parameters of a DO statement (i, k_1, k_2, k_3) must be of the type INTEGER*2.

4.4.2 CONTINUE STATEMENT.

Form:

```
CONTINUE
```

The CONTINUE statement causes no instructions to be executed; it serves only as a dummy statement to indicate that in the normal sequence of the program, the statement following is to be executed next.

The primary use of the CONTINUE statement is as the range limit for the DO loop. It must be used when the loop would otherwise terminate with an IF, GO TO, STOP, PAUSE, or RETURN statement. The CONTINUE also acts as a transfer point for IF and GO TO statements within the range of the DO that are intended to begin another iteration of the loop.

Example 1:

```
        .
        .
        .
      DO 30 I=1,20
    7 IF (A(I)-B(I)) 5,30,30
    5 A(I)=A(I)+1.0
      B(I)=B(I)-2.0
        .
        .
        .
      GO TO 7
   30 CONTINUE
      C=A(3)+B(7)
        .
        .
        .
```

In example 1, the CONTINUE statement is used as the last statement in the range of the DO in order to avoid ending the DO loop with the statement GO TO 7.

Example 2:

```
        .
        .
        .
      DO 30 I=1,20
      IF (A(I)-B(I))5,40,40
    5 A(I)=C(I)
      GO TO 30
   40 A(I)=0.0
   30 CONTINUE
```

In example 2, the CONTINUE statement provides a branch point enabling the programmer to bypass the execution of statement 40.

4.5 TRANSFER OF CONTROL TO SUBROUTINES

The following paragraphs describe the CALL statement and the RETURN statement.

4.5.1 CALL STATEMENT.

Form:

CALL s $(a_1,a_2,\ldots,a_n)$

The CALL statement causes a transfer of control to a SUBROUTINE subprogram (described in Section VI); s is the symbolic name of the subprogram and $a_1, a_2, ..., a_n$ are arguments that may be required by the subprogram. The symbolic name of the subprogram is not assigned a type and has no relation to the type of the arguments. If the subprogram requires no arguments, $a_1, a_2, ..., a_n$ are not required in the CALL statement.

Examples:

```
CALL MULTI (A,B,C)
CALL PRODUCE (A,10,50,P)
```

The CALL statement causes control to transfer to the first executable statement of the SUBROUTINE subprogram. The dummy variables are replaced by the values of the arguments in the CALL statement. Arguments appearing in a CALL statement may be any constant, any subscripted or nonsubscripted variable, any expression, or a subprogram name. A SUBROUTINE subprogram may use one or more of its arguments to return results to the calling program. A constant should not be used as an argument if a value is being returned to that argument by the subroutine. The following example illustrates a problem that might be encountered when a constant is used as an argument.

```
MAIN PROGRAM            SUBROUTINE SUBPROGRAM

        .
        .                SUBROUTINE MIN (N,M,I)
        .
CALL MIN (6,3,25)       M = N * I
        .
        .               RETURN
        .
20      S = 3           END
        .
        .
        .
```

Here the subroutine replaces the constant 3 with the value N * I, which is 150 in this case. This results in the variable S being assigned a value of 150 when statement 20 is executed, rather than the expected value of 3.

4.5.2 RETURN STATEMENT.

Form:

```
RETURN
```

This statement returns control from a subprogram to the calling program. Normally, the last statement executed in a subprogram is a RETURN statement. It need not be physically the last statement of the subprogram. Any number of RETURN statements may be used. RETURN may appear only in subprograms.

4.6 PAUSE, STOP AND END STATEMENTS

The following paragraphs describe the PAUSE statement, STOP statement and END statement.

4.6.1 PAUSE STATEMENT.

Forms:

PAUSE

PAUSE n

In the second form of the statement, n is an integer constant.

4.6.1.1 DX10 PAUSE Statement 2.X Releases. The PAUSE statement is included for compatibility with other versions of FORTRAN. The integer constant (or zero if the integer is omitted) is written in a runtime file by the FORTRAN language processor. The file is assigned a name consisting of:

.L (5000 + terminal id).

The contents of this file may be displayed by doing a Show File (SDS) following execution of the FORTRAN program. The file is recreated each time a program runs and the previous contents of the file are destroyed. When the integer has been written into the file, execution continues without waiting for a reply from the console. The PAUSE statement may be useful in program debugging.

4.6.1.2 DX10 PAUSE Statement 3.X Releases. The PAUSE statement is included for compatibility with other versions of FORTRAN. The integer constant (or zero if the integer is omitted) is written in the Terminal Local File (TLF) by the FORTRAN language processor.

The TLF contents are displayed on the terminal screen following execution of the FORTRAN program. The file is recreated each time a program runs, so that the previous contents of the file are destroyed. When the integer has been written into the file, execution continues without waiting for a reply from the console. This statement is sometimes useful in program debugging.

4.6.1.3 TXDS PAUSE Statement. The PAUSE statement is included for compatibility with other versions of FORTRAN. The integer constant (or zero if the integer is omitted) is written in a runtime file by the FORTRAN language processor. The contents of this file are displayed on the system console following execution of the FORTRAN program. When the integer has been written into the file, execution continues without waiting for a reply from the console. This statement is sometimes useful in program debugging.

4.6.2 STOP STATEMENT.

Forms:

STOP

STOP n

In the second form of the statement, n is an integer constant.

4.6.2.1 DX10 STOP Statement 2.X Releases. The integer constant (or zero if the integer constant is omitted) is written in a runtime file by the FORTRAN language processor. The file is created with the name:

.L (5000 + terminal id).

The contents of the file may then be displayed by doing a Show File (SDS). Program processing stops and control returns to the operating system. The file is created each time a program uses the STOP statement from a particular terminal. Previous contents of the file are destroyed.

4.6.2.2 DX10 STOP Statement 3.X Releases. The integer constant (or zero if the integer constant is omitted) is written in the TLF by the FORTRAN language processor.

The TLF contents are displayed on the terminal screen, program processing stops, and control returns to the operating system. The file is created each time a program uses the STOP statement from a particular terminal. Previous contents of the file are destroyed.

4.6.2.3 TXDS STOP Statement. The integer constant (or zero if the integer constant is omitted) is written in a runtime file by the FORTRAN language processor. The contents of this file are displayed on the system console, program processing stops, and control returns to the operating system.

4.6.3 END STATEMENT.

Form:

```
END
```

The END statement is a nonexecutable statement that defines the end of a source program or source subprogram for the compiler. Physically, it must be the last statement of each program or subprogram. If program execution reaches the END statement, the effect is exactly as if a STOP statement has been executed. The END statement must be on a single source line; continuation lines are not allowed. If the END statement has a statement number, it is equivalent to a STOP statement.

SECTION V

INPUT/OUTPUT STATEMENTS

5.1 GENERAL

The input/output (I/O) statements control the transmission of information between the computer and the peripheral I/O devices or units. I/O statements are classified as follows:

- *FORMAT Statement.* The FORMAT statement is a nonexecutable statement which specifies the conversions required between internal and external data forms. It describes how the data is arranged for input or output. Data transferred without a FORMAT statement is unformatted and reflects the data as it appears in computer memory.

- *READ and WRITE Statements.* READ and WRITE are statements which specify transmission of information between computer memory and various input/output devices.

- *Mass Storage File Input/Output Statements.* REWIND, BACKSPACE, and END FILE provide positioning and file termination for magnetic tape and disc I/O.

NOTE

All devices are opened on *first* I/O operation, *not* at initialization. It is recommended the user do an initial I/O operation, such as a REWIND, to open a device prior to the first actual I/O. This will ensure proper execution of the first actual I/O.

5.2 FORMAT STATEMENT

All formatted input or output requires the use of a data format specifying the external format of the data and the type of conversion to be used. The data format is given in a FORMAT statement or as an alphanumeric string in a data array.

Form:

FORMAT(S_1,S_2,. . . ,S_k)

where S is a data field specification as defined in the following paragraphs.

5.2.1 NUMERICAL DATA SPECIFICATIONS. Conversion of arithmetic data may be one of the following five types:

- *D field specification.* The internal form is double-precision binary floating point. The external form is double-precision decimal floating point. The specification has the form

  Dw.d

- *E field specification.* The internal form is binary floating point. The external form is decimal floating point. The specification has the form

  Ew.d

- *F field specification*. The internal form is binary floating point. The external form is decimal fixed point. The specification has the form

  Fw.d

- *G field specification*. The internal form is binary floating point. The external form is decimal fixed point or floating point. The specification has the form

  Gw.d

- *I field specification*. The internal form is binary integer. The external form is decimal integer. The specification has the form

  Iw

- *Z field specification*. The internal form is binary, the external form is hexadecimal integer. The specification has the form

  Zw

The letter D, E, F, G, I or Z designates the conversion type; w is an integer specifying the field width, which includes any preceding blanks, minus sign, decimal point, and exponent; d is an integer specifying the number of decimal places to the right of the decimal point. For example, the statement

```
FORMAT (I5,F10.2,D25.8)
```

could be used to output the line

COLUMNS:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
				3	2					-	1	7	.	6	0													.	5	9	6	2	5	4	7	6	0	+	0	3

The field width w, includes all characters in the number. If the number is too large for the field width, the field is filled with question marks.

The type of conversion used must correspond to the type of the variable in the input/output list. I conversion is used for integer or extended integer variables; E, F or G conversion is used for real variables; The D conversion is used for double-precision variables; and the Z conversion is used for any variable.

5.2.1.1 I Field Specifications. The I field specification is used to input or output an integer value. The form Iw specifies that w character positions are to be allotted to the number. The number is right justified within the w-field and unused space remains blank. If a value requires more than w positions, an error condition exists and is indicated by w question marks placed in the field. If the value is negative, a minus sign immediately precedes the leftmost digit. If the value is positive, a blank space immediately precedes the leftmost digit. A position must be reserved for the sign (either minus sign or blank space); thus, a number with a size greater than w-1 digit position results in an error condition. Blanks received as input are interpreted as zeros.

The following examples show how each of the values on the left is printed for the format specification I4:

Value Stored Internally	Appearance on Printed Line
459	459
+459	459
-360	-360
10	10
10000	????

5.2.1.2 F Field Specification. For this form of conversion, the specification is

Fw.d

The F indicates conversion between an internal real value and an external number without an exponent. The quantity w refers to the data length of the total field, and d refers to the number of places which are to appear to the right of the decimal point (the fractional portion). The total field length must provide positions for a sign, if any, a decimal point, and a digit to the left of the decimal point. The sign is printed only if the number is negative. In general, w should be greater than or equal to d+3 for output.

If a fractional portion requires more positions than are reserved by d, it is rounded by adding 5 to the digit beyond the rightmost digit to be output. If the positions reserved by d are not filled, zeros are placed from the right of the last digit to the end of the field. The portion of the field specified by w is treated in the same manner as numbers converted by the I field specification on both input and output. Blanks received as input are interpreted as zeros.

When determining the field specification, it is important to remember that a real value has only seven significant digits, and that a double precision value has 16 significant digits.

The following examples show how each of the values on the left is printed according to the specification F7.3:

Value Stored Internally	Appearance on Printed Line
32.447	32.447
-88.668	-88.668
-.33	-0.330
8.6452	8.645
-3.	-3.000
8.064	8.064
325.943	???????

Question marks are placed in all error fields.

For F field specification input, the decimal point in the data field is optional. If a decimal point is present that is inconsistent with the format specification, the actual position of the decimal point overrides that indicated in the format and governs the conversion. If no decimal point is present, space should not be reserved for it. The number is given a decimal point when converted internally; the format specification is used to place the decimal point in the correct position. For example, these two input fields yield the same result internally if the specification is F7.3:

246.135

246135

For F field specification output, fractional numbers are usually printed with a zero immediately to the left of the decimal point. If d is zero, a fractional value is printed as a sign, a zero, and a decimal point.

For example, for the specification F4.0:

-.3564 is converted to -0.

A zero value is printed with a zero preceding the decimal point.

Numbers in an E field specification format are accepted as data for F field specification input.

5.2.1.3 E Field Specification. For E-type conversion, the field specification is

Ew.d

The E indicates conversion between an internal real value and an external number with an exponent. The d indicates the fractional portion. For output, the minimum requirement is the d field, a decimal point, a digit to the left of the decimal point, a place for a sign (or blank), and four places for the exponential portion. Each of these must have an allotted space on output. If w≤d+6 an output error condition will occur. For input, space for all of these positions is not necessary. In general, the relationship w≥d+7 should be sufficient.

The exponent is a signed or unsigned one- or two-digit integer constant preceded by the letter E and within the range -78≤e≤75. The integer indicates the power of ten which is used to multiply the number to obtain its true internal value. Blanks received as input are interpreted as zeros.

The following examples show how each of the values on the left is printed according to the specification E11.4.

Value Stored Internally	Appearance on Printed Line
4381.	0.4381E 04
.00000000004381	0.4381E-10
-262610	-0.2626E 06

Question marks are placed in all error fields. For example, an attempt to print the above values in E10.4 (w=d+6) will result in a string of 10 question marks on the printed line.

For input, the start of the exponential field may be indicated by a + or - sign rather than an E. The following numerical fields would each yield the same result:

1.25E04,1.25E4,1.25E+04,1.25E+4,1.25+04,1.25+4

E conversion will accept F field specification data as input. For example, an input field of

12500.

produces the same value internally as the examples immediately above. If the decimal point is omitted from the input field, the format specification will place the decimal point in the correct position.

5.2.1.4 **G Field Specification.** The G field specification is used for numbers that may be represented in either fixed or floating point. The format editor automatically selects one of the two representations for data output. The numeric field descriptor Gw.d indicates that the external field occupies w positions with d significant digits. The value of the list item appears, or is to appear, internally as a real data item. Input processing is the same as for the F field specification.

The method of representation in the external output string is a function of the magnitude of the real data item being converted. Let N be the magnitude of the internal data item. The following tabulation exhibits a correspondence between N and the equivalent method of conversion that will be effected:

Magnitude of Data Item	Equivalent Conversion Effected
$0.1 \leq N < 1$	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
Otherwise ($N < 0.1$; $N \geq 10^{d}$)	sEw.d

where s is the current scale factor (refer to paragraph 5.4.1.7), and E field conversion is used. Note that the effect of the scale factor is suspended unless the magnitude of the data item to be converted is outside of the range that permits effective use of F field conversion.

When used for data input, values read into variables using a G field specification are converted according to the type of the corresponding variable in the input list.

5.2.1.5 **D Field Specification.** In D field conversion, the internal value must be double precision. The numeric field descriptor Dw.d indicates that the external field occupies w positions, the fractional part of which consists of d digits.

The basic form of the external input field is the same as for real conversions. Any of the following are valid inputs in a D field specification, and will produce the same result internally:

3D1

3E1

30.

The external output field is the same as for E conversion, except that the character D replaces the character E in the exponent.

5.2.1.6 **Z Field Specification.** The Z field specification is used for the transmission of hexadecimal data. The input fields are scanned right to left with any leading, imbedded, or trailing blank treated as zeroes. In internal storage, 2 hexadecimal digits occupy one byte of storage. If the input data is an odd number of digits padding occurs on the left in storage. When the input data is too large

for the storage area, truncation occurs at the high-order digits. With output, if the number of characters in storage is less than w, the leftmost positions are filled with blanks. If storage is greater than w, the leftmost digits are truncated.

5.2.1.7 **Scale Factors.** Scale factors may be specified for D, E, and F field specifications.

Form:

nPS

where n is a signed or unsigned integer specifying the scale factor, P is the identifying character, and S is a data field specification. The factor changes the position of the decimal point for both input and output operations.

For F field conversion, the scale factor specifies a power of ten such that

external number=(internal number)*(power of ten)

For D and E field conversions, the scale factor multiplies the number by a power of ten, but the exponent is changed accordingly, leaving the number unchanged except in form. For example, if the statement

```
FORMAT (F8.3,E16.5)
```

corresponds to the line

```
26.451    -0.41321E-01
```

then the statement

```
FORMAT (-1PF8.3,2PE16.5)
```

corresponds to the line

COLUMNS.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
				2	.	6	4	5				-	4	1	.	3	2	1	0	0	E	-	0	3																

When no scale factor is given, it is understood to be 0. However, once a scale factor is given, it holds for all following D, E and F field conversions within the same format. The scale factor is reset to zero by giving a scale factor of zero. Refer to paragraph 5.2.1.4 for the effect of a scale factor on G field conversions.

Scale factors have no effect on I or Z conversions.

For input operations, the scale factor may be supplied for all real number inputs, but is ignored for all inputs containing an exponent in the external field. When used with input data, the direction of the decimal point movement is opposite that for an equivalent output specification. For example, if the statement

```
FORMAT (F8.3)
```

inputs data to be used internally as

26.451

then the statement

FORMAT (-1PF8.3)

inputs data to be used internally as:

2.645

5.2.2 LOGICAL DATA FIELD SPECIFICATION. Logical data can be transmitted in a manner similar to numerical data by use of the logical data format. The field specification for the format is:

Lw

where L is the control character and w is an integer specifying the field width.

Data is transmitted as the value of a logical variable in the input/output list.

On input, the first non-blank character in the data field must be T or F, and the value of the logical variable is stored as true or false, respectively. The remainder of the field is ignored. If the entire data field is blank, a value of false is stored.

On output, w-1 blanks followed by T or F is output, depending on whether the value of the logical variable is true or false, respectively.

5.2.3 ALPHANUMERIC AND LITERAL DATA SPECIFICATIONS. The following paragraphs describe printer carriage control characters, the alphanumeric data field specification, and the specifications for the H and X fields.

5.2.3.1 Carriage Control. The first character in a record to be printed on a listing device is used for carriage control. This character is not printed and is usually specified at the beginning of the format field specification for the record as 1Hx, where x is a blank, 0, 1, or +. The specification may also be a slash (/). The effect of each character is given as follows:

Blank	Skip one line before printing.
0	Skip two lines before printing.
1	Skip to first line of next page before printing.
+	Suppress line spacing before printing (overprint).
/	Skip one line before printing – does not require 1H prefix.

Example:

10 FORMAT(9H1bbbPAGEb,I3/1H0)

In this statement the character "1" following the 9H prefix is interpreted as carriage control and causes the literal message " PAGE " to be printed at the top of the next page. Similarly, the character "0" following the 1H prefix is interpreted as carriage control to skip two lines before the next line is printed. The slash (/) represents an end-of-record and causes a line feed before the 1H0 specification is interpreted. The net result is three blank lines between the " PAGE " label output and any subsequent lines.

5.2.3.2 A Field Specification. The specification for alphanumeric data has the form:

```
Aw
```

where A is the control character and w is the number of characters in the field.

Alphanumeric data can be transmitted in a manner similar to numerical data by use of the form Aw. The alphanumeric characters are transmitted as the value of a variable in an input/output list. The variable may be of any type. For example, the sequence:

```
      READ (2,5)V
    5 FORMAT (A2)
```

causes two characters to be read and placed in memory as the value of the variable V.

The number of characters transmitted is limited by the maximum number of characters which can be stored in the space allotted for the variable. If w exceeds the available space, leading characters are lost on input and replaced with blanks on output. When w is less than the available space, blanks are filled in after the given characters until the maximum is reached. If character strings are read one character per word, each character is stored in the word left justified, and an ASCII blank is inserted in the right half of the word. Thus, character strings may be read and tested at the FORTRAN level as follows:

```
      COMMON ICHAR (80)
      DATA IB/2HBƀ/
      READ (5,6) ICHAR
    6 FORMAT (80A1)
      IF (ICHAR(1).EQ.IB) GO TO 7
```

5.2.3.3 H Field Specification. An alphanumeric format, or Hollerith data, field may be specified within a format by preceding the alphanumeric string by the form nH. H is the control character and n is the number of characters in the string, counting blanks. In all examples, ƀ represents a blank.

```
   FORMAT (17HƀPROGRAMƀCOMPLETE)
```

can be used to output

```
   PROGRAM COMPLETE
```

on the output listing. On input the external characters are stored in the format itself only if there is no list in the READ statement.

An alternate form of this field is to specify the alphanumeric string within single quotation marks. The FORMAT statement above could have been written this way:

FORMAT ('ƀPROGRAMƀCOMPLETE')

The input and output results are the same as the first FORMAT statement in this paragraph. Where an apostrophe appears in a character string of this nature, it must be represented by two single quotation marks in succession ('').

An alphanumeric format field may be placed among other fields of the format. For example, the statement

FORMAT (I5,8HƀFORCEƀ=F10.5)

can be used to output the line

22 FORCE = 17.68901

The separating comma may be omitted after an alphanumeric format field.

5.2.3.4 X Field Specification. Blanks may be introduced into an output record or characters skipped in an input record by use of the specification nX. The control character is X, and n is the number of blanks or characters skipped. The quantity n must be greater than zero. For example, the statement

FORMAT (5HƀSTEPI5,10X,3HYƀ=F7.3)

may be used to output the line

STEP 28ƀƀƀƀƀƀƀƀƀƀY = 3.872

where ten blanks separate the two quantities.

5.2.3.5 T Field Specification. For tabular output, the T field specification can be used to designate the position in the output record where data transfer will begin. The form of the specification is

Tn

where T is the control character and n designates the character position within the output record that will be occupied by the first character of the transfer. For printed output, the first character of the output record is reserved for carriage control. Therefore, a one-character skew occurs between the character's position in the output record and its position on the printed line.

Example:

FORMAT(T20,'NAME')

This statement results in printing the character string NAME beginning at printed column position 19. Also, because of the presence of the carriage control character, the following two statements produce identical output:

FORMAT(T1,6HbGRADE)

FORMAT(T2,5HGRADE)

Both of these statements print the character string GRADE beginning at the leftmost print position. A combination of these two statements, however, is not correct:

FORMAT(T1,5HGRADE)

causes the format editor to interpret the letter G as a carriage control character and output the character string RADE beginning at the leftmost print position.

To create output in many columns, several T field specifications may be included in one FORMAT statement. The order that the specifications appear in the FORMAT statement is not important; that is, a T70 specification may appear in the statement before a T12 specification without affecting the desired appearance of the output.

Example:

FORMAT(T20,'NAME',T40,'AGE',T1,6HbGRADE)

prints the character string GRADE beginning at the leftmost print position, NAME beginning at printed column position 19, and AGE beginning at printed column position 39.

5.2.4 COMPLEX QUANTITIES. Complex quantities are transmitted as two independent real quantities. The format specification is given as two successive real specifications or one repeated real specification. For instance, the statement

FORMAT (2E15.4,2(F8.3,F8.5))

could be used in the transmission of three complex quantities.

5.2.5 REPEATED GROUP AND FIELD SPECIFICATIONS. Repetition of the same field specification may be indicated conveniently in a FORMAT statement. In addition, groups of repeated specifications may be represented conveniently.

5.2.5.1 Repetition of Field Specifications. Repetition of a field specification may be performed by preceding the control character D, E, F, G, I, L, A or Z by an unsigned integer giving the number of repetitions desired. For example,

FORMAT (2E12.4,3I5)

is equivalent to

FORMAT (E12.4,E12.4,I5,I5,I5)

5.2.5.2 Repetition of Groups. A group of field specifications may be repeated by enclosing the group in parentheses and preceding the whole with the repetition number. For example,

FORMAT (2I8,2(E15.5,2F8.3))

is equivalent to

FORMAT (2I8,E15.5,2F8.3,E15.5,2F8.3)

Up to two levels of parentheses are allowed in group repetition, in addition to those enclosing the entire format. For example, the statement

FORMAT (I5,2(E15.5,2(F8.3,F7.1)))

exhibits the maximum level of nesting. The outermost parenthesis pair is termed level zero, the next level one, and the innermost level two.

5.2.6 RECORD SEPARATION INDICATOR (SLASH). To specify a group of input/output records with different records having different field specifications, use a slash "/" to indicate a new record. For example, the statement

FORMAT (3I8/I5,2F8.4)

is equivalent to

FORMAT (3I8)

for the first record and

FORMAT (I5,2F8.4)

for the second record. The separating comma may be omitted when a slash is used. Blank records may be written on output or records skipped on input by using consecutive slashes.

Both the slash and the closing parenthesis at the end of the format indicate the termination of a record. If the list of an input/output statement dictates that transmission of data is to continue after the closing parenthesis of the format is reached, the format is repeated from the last left parentheses of level one or zero. Level one is used when it exists; otherwise, level zero is used. Thus, the statement

FORMAT (F7.2,(2(E15.5,E15.4),I7))

causes the format

F7.2,2(E15.5,E15.4),I7

to be used on the first record and the format

2(E15.5,E15.4),I7

on succeeding records.

As a further example, consider

FORMAT (F7.2/(2(E15.5,E15.4),I7))

The first record has the format

```
F7.2
```

and successive records have the format

```
2(E15.5,E15.4),I7
```

5.2.7 FORMATS STORED AS DATA. The alphanumeric string comprising a format specification may be stored as the values of an array. Input/output statements may reference the format by giving the array name rather than the statement number of a FORMAT statement. The stored format must have the same form as a FORMAT statement, excluding the word FORMAT. The enclosing parentheses must be included.

As an example, consider the sequence

```
    INTEGER SKELETON(6)
    READ (5,1)(SKELETON(I),I=1,6)
1   FORMAT (6A2)
    READ (5,SKELETON)K,X
```

The first READ statement enters the character string into the array SKELETON. The input record contains

```
(I5,F9.3)
```

The second READ statement will cause K and X to be input as if the following format were used:

```
FORMAT (I5,F9.3)
```

5.2.8 COMBINATIONS OF FORMATS. The different types of field specifications described in the preceding paragraphs may be combined and used in a single FORMAT statement. An alphanumeric format field (H field specification) may be combined with other field specifications as described in paragraph 5.2.3.3.

5.2.9 FREE FIELD FORMAT. When performing a Formatted READ with 990 FORTRAN, the data can be arranged in a "free field" format that terminates each field with a comma. The comma denotes both the end of one field and the start of the next field. Data in the field is right justified. To use this format the field specification must account not only for the positions used by the largest number, but also for the position used by the comma. For example, to place a 1- to 4- digit positive integer into five sequential fields using the "free field" format, the format statement must be changed to

```
FORMAT(5I5)
```

to allow for the comma position. The input record can appear as

```
3,52,538,6744,5313,
```

WARNING

The "free field" format may not be used to read alphanumeric data.

The last comma in the input record is necessary to maintain the value of the parameter. Without that comma, the value is interpreted as a standard I5 format parameter and is assigned the value of 53130. The comma can be omitted if a zero or blank precedes the 5313 in the input record, i.e.,:

,ƀ5313

NOTE

The character ƀ used throughout this manual represents one blank character.

When the "free field" format is used from a data terminal, a carriage return may replace the last comma to terminate the data field and the input record.

5.3 READ AND WRITE STATEMENTS

In general, a READ or WRITE statement provides:

- Specification of the operation required, by using the verbs READ or WRITE to indicate input or output, respectively.
- The logical input/output unit to be used for transmission, i.e., the particular device involved.
- Reference to a data format which specifies the types of conversions required between internal and external data forms. The reference is either the number of a FORMAT statement or the identifier of an array which contains a data format. Data conversion is either from the input form to the form accepted by the FORTRAN runtime, or from the form accepted by the FORTRAN runtime to the output form.
- A list of the variables the values of which are to be transmitted. The values are transmitted in the order in which the information exists on the input medium or will exist on the output medium.

Example:

```
WRITE(1,5)A,X,K
```

This statement specifies that the values of A, X, and K, in that order, are to be written on logical I/O unit 1 according to the format given in the FORMAT statement numbered 5.

NOTE

When reading from or writing to the CRT screen, the appropriate carriage control (paragraph 5.3.5) must be specified in the format statement.

5.3.1 READ AND WRITE SEQUENTIAL FILE CHARACTERISTICS. A sequential file has the following characteristics:

- If the file contains one or more records, those records exist as a totally ordered set.
- There exists a unique position of the file called its initial point. If a file contains no records, the unit is positioned (physically or logically) to write starting at the initial point. If the unit is positioned at the initial point and the file contains records, the first record of the file is defined as the next record. No record precedes the initial point.
- If a unit is positioned at a point of the file beyond the initial point, a unique preceding record and a unique next record are associated with that position.
- Upon completion of the execution of a WRITE or END FILE statement, there exist no records following the records created by that statement.
- When the next record is transmitted, the position of the records being read or written by the unit is changed so that the record just transmitted becomes the preceding record.

5.3.2 READ AND WRITE RECORD CHARACTERISTICS. A data file consists of one or more records. Records may be formatted or unformatted. *A formatted record consists of a string of the characters that are permissible in alphanumeric constants.* The transfer of such a record requires that a FORMAT statement be referenced to supply the necessary positioning and conversion specifications. The number of records transferred by the execution of a formatted READ or WRITE is dependent upon the list and format specification. An unformatted record consists of binary data. When an unformatted or formatted READ statement is executed, the required records on the identified unit must be, respectively, unformatted or formatted records.

5.3.3 READ AND WRITE INPUT AND OUTPUT LISTS. Each READ or WRITE statements includes a list of the names of variables, arrays, and array elements. The named elements are assigned values on input and have their values transferred on output. The list of a READ or WRITE statement specifies the order of transmission of variable values. During input, the new values of listed variables may be used in subscripts for variables appearing later in the list. For example:

```
READ(2,3)L,A(L),B(L+1)
```

reads a new value of L and uses this value in the subscripts of A and B.

The transmission of array variables may be controlled by indexing similar to that used in the DO statement. The list of controlled variables, followed by index control, is enclosed in parentheses and the whole acts as a single element of the list. For example:

```
READ(7,23)(X(K),K=1,4)
```

is equivalent to

```
READ(7,23)X(1),X(2),X(3),X(4)
```

The indexing may be nested as in the following

```
READ(2,13)((MASS(K,L),K=1,5),L=1,4)
```

This statement reads in the elements of array MASS in the order

```
MASS(1,1),MASS(2,1),...,MASS(5,1),MASS(1,2),...,MASS(5,4)
```

If an entire array is to be transmitted, the indexing may be omitted and only the array identifier written. The array is transmitted in order of increasing subscripts with the first subscript varying most rapidly. Thus, the example above can be written

```
READ(2,13)MASS
```

5.3.4 **FORMATTED READ STATEMENT.** The purpose of the READ statement is to transfer information to the computer's memory from an input device.

Form:

READ(c,f,END=S_1,ERR=S_2)list

where c is an unsigned integer (I*2) constant, or variable whose value is the logical unit number of the input device; f is the statement number of the FORMAT statement describing the data, or the name of an array containing FORMAT data; END=S_1 defines a statement number to receive control when an end of file is encountered; ERR=s_2 defines a statement number to receive control when an error is detected; and *list* is a list of variable names, separated by commas, which are assigned the values of the input data. The parameters, END=S_1 and ERR=S_2 are optional and may be in reverse order. In order to process the detected error function, NERRST (described in table 7-1) should be used. NERRST clears the error indicator and determines the error condition. The list indicates the number of items of data to be placed in memory and the order in which they are read. Any number of items may appear on a list.

For example, consider the following logical record:

```
11ƀ-337ƀƀ2ƀƀƀ400
```

The following statements are used in a program to input the information on the card:

```
     READ(5,3)N,I,S,K
   3 FORMAT(I2,1X,I4,2X,I1,3X,I3)
```

causing the variable N to be assigned the value 11, I the value -337, S the value 2, and K the value 400.

Each time the READ statement is executed, the variables obtain new values depending upon what information is in the next record read.

If an input record contains more values than there are items in a list, only enough values to fill the list are read. The remaining values are ignored. If a list requires more values than are on a record, the next sequential record or records are read until the list has been satisfied.

The READ (c,f) form may be used with a FORMAT statement to read alphanumeric data and modify an existing H-type field (described in paragraph 5.2.3.3). The information read is inserted into the memory location of the H-type field. The amount of data to be read in depends on the size of the field.

Example:

```
6   FORMAT (21HTHISƀISƀAƀREPLACEMENT)
        .
        .
        .
    READ(5,6)
```

These statements cause the 21 characters in the H-type alphanumeric field:

THIS IS A REPLACEMENT

to be replaced by the first 21 characters from the next record of the file on unit 5.

5.3.5 FORMATTED WRITE STATEMENT. The WRITE statement causes the computer to transfer information from memory to any one of the output devices.

Form:

WRITE(c,f,ERR=S_2)[list]

where c is an unsigned integer (I*2) constant, or variable whose value is the logical unit number of the output device; f is the statement number of the FORMAT statement describing the data or the name of an array containing FORMAT data; ERR=S_2 defines a statement number to receive control when an error is detected; and *list* is a list of variable names which are assigned the values of the output data. The parameters ERR=S_2 and list, are optional.

The WRITE(c,f) *list* form is used to write the data currently stored in memory, corresponding to the variable names in the *list*, onto the file on unit c, using a form specified by the FORMAT statement f. The conventions described in paragraph 5.2.4 are also used to define an output record.

The WRITE(c,f) form is a means of writing alphanumeric data. The FORMAT statement specifies the exact data to be written so that the need for an I/O list is eliminated.

Example:

```
    WRITE(6,10)
        .
        .
        .
10  FORMAT(19HTHISƀISƀAƀQUOTATION)
```

This may also be written:

```
    WRITE(6,10)
10  FORMAT('THISƀISƀAƀQUOTATION')
```

Two apostrophes together represent one in the format. Example:

FORMAT('DON''T')

is interpreted as (5HDON'T).

When using the WRITE statement, carriage control is edited into the output buffer for the teleprinter, line printer, and VDT. The editing is performed by the run-time subroutine F$XIOF and involves the first character in the output buffer.

For no advance, the character + is replaced by a carriage return.

For double spacing, the character 0 is replaced by a line feed, and a carriage return and line feed are inserted on the front of the buffer. Then the buffer address is decremented by two, and the character count is incremented by one.

For single spacing, all other characters in the first character position in the output buffer are replaced by a line feed, and a carriage return is inserted on the front of the buffer. Then the buffer address is decremented by one, and the character count is incremented by one.

Editing is not performed on output to disk, cassette, or device types greater than six. Refer to the *Model 990 Computer DX10 Operating System Reference Manual*, part number 946250-9703, table 6-3 for device types.

Application note: If an EIA device is interfaced to the computer via a line printer interface, carriage control is lost if output is written to a disk file and then copied from disk to the line printer.

5.3.6 UNFORMATTED READ AND WRITE STATEMENTS.

Forms:

READ(c,END=S_1,ERR=S_2)list

WRITE(c,ERR=S_2)list

where c is an unsigned integer constant, variable, or expression assigned the value of the logical unit number used; END=S_1 defines a statement number to receive control when an end of file is encountered; ERR=S_2 defines a statement number to receive control when an error is detected; and *list* is a list of variable names separated by commas. The parameters END=S_1 and ERR=S_2, are optional and may be in reverse order.

The READ(c) *list* form reads a record of binary information, with no conversion, into memory from unit c. No FORMAT statement is required. The number of items in the list determines the amount of data read. If the record contains more values than are listed, the unread items are skipped. The total length of the list of variable names must not be longer than the record.

Omitting the list from the READ statement causes the record on unit c to be skipped and no information is transmitted from the input device.

The WRITE(c) *list* form is used to write binary information, with no data conversion, on unit c.

5.4 INTERNAL TRANSMISSION

The ENCODE and DECODE statements are similar to formatted READ and WRITE statements except that no I/O unit is used in the data transfer. Data is transferred under format specifications from one area of computer memory to another.

5.4.1 ENCODE STATEMENT. LIST → BUFFER (≙ WRITE)

Form:

ENCODE(c,f,b,n)list

ENCODE(c,f,b)list

where:

c	is an integer constant or integer variable describing the number of characters per record in storage.
f	is a format reference.
b	is a simple variable, array reference, or array name at which the first record is to start. This is the "buffer".
n	is a simple integer variable into which the number of characters actually processed will be stored.
list	is as defined for I/O list.

This statement is analogous to a WRITE statement. It converts the data in the list according to the format and stores it in records beginning at b, with c characters per record. If the format attempts to convert more than c characters per record, an error message is given on the device assigned to the list file. If the format converts less than c characters, the remainder of the record is filled with blanks.

Example:

ENCODE(80,100,BUFF)A,B,C

ENCODE(I,FMT,B,COUNT)D

The first statement uses the format in statement 100 to transfer data from the list members A, B and C and store it in location BUFF in 80 character records.

The second statement uses the format defined in FMT to transfer data from the single list member D and store it in location B in records whose size is defined by I. When the transfer is complete, COUNT contains the actual number of characters stored in B.

5.4.2 DECODE STATEMENT.

BUFFER → LIST (READ)

Form:

```
DECODE(c,f,b,n)list

DECODE(c,f,b)list
```

where c,f,b,n are as defined for ENCODE.

DECODE is particularly useful when inputting records in several different forms with an unknown or random order. For example, the program might input a record in alphanumeric (literal) FORMAT, and then on the basis of keywords or character strings in the input use a DECODE statement to place numeric values in the appropriate numeric variable.

This statement is analogous to a READ statement. It converts and edits the data from the records starting at b and consisting of c characters each, and stores it in the variables specified in the list. When the format specifies more than c characters per record, an error message is given. When fewer than c characters per record are specified, the remainder of the record is ignored.

Example:

```
DECODE(78,103,BUFFER)LDATA
```

The statement transfers records of 78 characters from BUFFER to LDATA and uses the format defined in statement 103.

5.5 MASS STORAGE FILE INPUT/OUTPUT STATEMENTS

There are three types of mass storage file input/output statements. Use of these statements is limited to magnetic tape, cassette and disk applications. They are REWIND, BACKSPACE, and END FILE.

5.5.1 REWIND STATEMENT.

Form:

```
REWIND u
```

where u is an I/O unit designation.

This statement directs the I/O unit designated to reposition to the first record of the first file. The unit designation is given as a constant or a simple integer (I*2) variable.

Examples:

```
REWIND 7
REWIND KMIO
```

5.5.2 BACKSPACE STATEMENT.

Form:

```
BACKSPACE u
```

where u is an I/O unit designation.

This statement directs the designated I/O unit to backspace one logical record. The unit designation is given as a constant or a simple integer (I*2) variable. If backspace is requested after a write, an end of file is done before the backspace is executed.

Examples:

```
BACKSPACE 7
BACKSPACE N
```

5.5.3 END FILE STATEMENT.

Form:

```
END FILE u
```

where u is an I/O unit designation.

The statement directs the I/O unit designated to terminate the file being written. The unit designation is given as a constant or a simple integer (I*2) variable.

Examples:

```
END FILE 4
END FILE K
```

5.6 DIRECT ACCESS INPUT/OUTPUT

Direct access I/O statements allow the programmer to access records within a relative record file in a random, rather than sequential, manner. For direct access, a file is viewed as a collection of n records, and each record is assigned a unique record number in the range 0 to n-1. The programmer must specify in READ, WRITE, and FIND statements not only the unit number, as for sequential I/O, but also the number of the first record to be read, written, or found. There are four direct access I/O statements: READ, WRITE, DEFINE FILE, and FIND.

5.6.1 DEFINE FILE STATEMENT. The DEFINE FILE statement describes the characteristics of an I/O unit to be used for direct access I/O operations. To use a direct access READ, WRITE, or FIND statement, the I/O unit must be described with a DEFINE FILE statement. Only direct access I/O statements may refer to units defined by DEFINE FILE statements. The DEFINE FILE statement must logically precede (i.e., must be executed prior to) any I/O statement referring to the unit defined.

Form:

DEFINE FILE $u_1(n_1,s_1,f_1,v_1),u_2(n_2,s_2,f_2,v_2)\ldots$

where:

u is an integer constant, or a simple integer (I*2) variable, I/O unit designation (logical unit number).

n is an integer constant specifying the number of records in the file.

s is an integer constant specifying the maximum size of each record. The record size is in terms of bytes or memory words, depending upon the specification of f.

f indicates whether or not the file is to be accessed with format control. The f parameter may be one of the following:

L indicates the file is to be read or written with or without format control. The maximum record size s is in bytes.

E indicates the file is to be read or written with format control. The maximum record size s is in bytes.

U indicates the file is to be read or written without format control. The maximum record size s is in words.

v is a simple integer variable. At the conclusion of each read or write operation, v is set to the record number of the record that immediately follows the last record transmitted. At the completion of a find operation, v is set to the record number of the record found. The record numbers range from 0 to n-1, where v is called the associated variable.

Under the TX990 operating system a relative record file must be created before execution by using the TX Development System (TXDS). With the DX10 operating system a file can be created either before execution or at runtime. If it is created at runtime, the file will be a noncontiguous, relative record file. The file will be created with a physical record length of 864 (360_{16}) characters. If a file is to be created before execution, the DX10 System Command Interpreter (SCI) must be used to create a relative record file. Additional information describing logical and physical records is found in paragraph 5.8.

Figure 5-1 is an example of the first four logical records from a precreated relative record file.

```
FILE   TSTFIL RECORD 0
0000   FFFF 5448 4953 2049 5320 4120 5445 5354   .. TH IS  I S  A  TE ST
0010   204C 494E 4520 4652 4F4D 2052 4543 443A    L IN E  FR OM  R EC D:
0020   0000 5448 4953 2049 5320 4120 5445 5354   .. TH IS  I S  A  TE ST
0030   204C 494E 4520 4652 4F4D 2052 4543 443A    L IN E  FR OM  R EC D:
0040   0000 5448 4953 2049 5320 4120 5445 5354   .. TH IS  I S  A  TE ST
0050   204C 494E 4520 4652 4F4D 2052 4543 443A    L IN E  FR OM  R EC D:
0060   0000 5448 4953 2049 5320 4120 5445 5354   .. TH IS  I S  A  TE ST
0070   204C 494E 4520 4652 4F4D 2052 4543 443A    L IN E  FR OM  R EC D:
0080   0000                                      ..

FILE   TSTFIL RECORD 1
0000   FFFF 5448 4953 2049 5320 4120 5445 5354   .. TH IS  I S  A  TE ST
0010   204C 494E 4520 4652 4F4D 2052 4543 443A    L IN E  FR OM  R EC D:
0020   0001 5448 4953 2049 5320 4120 5445 5354   .. TH IS  I S  A  TE ST
0030   204C 494E 4520 4652 4F4D 2052 4543 443A    L IN E  FR OM  R EC D:
0040   0001 5448 4953 2049 5320 4120 5445 5354   .. TH IS  I S  A  TE ST
0050   204C 494E 4520 4652 4F4D 2052 4543 443A    L IN E  FR OM  R EC D:
0060   0001 5448 4953 2049 5320 4120 5445 5354   .. TH IS  I S  A  TE ST
0070   204C 494E 4520 4652 4F4D 2052 4543 443A    L IN E  FR OM  R EC D:
0080   0001                                      ..

FILE   TSTFIL RECORD 2
0000   FFFF 5448 4953 2049 5320 4120 5445 5354   .. TH IS  I S  A  TE ST
0010   204C 494E 4520 4652 4F4D 2052 4543 443A    L IN E  FR OM  R EC D:
0020   0002 5448 4953 2049 5320 4120 5445 5354   .. TH IS  I S  A  TE ST
0030   204C 494E 4520 4652 4F4D 2052 4543 443A    L IN E  FR OM  R EC D:
0040   0002 5448 4953 2049 5320 4120 5445 5354   .. TH IS  I S  A  TE ST
0050   204C 494E 4520 4652 4F4D 2052 4543 443A    L IN E  FR OM  R EC D:
0060   0002 5448 4953 2049 5320 4120 5445 5354   .. TH IS  I S  A  TE ST
0070   204C 494E 4520 4652 4F4D 2052 4543 443A    L IN E  FR OM  R EC D:
0080   0002                                      ..

FILE   TSTFIL RECORD 3
0000   FFFF 5448 4953 2049 5320 4120 5445 5354   .. TH IS  I S  A  TE ST
0010   204C 494E 4520 4652 4F4D 2052 4543 443A    L IN E  FR OM  R EC D:
0020   0003 5448 4953 2049 5320 4120 5445 5354   .. TH IS  I S  A  TE ST
0030   204C 494E 4520 4652 4F4D 2052 4543 443A    L IN E  FR OM  R EC D:
0040   0003 5448 4953 2049 5320 4120 5445 5354   .. TH IS  I S  A  TE ST
0050   204C 494E 4520 4652 4F4D 2052 4543 443A    L IN E  FR OM  R EC D:
0060   0003 5448 4953 2049 5320 4120 5445 5354   .. TH IS  I S  A  TE ST
0070   204C 494E 4520 4652 4F4D 2052 4543 443A    L IN E  FR OM  R EC D:
0080   0003                                      ..
```

Figure 5-1. Define File Example Output

Example:

```
      INTEGER*2 IREC
      DIMENSION K(15)
      DATA K/'THIS IS A TEST LINE FROM REC D'/

      DEFINE FILE 8(50, 64, U, IREC)
      DO 10 J=1, 50
      I=50-J
10    WRITE (8'I)K,I,K,I,K,I,K,I
      END
```

The above example is a program that writes 50 (32_{16}) records beginning with record 31. The file is on unit 8. Each record is 64 words of unformatted data. IREC contains the number of the record that immediately follows the last record transmitted. The file has been created before execution with a logical record size of 64, and physical record size of 130. The first four records generated by this example program are shown in figure 5-1.

5.6.2 READ STATEMENT.

Form:

READ (u'n,f,END=S_1,ERR=S_2)list

where:

u is an I/O unit designation.

n is an integer expression representing the relative record number.

f is an optional format reference.

END=S_1 and ERR=S_2 are optional and their order may be reversed. They indicate exit destinations in the event of an end-of-file or an error, respectively.

Example:

```
          DIMENSION K(6)

          DEFINE FILE 4(500,80,E,IF4)

  100     FORMAT (/5(7X,I3))

  1001    READ (4'6,100)(K(I),I=1,6)

  1002    READ (4'IF4-3,100)K
```

The READ statement 1001 reads records 7 and 8 from the file associated with unit 4. The READ statement specifies record 6 but the format statement contains a leading slash which causes a skip to the following record. Two records must be read in order to satisfy the I/O list of six elements in the array K. At the completion of the first READ, the associated variable IF4 is set to the value 9, the record following the last one read. Thus, the second READ, statement 1002, performs exactly the same as the first READ because the relative record expression, IF4-3, is equal to 6.

5.6.3 WRITE STATEMENT.

Form:

```
WRITE (u'n,f,ERR=S2)list
```

where:

u is an I/O unit designation.

n is an integer expression representing the relative record number.

f is an optional format reference.

$ERR=S_2$ is optional and indicates an exit destination in the event of an error.

Example:

```
    DEFINE FILE 5(40,20,U,IF5)
    I = 25
88  WRITE (5'I)A,B,C
99  WRITE (5'IF5)X,Y
```

The first WRITE statement, number 88, writes record number 25 and the second WRITE, number 99, writes record 26.

5.6.4 FIND STATEMENT. The FIND statement causes the next input record to be located while the present record is being processed. This increases the execution speed of the program. FIND does not transfer any data to or from the file. There is no advantage to preceding a WRITE with a FIND.

Form:

```
FIND (u'n)
```

where:

u is an I/O unit designation.

n is an integer expression representing the relative record number.

Example:

```
FIND (3'1526)
  .
  .
  .
READ (3'1526)X
```

5.7 CONSOLE DISPLAY INPUT/OUTPUT

The ACCEPT and DISPLAY statements allow the programmer to transfer data to and from the screen of cathode ray tube I/O devices (CRTs).

5.7.1 ACCEPT STATEMENT.

Form:

ACCEPT (u,f,LINE=n_1,POSITION=n_2,ERASE,PROMPT,ECHO,ERR=S_1)item

where:

u is an I/O unit designation (TX operating system ignores an I/O unit designation; see G.2.3).

f is a format reference.

LINE=n_1 is a line number on the CRT screen that is an integer constant or integer variable in the range 1 to the number of lines on the screen. If the value is greater than the number of lines on the CRT screen, it is adjusted modulo to the maximum. If the value is zero or the LINE phrase is not present, then the data is accepted from the next line below the current cursor position. Unless the value specified in the POSITION phrase is zero, then the data is to be accepted from the line at the current cursor position. If incrementing to the next line generates a line number greater than the maximum number of lines on the screen, an automatic erase occurs with reset to line one.

POSITION=n_2 is a character position in a line that is an integer constant or integer variable in the range 1 to the number of characters per line. If the value is greater than the maximum number of characters within a line on the screen, it is adjusted modulo to the maximum numbers of characters.

ERASE means clear the screen before input.

PROMPT means fill input field on the screen with asterisks before input.

NOTE

> When using prompt for alphanumeric input, the system will not allow asterisks to be keyed in.

ECHO causes the contents of an item to be displayed on the CRT screen. Conversion and justification occur prior to display. If the ECHO phrase is not specified, the original input data remains in the field.

ERR=S_1 indicates a statement number to transfer control to when an error is detected.

Item is a scalar variable, a subscripted array reference, or an unsubscripted array name.

All of the parameters in the parenthesis with the exception of u and f are optional and may appear in any order.

Example:

```
    ACCEPT(10,99,LINE=3,POSITION=10,PROMPT)ICOUNT
99  FORMAT(I5)
```

This ACCEPT statement would display five asterisks on line 3 in positions 10-14, position the cursor at the first asterisk, and wait for a five digit integer to be input. Input is terminated by the pressing of the NEW LINE key. When it is, the value is transferred to the variable ICOUNT.

NOTE

The item is accepted under normal FORTRAN input conversion rules. Under some input specifications, positioning is significant.

To input alphanumeric data you may use integer, real or double precision scalar variables or arrays. To insure that the input field width agrees with the alphanumeric character capacity of the variable type, the following formats should be used.

Variable Type	Format Specification
Integer*2	A2
Real*4	A4
Real*8	A8

No repetition factor need be specified for arrays since the field width is automatically adjusted according to the array size. The use of variables and arrays which are of type COMPLEX should be avoided for input of alphanumeric data.

When using the ACCEPT statement to input numeric data into arrays, the repetition factor is automatically determined as in the case of alphanumeric input. Care should be taken to insure that each data item is in its proper field, unless commas are used to separate the data items, in which case the only requirement is that all the input items fit into the field width for the entire array.

5.7.2 DISPLAY STATEMENT.

Form:

DISPLAY(u,f,LINE=n_1,POSITION=n_2,ERASE,ERR=S_1)list

where:

u is an I/O unit designation. (Ignored by TX operating system, see G.2.3.)

f is a format reference.

LINE=n_1 is a line number on the CRT screen that is an integer constant or integer variable in the range 1 to the number of lines on the screen. If the value is greater than the number of lines on the CRT screen, it is adjusted modulo to the maximum. If the value is zero or the LINE phrase is not present, then the data is accepted from the next line below the current cursor position. Unless the value specified in the POSITION phrase is zero, the data is to be displayed from the line at the current cursor position. If incrementing to the next line generates a line number greater than the maximum number of lines on the screen, an automatic erase occurs with reset to line one.

POSITION=n_2 is a character position in a line that is an integer expression in the range 1 to the number of characters per line. If the value is greater than the maximum number of characters within a line on the screen, it is adjusted modulo to the maximum number of characters.

ERASE is clear screen before output.

ERR=S_1 indicates a statement number to transfer control to when an error is detected.

and list is a normal I/O list.

Example:

```
27  DISPLAY(10,55,LINE=5,POSITION=20,ERASE)
55  FORMAT('STATEMENT 27 EXECUTED')
```

All of the parameters in the parentheses except u and f are optional and may appear in any order.

If a slash appears in the DISPLAY's format statement the output for each line begins at the specified position number and does not begin at column 1.

Since display output is a normal I/O list, an array containing alphanumeric characters can be output in the normal fashion under regular format specifications. See the preceding section on ACCEPT statements for information on alphanumeric field widths. Repetition factors should be specified in DISPLAY formats to output an entire string on one line.

5.8 LOGICAL AND PHYSICAL RECORD CHARACTERISTICS

Input/output record formats associated with the FORTRAN READ, WRITE, REWIND, BACKSPACE, and END FILE statements and the BUFOUT subroutine are described in the following paragraphs.

A FORTRAN logical record is a unit of data which contains one or more FORTRAN physical records. To compute the logical record length (LRECL) for formatted I/O, multiply the LRECL in words by 2. To compute the LRECL for unformatted I/O, multiply the LRECL in words by 2 and add 2 words for a control word.

A relative record file has one physical record in a logical record, and a sequential file has one or more physical records in a logical record.

The maximum FORTRAN buffer size is 144 bytes. Additional information about the I/O buffer may be found in Appendix E paragraph E.3.2

Each unformatted write statement produces one logical record consisting of one or more physical records. The first word of each physical record contains a zero, and the last physical record of the logical record contains a nonzero count. The count in the last physical record indicates the number of physical records in the logical record. If the physical record count is positive, the file is an ASCII file. If the count is negative, the file is an unformatted file. The physical record count is used in the execution of the backspace statement.

For a sequential file, the logical record can be any size since the length is dependent on the amount of data written to the file. The logical record is broken up into physical records 144 bytes in length when using READ and WRITE statements. When using BUFOUT, the physical record length may be any size.

When using relative record files, a DEFINE FILE statement is required. Since the maximum buffer size is 144 bytes, the largest usable logical record length in the DEFINE FILE statement is 144 bytes. If the specified logical record length is greater than 144 bytes, only 144 bytes are transferred for a READ or WRITE operation. A truncation warning message is issued if the LRECL specified in the DEFINE FILE statement is greater than 144 bytes, or if the defined LRECL exceeds the created LRECL.

When using the DX10 operating system, a relative record file can be created at runtime. The file will be noncontiguous and have a physical record length of 864 bytes. If the file is to be created before execution, the DX10 System Command Interpreter (SCI) must be used to create a relative record file. For efficient use of disk space and faster random access, the physical record length should be evenly divisible by the FORTRAN logical record length.

Under the TX990 operating system, a relative record file must be precreated using the TX development system (TXDS). At present, the sector size on a diskette is 128 bytes. When writing to the diskette, each logical record begins on a sector boundary. Therefore, for maximum utilization of the space on a diskette, the FORTRAN logical record length should be evenly divisible by 128.

Attempts to read or write unformatted records on printing devices result in an error condition and control is returned to the operating system. Attempts to read from a device which cannot read, and attempts to write on a device which cannot write, also result in an error condition and a return to the system. However, use of the ERR=S_1 specification in the I/O statement preempts the return of control to the operating system and transfers control to the indicated statement number.

5.9 FORTRAN UNIT NUMBERS

Input and output unit numbers used in FORTRAN programs may be any decimal INTEGER*2 value from 1 to 99. These unit numbers must be assigned to actual units to be used before the program is executed. Appendixes G and H contain LUNO assignment for TXDS and DX10 respectively.

A device is treated as a sequential file. If one logical unit number is used to perform a Write, Read sequence (i.e. both input and output), a Rewind must be performed after the Write command.

5.10 VALID FORTRAN I/O OPERATIONS

When working with FORTRAN I/O operations at a particular unit number, only certain operations can follow a given operation. Table 5-1 may be used to validate DX FORTRAN I/O operations, and table 5-2 may be used to validate TX FORTRAN I/O operations. Given the last operation, the table indicates if the current operation is valid. For example, a WRITE operation followed by a READ operation causes an error found in table 9-4. The error type is an illegal operation.

Table 5-1. Validate DX FORTRAN I/O Operations

LAST OPERATION WAS \ CURRENT OPERATION IS	RDS	WRT	RWD	BSP	EOF	BIO	BTS
RDS	YES	YES	YES	YES	YES	ERR	ERR
WRT	ERR	YES	EOF	EOF	YES	ERR	ERR
RWD	YES	YES	YES	WR1	YES	YES	YES
BSP	YES	YES	YES	YES	YES	YES	YES
EOF	ERR	YES	YES	YES	YES	YES	YES
BIO	ERR	ERR	ERR	ERR	ERR	ERR	YES
BTS	ERR	ERR	YES	YES	YES	YES	WR2
READ EOF	ERR	ERR	YES	YES	ERR	ERR	ERR

OPERATIONS:

RDS – READ
WRT – WRITE
RWD – REWIND
BSP – BACKSPACE
EOF – END OF FILE
BIO – BUFFER I/O (BUFIN/BUFOUT)
BTS – BUFFER I/O STATUS CHECK (IUNIT)

OPERATION RESULTS:

YES – Operation is performed.
ERR – Operation is not allowed. Refer to table 9-4 for error messages.
EOF – An end of file is generated.
WR1 – Warning, backspace after rewind.
WR2 – Warning, redundant buffer I/O status check.

Table 5-2. Validate TX FORTRAN I/O Operations

CURRENT OPERATION IS

LAST OPERATION WAS	RDS	WRT	RWD	BSP	EOF	BIO	BTS
RDS	YES	ERR	YES	YES	ERR	ERR	ERR
WRT	ERR	YES	EOF	ERR	YES	ERR	ERR
RWD	YES	YES	YES	WR1	YES	YES	YES
BSP	YES	ERR	YES	YES	YES	YES	YES
EOF	ERR	YES	YES	ERR	YES	YES	YES
BIO	ERR	ERR	ERR	ERR	ERR	ERR	YES
BTS	ERR	ERR	YES	YES	YES	YES	WR2
READ EOF	YES	YES	YES	ERR	YES	ERR	ERR

OPERATIONS:

RDS – READ
WRT – WRITE
RWD – REWIND
BSP – BACKSPACE
EOF – END OF FILE
BIO – BUFFER I/O (BUFIN/BUFOUT)
BTS – BUFFER I/O STATUS CHECK (IUNIT)

OPERATION RESULTS:

YES – Operation is performed.
ERR – Operation is not allowed. Refer to table 9-4 for error messages.
EOF – An end of file is generated.
WR1 – Warning, backspace after rewind.
WR2 – Warning, redundant buffer I/O status check.

SECTION VI

FUNCTIONS AND SUBPROGRAMS

6.1 GENERAL

In the FORTRAN language it is possible to specify relatively complicated operations, such as the determination of the sine of an angle or the printing of variable multiline header information, in one program instruction. This is accomplished by first programming the operation to be performed, and then accessing the code for this operation explicitly by a CALL statement, or implicitly by a function reference. Several methods and requirements exist for using the available FORTRAN facilities; they are discussed in detail in this chapter.

6.2 SUBPROGRAMS

The following paragraphs define several terms needed to explain arithmetic functions and subprograms, and describe the purpose of dummy identifiers.

6.2.1 DEFINITIONS. FORTRAN subprograms may be internal or external. Internal subprograms are defined within the program which calls them. They are defined within a single statement, the function definition statement. Internal subprograms are defined and may be used only within the program containing the definition. External subprograms are defined separately from (external to) the program which calls them and are complete programs conforming to all the rules of FORTRAN programs. They are compiled independently.

Two types of external subprograms may be defined: FUNCTION subprograms and SUBROUTINE subprograms. The use of the declarations FUNCTION and SUBROUTINE in the definition of these subprograms is described in paragraphs 6.4 and 6.5.

Intrinsic functions are external subprograms which are predefined in the FORTRAN language. The intrinsic function identifiers, definitions, and types are given in Section VII. These definitions are overridden by using the function identifier in any context other than a function reference.

Any subprogram, internal or external, may call other subprograms; recursion is allowed. If a subprogram calls itself, either directly or indirectly, recursion exists (paragraph 6.8).

6.2.2 DUMMY IDENTIFIERS. Subprogram definition statements declare certain identifiers to be dummies representing the arguments of the subprogram. They are used as ordinary identifiers within the subprogram definition and indicate what sort of arguments may appear and how the arguments are used. The dummy identifiers are replaced by the actual arguments when the subprogram is executed.

Dummy identifiers may not appear in COMMON, EQUIVALENCE, or DATA statements.

When a subprogram is called, the dummy parameters must agree with the actual parameters as to number, and should agree as to order, type, and length (except for Hollerith strings). For example, if an actual parameter is a 16-bit integer constant then the corresponding dummy parameter should be of INTEGER*2 type.

The number of parameters is checked and must match both at compile time and at runtime. Other attributes are not checked. If the actual and dummy parameters disagree in type, the subprogram must be aware of the disagreement. For example, a Hollerith constant may be used as an actual parameter corresponding to a dummy parameter which is a real array. If this is the case, the subroutine should not operate upon the array as though it contained floating point values.

If a dummy parameter is an array name, the corresponding actual parameter may be either an array name or an array element.

If a dummy parameter is assigned a value in the subprogram, the corresponding actual parameter should be a simple variable, array element, or array name. A constant or expression should not be used as an actual parameter if the corresponding dummy parameter may be assigned a value.

6.3 STATEMENT FUNCTION DEFINITIONS

Form:

identifier(identifier$_1$,identifier$_2$, . . .)=expression

The statement function definition statement defines an internal subprogram. The single statement contains the entire definition of the function. All statement function definitions must precede the first executable (nondeclarative) statement of the program.

To the left of the equal sign, an identifier followed by one or more identifiers in parentheses appear. The first identifier is the function name; the identifiers in parentheses represent arguments of the function and are known as dummy identifiers. To the right of the equal sign, the statement contains an expression whose value is the value of the function.

The function identifier is the name of the subprogram being defined. A function is single-valued and must have at least one argument. The data type of the function is determined by the type of the function identifier.

The arguments of the function are named by the dummy identifiers, which appear in parentheses after the function name. The following rules apply to dummy identifiers:

- They must be unique only within the individual function definition statement.
- They may be identical to identifiers of the same type appearing elsewhere in the program.
- They must agree in order, number, type and length with the actual arguments provided at execution time.

The portion of the function definition statement to the right of the equal sign must be an expression. The expression defines the value of the function. The occurrence of a dummy identifier determines how the identifier is handled in the defining expression. Identifiers that do not represent arguments are treated as ordinary variables. The following rules apply to the defining expression:

- Dummy identifiers may appear only as scalar variables in the defining expression, not as subscripted variables.
- The defining expression may include references to external functions or other previously defined internal functions.

Examples of function definition statements follow:

```
SSQR(K) = K*(K+1)*(2K+1)/6
NOR(T,S) = .NOT.(T.OR.S)
ACOSH(X) = (EXP(X/A)+EXP(-X/A))/2
```

In the second example above, the data type of the function NOR must have been previously declared to be logical. In the last example above, X is a dummy identifier and A is an ordinary identifer. At runtime the function is evaluated using the current value of the quantity represented by A.

6.4 FUNCTION SUBPROGRAM

A FUNCTION subprogram is a function which is single-valued and is referenced as a basic element in an expression. The subprogram is compiled separately from the main program, yet there may be an interchange of data between it and the main program. Its variable names are completely independent of the main program and other subprograms. A FUNCTION subprogram is called implicitly by use of a function reference in the main program.

A FUNCTION subprogram begins with a FUNCTION declaration and returns control to the calling program by means of one or more RETURN statements. The FUNCTION subprogram must terminate with an END statement.

Form:

[type] FUNCTION identifier(identifier$_1$,identifier$_2$, . . .)

This statement declares the program which follows to be a function subprogram. The argument type defines the type of the function. The first identifier is the name of the function being defined. This identifier must appear as a scalar variable and be assigned a value during execution of the subprogram. This value is the function value.

Example:

```
FUNCTION ROOT (A,B,C)
ROOT=(-B+SQRT(B**2-4.0*A*C))/(2.0*A)
RETURN
END
```

Identifiers appearing on the list enclosed in parentheses are dummy identifiers representing the function arguments. They must agree in number, order, type and length with the actual arguments given at runtime. A FUNCTION subprogram may have any number of arguments, including arrays, within the limitations of available storage area. FUNCTION subprogram arguments may be expressions, array names, or subprogram names. Dummy identifiers may appear in the subprogram as scalar identifiers, array identifiers, or subprogram identifiers.

Dummy identifiers which represent array names must be dimensioned within the subprogram by a DIMENSION or type statement. Dimensions given as constants must equal the dimensions of the actual arrays given at runtime. In a DIMENSION or type statement, dummy identifiers may be used to specify variable dimensions for array name arguments. For example, in the statement sequence

```
FUNCTION TABLE (A,M,N,B,X,Y)
    .
    .
    .
DIMENSION A(M,N),B(10),C(50)
    .
    .
    .
```

the dimensions of array A are specified by the dummies M and N, and the dimension of array B is given as a constant. The values given for M and N at runtime must be the actual dimensions of array A.

Dummy dimensions may be given only for dummy arrays. In the example above, the array C must be given absolute dimensions since C is not a dummy identifier.

The type of the function is the type of the identifier which names the function. Alternately, the function may be explicitly typed in the FUNCTION statement by replacing the word FUNCTION with one of the following:

INTEGER FUNCTION
INTEGER*2 FUNCTION
INTEGER*4 FUNCTION
FIXED (scale) FUNCTION
REAL FUNCTION
REAL*4 FUNCTION
REAL*8 FUNCTION
DOUBLE PRECISION FUNCTION
COMPLEX FUNCTION
LOGICAL FUNCTION

The symbolic name of the function may not appear in any nonexecutable statement except as the name in a function statement.

For example, the statement

```
COMPLEX FUNCTION HPRIME(S,N)
```

defines a complex function HPRIME.

Examples:

```
FUNCTION MAY (RANGE,XP,YP,ZP)
REAL FUNCTION COT(ARG)
```

The first example defines a FUNCTION subprogram MAY, with four dummy arguments. The second example explicitly defines the subprogram COT to be REAL.

6.5 SUBROUTINE SUBPROGRAM

Another type of subprogram that is compiled by itself is the SUBROUTINE subprogram. A SUBROUTINE subprogram is not a function; it may be multivalued and can be referred to only by a CALL statement. A SUBROUTINE subprogram begins with a SUBROUTINE declaration and returns control to the calling program by means of one or more RETURN statements. There are two possible forms for the SUBROUTINE declaration.

Forms:

```
SUBROUTINE identifier
SUBROUTINE identifier(identifier1identifier2,...)
```

This statement declares the program which follows to be a SUBROUTINE subprogram. The first identifier is the subroutine name. The identifiers in the list enclosed in parentheses are dummy identifiers representing the arguments of the subprogram. These identifiers must agree in number, order, type and length with the actual arguments given to the subprogram at runtime.

SUBROUTINE subprograms may have expressions, alphanumeric strings, array names, and subprogram names as arguments. The dummy identifiers may appear as scalar, array, or subprogram identifiers within the subprogram.

Dummy identifiers that represent array names must be dimensioned within the subprogram by a DIMENSION or type statement. As in the case of a FUNCTION subprogram, either constants or dummy identifiers may be used to specify dimensions in a DIMENSION or type statement.

A SUBROUTINE subprogram may use one or more of its dummy identifiers to represent results. The subprogram name is not used for return of results.

A SUBROUTINE subprogram need not have any arguments at all.

Examples:

```
SUBROUTINE EXIT
SUBROUTINE FACTOR (COEF,N,ROOTS)
SUBROUTINE RESIDU (NUM,D,DEN,M,RES)
```

The example shown in figure 6-1 illustrates the principal features of a SUBROUTINE subprogram. The arguments of subprogram AVGN include values that are inputs to the subroutine and results calculated by the subroutine for output to the main program. The inputs are an array name and the number of values to be averaged. The subprogram calculates an average value and the number of values that are greater than or equal to zero.

To call this subprogram, the main program might contain a statement like this:

```
CALL AVGN (ALTA,30,YMEAN,NPOS)
```

```
      SUBROUTINE AVGN(X,J,AVG,IZ)
C SUBROUTINE COMPUTES AVERAGE OF FIRST J ELEMENTS
C AND THE NUMBER OF ELEMENTS THAT ARE GREATER THAN
C OR EQUAL TO ZERO.
      DIMENSION X(100)
      AVG = 0.0
      IZ = 0
      DO 10 I=1,J
      AVG = AVG + X(I)
      IF (X(I) .GE. 0.0) IZ = IZ+1
   10 CONTINUE
      XN = J
      AVG = AVG/XN
      RETURN
      END
```

Figure 6-1. Example of Subroutine

This statement calls the subprogram to calculate the average of the first 30 values of an array named ALTA. The computed average is stored as the value of YMEAN, and the value IZ computed by the subprogram is stored as the value of NPOS. If the main program later requires the average of the first 52 values of an array named ARRAY, with the computed average to be stored in FPT3 and the number of values greater than or equal to zero stored in K, it would include this statement:

```
CALL AVGN (ARRAY,52,FPT3,K)
```

The calling program must provide values for the subprogram's input variables and must contain defined variables to store the values calculated by the subprogram.

The subprogram need not have arguments, as illustrated in this example:

```
      SUBROUTINE WARN
      WRITE(5,15)
   15 FORMAT('A DEFAULT HAS OCCURRED')
      RETURN
      END
```

6.6 SPECIFICATION SUBPROGRAMS FOR DATA INITIALIZATION

A specification subprogram, unlike the FUNCTION and SUBROUTINE subprograms, may not be referenced by any other program. This is indicated in the structure of the specification subprogram, i.e., it contains no executable statements, and its beginning statement contains no identifier. The purpose of this type of subprogram is to establish initial values of variables in the labeled common areas.

A specification subprogram begins with a BLOCK DATA declaration; therefore, it is sometimes referred to as a block data subprogram. Following the opening declaration, there may be DIMENSION, type, COMMON, and DATA declarations as required to specify the values to be assigned to the variables.

The data specification statements DATA and BLOCK DATA are used to specify initial values for variables. These values are compiled into the object program. They become the values assumed by the variables when execution begins.

6.6.1 DATA STATEMENT.

Form:

DATA $v_1/d_1/,v_2/d_2/,\ldots$

where v is a variable list and d is a data list.

The variable lists in a DATA statement consist of scalar variables, array names or array elements separated by commas.

Variables in common may appear in the lists only if the DATA statement occurs in a BLOCK DATA subprogram.

DATA statements may also be used in main programs and in FUNCTION and SUBROUTINE subprograms to set values for noncommon variables. When used in this way, the DATA statements must be placed before arithmetic function definition statements, if there are any; they *must* precede all executable (nondeclarative) statements.

The storage area that will be occupied by the initialized data items of each data list must match the storage length of the initialization. For example, simple integer data requires one word and real number data requires two words. The storage requirements of the six data types are listed in the discussion of types of data in Section II.

The following considerations apply to data items in a DATA statement:

- Data items may be numerical constants or alphanumeric strings. For example

  DATA ALPHA,BETA/5.,16.E-2/

  specifies the value 5.0 for ALPHA and the value 0.16 for BETA. The form of the constant, not the type of the variable, determines the data type of the stored constant.

- Alphanumeric data are packed into words, one or two characters per word, as in the case of A-conversion. Excess characters are not permitted.

  Example:

  DATA NOTE/2HNO/,REALVR/4HADCD/

- Any data item may be preceded by an integer and an asterisk. The integer indicates the number of times the item is to be repeated. For example

  DATA A(1),A(2),A(3),A(4),A(5)/61E2,4*32E1/

  specifies five values for the array A; the value 6100 for A(1) and the value 320 for A(2) through A(5).

- When an unsubscripted array name is included in the variable list, all elements of the array are initialized to the indicated value. The notation is equivalent to specifying each element of the array individually.

  Example:

  ```
  DIMENSION A(5)
  DATA A/61E2,4*32E1/
  ```

  is equivalent to

  ```
  DATA A(1),A(2),A(3),A(4),A(5)/61E2,4*32E1/
  ```

- The form of the constant, not the type of the variable, determines the data type.

Hollerith constants, or alphanumeric strings, are treated specially in the DATA statement. A single Hollerith constant may initialize more than one variable element, whereas any other type of constant corresponds to a single element. A Hollerith constant may not contain excess characters, but it is extended with blanks so as to fill an integral number of elements.

Examples:

The statements:

```
DIMENSION I(2)
DATA I/'A','B'/
```

imply:

```
I(1) = 'Ab'
I(2) = 'Bb'
```

and the additional statement

```
DATA I/'ABC'/
```

implies

```
I(1) = 'AB'
I(2) = 'Cb'
```

However, the statement:

```
DATA J/'ABC'/
```

is an error since J is not dimensioned.

Further, the statements:

```
DIMENSION R(2)
DATA R/2*'A'/
```

imply

```
R(1) = 'Ab'
R(2) = 'Ab'
```

6.6.2 BLOCK DATA STATEMENT.

Form:

```
BLOCK DATA
```

This statement declares the program which follows to be a data specification subprogram. Data specification for variables in common blocks requires the use of a BLOCK DATA subprogram.

The first statement of the subprogram must be the BLOCK DATA statement. The subprogram may contain only the declarative statements associated with the data being defined.

Example:

```
BLOCK DATA
COMMON/R/X,Y/C/Z,W,V
DIMENSION Y(3)
COMPLEX Z
DOUBLE PRECISION X
DATA Y(1),Y(2),Y(3)/1E-1,2*3E2/
DATA X,Z/11.877D0,(-1.41421,1.41421)/
END
```

Data may be entered into more than one common block in one subprogram. A blank common can not be initialized. Any common block mentioned must be listed sufficiently to define those variables being used. However, the first reference to a common block must be listed in full. In the example above, W and V are listed in block C although no data values are defined for them.

6.7 EXTERNAL STATEMENT

Form:

EXTERNAL $identifier_1, identifier_2, \ldots, identifier_n$

This statement declares the listed identifiers to be subprogram names. Any subprogram name given as an argument to another subprogram must appear in an EXTERNAL declaration in the calling program.

Example:

```
EXTERNAL SIN,COS
  .
  .
  .
CALL TRIGF (SIN,1.5,ANSWER)
  .
  .
  .
CALL TRIGF (COS,.87,ANSWER)
  .
  .
  .
END

SUBROUTINE TRIGF(FUNC,ARG,ANSWER)
  .
  .
  .
ANSWER = FUNC(ARG)
  .
  .
  .
RETURN
END
```

6.8 REENTRANT STATEMENT

Form:

```
REENTRANT      identifier,identifier,. . .,identifier
```

This statement declares the listed identifiers to be the names of subprograms that are to be called using the special reentrant subprogram calling sequence. The generated code for calling these subprograms is as follows:

```
*    REENTRANT SUB
*    CALL SUB(A,B)
          BLWP @F$XREC
          DATA SUB
          DATA 2
          DATA A
          DATA B
```

In order for the above to work properly, subroutine SUB must be coded with the special reentrant receiving protocol (Appendix B, paragraph B.3).

NOTE

The concept of reentrant subprograms serves two purposes. The first is to allow the user to share a subprogram between two or more separate tasks. This facility is only possible with a mapped processor running under the DX10 operating system and does not require the use of the REENTRANT statement nor the special calling sequence that it generates. Appendix H, paragraph H.2.2.1, explains how to share subprograms (procedures) between two or more FORTRAN tasks running under DX10.

The other advantage inherent in reentrant programming is the ability for a subprogram to call itself directly or indirectly. This technique is known as recursion and has many useful applications in computer programming. In the case of direct recursion (a subprogram which calls itself), the special calling sequence is automatically generated if the compiler R option is in effect. If the R option is not in effect, recursive calls generate an error. In order to properly call a recursive subprogram from another program module, the REENTRANT statement must be used to declare the special calling sequence for that subprogram. Refer to Appendix E, paragraph E.3.6 for information about using recursive FORTRAN subprograms.

SECTION VII

FORTRAN LIBRARY

7.1 GENERAL

The FORTRAN system supplies a library of standard functions that may be referenced from any program.' These functions are divided into two sets, basic external functions and intrinsic functions. The basic external functions are called by the object program in the same manner as normal, user-supplied functions. Table 7-1 lists and defines these functions. Intrinsic function names are known by the compiler and intrinsic function references may be treated in nonstandard ways (such as inline function expansion). The programmer can replace any of the intrinsic functions with his own function by including the name of the new function in EXTERNAL statements in all calling programs. Table 7-2 lists and defines the intrinsic functions included in the 990 FORTRAN library.

7.2 LIBRARY SUBROUTINES

In addition to the basic external and the intrinsic functions provided in the FORTRAN library, two other subroutines are available to perform buffered input and output. Any program may use these subroutines by referencing them with a CALL statement. Both of these subroutines enable the programmer to implement overlapped I/O. That is, by calling one of these subroutines the program can place the I/O operation under control of the called subroutine and return to program execution while the I/O operation is in progress. If this technique is used, the program must ensure that the previous I/O operation is complete before starting another I/O operation with the same unit. This is done by including an IUNIT test (from the basic external library functions) in the program to check the status of the unit involved. Only when the previous I/O operation is no longer incomplete can the program proceed with the next I/O operation. The following paragraphs describe the buffered I/O subroutines contained in the FORTRAN library.

7.2.1 BUFIN SUBROUTINE. The BUFIN subroutine transmits one physical record from an I/O device to a prescribed buffer area in memory. The subroutine is activated with a CALL statement in the following form:

```
CALL BUFIN(u,m,s,w)
```

where

u is an I/O unit designator

m is an integer mode indicator (0 = ASCII, 1 = binary)

s is a variable name indicating the starting address of the data transfer

w is an integer variable containing the number of bytes to be transferred upon input and the number of bytes actually transferred after an IUNIT test indicates successful completion of the operation.

The subroutine retrieves one physical record from I/O unit number u and stores it in memory beginning at the address s using the format indicated by the mode indicator m. If the value of w is less than the number of bytes in the physical record, only w bytes are transferred and the remaining bytes in the record are ignored. If the value of w is greater than the number of bytes in the physical record, the transmission stops at the end of the physical record.

Table 7-1. FORTRAN Library Basic External Functions

Function	Format	Type of Parameter	Type of Result	Definition
Exponential	EXP (a)	Real	Real	e^a
	DEXP (a)	Double	Double	
	CEXP (a)	Complex	Complex	
Natural Log	ALOG (a)	Real	Real	ln (a)
	DLOG (a)	Double	Double	
	CLOG (a)	Complex	Complex	
Common Log	ALOG10 (a)	Real	Real	$\log_{10}$ (a)
	DLOG10 (a)	Double	Double	
Sine (Angles are in radians.)	SIN (a)	Real	Real	sin (a)
	DSIN (a)	Double	Double	
	CSIN (a)	Complex	Complex	
Cosine (Angles are in radians.)	COS (a)	Real	Real	cos (a)
	DCOS (a)	Double	Double	
	CCOS (a)	Complex	Complex	
Hyperbolic Sine	SINH(a)	Real	Real	sinh(a)
Hyperbolic Cosine	COSH(a)	Real	Real	cosh(a)
Hyperbolic Tan	TANH (a)	Real	Real	tanh (a)
Square Root	SQRT (a)	Real	Real	$(a)^{1/2}$
	DSQRT (a)	Double	Double	
	CSQRT (a)	Complex	Complex	
Arctangent (Angles are in radians.)	ATAN (a)	Real	Real	arctan (a)
	DATAN (a)	Double	Double	
	ATAN2 (a_1,a_2)	Real	Real	arctan (a_1/a_2)
	DATAN2 (a_1,a_2)	Double	Double	
Remaindering (The function DMOD (a_1,a_2) is defined as $a_1 - [a_1/a_2]$ where $[a_1/a_2]$ is the integer whose magnitude does not exceed (a_1/a_2).	DMOD (a_1,a_2)	Double	Double	a_1 (mod a_2)
Absolute Value	CABS (a)	Complex	Complex	Absolute value of complex
	IOR (a_1,a_2)	Integer	Integer	Inclusive OR
	LAND (a_1,a_2)	Integer	Integer	Logical AND
	NOT (a_1)	Integer	Integer	Logical negation
	IEOR (a_1,a_2)	Integer	Integer	Exclusive OR

Table 7-1. FORTRAN Library Basic External Functions (Continued)

Function	Format	Type of Parameter	Type of Result	Definition

NOTE

Boolean functions IOR, LAND, NOT and IEOR are expanded in line.

Function	Format	Type of Parameter	Type of Result	Definition
Bit Manipulation	ISHFT (a_1,a_2)	Integer	Integer	Shift a_1 by a_2 bits (shift left logical if a_2 is positive; shift right logical if a_2 is negative)
	IBTEST (a_1,a_2)	Integer	Logical	Test the a_2 th bit of a_1 (Bits are numbered right to left, 0 to 15.)
	IBSET (a_1,a_2)	Integer	Integer	Set the a_2 th bit of a_1 (Bits are numbered right to left, 0 to 15.)
	IBCLR (a_1,a_2)	Integer	Integer	Clear the a_2 th bit of a_1 (Bits are numbered right to left, 0 to 15.)

NOTE

Bit manipulation functions ISHFT, IBTEST, IBSET and IBCLR will be expanded in line if a_2 is a constant.

Function	Format	Type of Parameter	Type of Result	Definition
Check I/O Status	IUNIT (a)	Integer	Integer	a is an I/O unit designator; Result is: 1 = I/O operation incomplete 2 = I/O operation successful 3 = End of file 4 = Error
Find I/O Error Condition	NERRST (a)	Integer	Integer	Calling NERRST clears the error indicator; a is an I/O unit designator; Result is: 0 = No error 1 = Illegal unit 2 = Illegal operation 3 = I/O error 4 = End of file (Only if the END=S_2 parameter has not been selected; if END=S_2, NERRST returns a no error code, 0. 5 = File previously open 11 = Invalid format 12 = Overflow on input 13 = Field overflow on output 14 = Illegal input character 15 = Numeric or logical mismatch between format and list

Table 7-2. FORTRAN Library Intrinsic Functions

Function	Format	Type of Parameter	Type of Result	Definition
Absolute Value	ABS (a)	Real	Real	|a|
	IABS (a)	Integer	Integer	
	LABS (a)	Extended	Extended	
	DABS (a)	Double	Double	
Truncation	INT (a)	Real	Integer	
	LINT(a)	Real	Extended	
	AINT(a)	Real	Real	
	IDINT(a)	Double	Integer	
	DINT(a)	Double	Extended	
Remaindering	AMOD (a_1,a_2)	Real	Real	a_1 (mod a_2)
	MOD (a_1,a_2)	Integer	Integer	
	LMOD (a_1,a_2)	Extended	Extended	
Choose Largest Value	AMAX0 (a_1,a_2, . . .)	Integer	Real	Max (a_1,a_2, . . .)
	AMAX1 (a_1,a_2, . . .)	Real	Real	
	MAX0 (a_1,a_2, . . .)	Integer	Integer	
	MAX1 (a_1,1_2, . . .)	Real	Integer	
	LMAX0 (a_1,a_2, . . .)	Extended	Extended	
	LMAX1 (a_1,a_2, . . .)	Real	Extended	
	DMAX0 (a_1,a_2, . . .)	Integer	Double	
	DMAX1 (a_1,a_2, . . .)	Double	Double	
Choose Smallest Value	AMIN0 (a_1,a_2, . . .)	Integer	Real	Min (a_1,a_2, . . .)
	AMIN1 (a_1,a_2, . . .)	Real	Real	
	MIN0 (a_1,a_2, . . .)	Integer	Integer	
	MIN1 (a_1,a_2, . . .)	Real	Integer	
	LMIN0 (a_1,a_2, . . .)	Extended	Extended	
	LMIN1 (a_1,a_2, . . .)	Real	Extended	
	DMIN0 (a_1 , a_2 . . .)	Integer	Double	
	DMIN1 (a_1,a_2, . . .)	Double	Double	
Type Conversion	FLOAT (a)	Integer	Real	Conversion from Integer to Real
	LFLOAT(a)	Extended	Real	Conversion from Extended Integer to Real
	IFIX (a)	Real	Integer	Conversion from Real to Integer
	LFIX (a)	Real	Extended	Conversion from Real to Extended Integer
	SNGL (a)	Double	Real	Conversion from Double to Real
	DBLE (a)	Real	Double	Conversion from Real to Double
	CMPLX (a_1,a_2)	Real	Complex	$a_1 + a_2 \sqrt{-1}$
Transfer of Sign	SIGN (a_1,a_2)	Real	Real	Sign of a_2 times $\|a_1\|$
	ISIGN (a_1,a_2)	Integer	Integer	
	LSIGN (a_1,a_2)	Extended	Extended	
	DSIGN (a_1,a_2)	Double	Double	
Positive Difference	DIM (a_1,a_2)	Real	Real	a_1–min (a_1,a_2)
	IDIM (a_1,a_2)	Integer	Integer	
	LDIM (a_1,a_2)	Extended	Extended	

Table 7-2. FORTRAN Library Intrinsic Functions (Continued)

Function	Format	Type of Parameter	Type of Result	Definition
Complex Number Manipulation	CONJG (a)	Complex	Complex	Obtain Conjugate of Complex
	REAL (a)	Complex	Real	Obtain Real Part of Complex
	AIMAG (a)	Complex	Real	Obtain Imaginary Part of Complex

The status of I/O unit number u must be checked with an IUNIT test before proceeding with the operation. When the IUNIT test indicates the BUFIN operation is complete, the number of bytes read is returned in W.

Example:

```
      DIMENSION IARRAY(40)
      REWIND 15
      DO 20 I=1, 100
          CALL BUFIN(15,1,IARRAY,ICHAR=80)
               •
               • (Any processing not involving UNIT 15 or IARRAY)
               •
10        ISTAT = IUNIT(15)
          IF (ISTAT .EQ. 1) GO TO 10
          IF (ISTAT .NE. 2) STOP
               •
               •
               •
20        CONTINUE
      END
```

7.2.2 **BUFOUT SUBROUTINE.** The BUFOUT subroutine transfers one physical record from a buffer area in memory to a specified I/O device. The subroutine is activated with a CALL statement in the following form:

```
CALL BUFOUT(u,m,s,w)
```

where:

u is an I/O unit designator

m is an integer that indicates transfer mode (0 = ASCII, 1 = binary)

s is a variable name indicating the starting address of the data transfer

w is the number of bytes to be transferred.

The subroutine transfers w bytes from the buffer area in memory beginning at location s to I/O unit number u. The data is written in one physical record using the format indicated by the mode indicator m.

The status of I/O unit number u must be checked with a IUNIT test before proceeding with the operation.

NOTE

> The file must be terminated with an END FILE statement before the I/O unit may be repositioned.

Example:

```
      DIMENSION IARRAY(40)
      REWIND 15
      DO 20 I=1,100
           .
           .
           .
         ICHAR=80
         CALL BUFOUT (15,1,IARRAY,80)
           .
           . (Any processing not involving UNIT 15 or IARRAY)
           .
10       ISTAT = IUNIT(15)
         IF (ISTAT.EQ.1) GO TO 10
         IF (ISTAT.NE.2) STOP
20       CONTINUE
      ENDFILE 15
      END
```

7.3 ISA EXTENSIONS

The FORTRAN library also includes a set of subroutines that satisfy the extensions to the FORTRAN language recommended by the Instruments Society of America (ISA Extensions, S61.1-1975 and S61.2-1976). These subroutines permit interface with executive programs, process input and output functions, provide access to time and date information, and provide procedures for file access and control of file contention. The following paragraphs describe the function and calling sequence of each of the ISA Extension functions provide in the 990 FORTRAN library.

NOTE

> To reduce the size of a program link, an ISA subroutine may be declared an integer (example: INTEGER*2 OCRU) to prevent the FORTRAN RUNTIME real arithmetic package from being linked in.

7.3.1 START A PROGRAM. The START subroutine allows the user to specify a specific time delay before beginning a specified task. On DX10, the specified task must be installed nonreplicatable on the system program file. The form of the call to this subroutine is

```
CALL START (task, delay, units, code)
```

The required parameters for this call are defined as follows:

- task – the name of a 3-element INTEGER*2 array. The first element contains the task id of the task to be started. The last two elements contain the four bytes of task parameters, i.e., terminal I/O, data etc.

- delay – an I*2 integer indicating the number of units that the task will be delayed before starting. A negative or zero value results in an immediate start.

- units – an I*2 integer, either 0, 1, 2 or 3 that specifies the units to be used in measuring the time delay. The digits indicate the following units:

  0 = 8.3 millisecond units (system clock)

  1 = 1 millisecond units

  2 = 1 second units

  3 = 1 minute units

- code – an I*2 variable to accept a return code from the subroutine that indicates the disposition of the request as follows:

  1 = Request accepted

  2 = The specified task id does not exist

  3 = Illegal time

  4 = The system is unable to accept the request

  5 = Error

NOTE

Clock resolution is limited to one second intervals.

7.3.2 START A PROGRAM AT A SPECIFIED TIME. The TRNON subroutine allows the user to specify a specific time of day at which to start execution of a specified task. The specified task must be installed nonreplicatable on the system program file. The form of the call to this subroutine is

```
CALL TRNON (task, time, code)
```

The required parameters for this call are defined as follows:

- task – the name of a 3-element INTEGER*2 array. The first element contains the task id of the task to be started. The last two elements contain the four bytes of task parameters.

- time – the name of an INTEGER*2 array. The first three elements of the array contain the hour, minute, and second values that define the military (24-hour) time at which the program will be started.

- code – an argument to accept a return code from the subroutine that indicates the disposition of the request as follows:

  1 = Request accepted

  2 = The specified task ID does not exist

  3 = Illegal time

  4 = The system is unable to accept the request

NOTE

The TRNON subroutine is not supported on the TX(990/4) system.

7.3.3 DELAY CONTINUATION OF A PROGRAM. The WAIT subroutine allows the user to specify a specific time delay before continuing with the execution of a program sequence. The form of the call to this subroutine is

```
CALL WAIT (delay, units, code)
```

The required parameters for this call are defined as follows:

- delay – an I*2 integer indicating the number of units that the program sequence will be delayed. A negative or zero value results in no program delay.

- units – an I*2 integer, either 0, 1, 2 or 3, that specifies the units to be used in measuring the time delay. The digits indicate the following units:

  0 = 8.3 millisecond units (system clock)

  1 = 1 millisecond units (delay is computed in 50 millisecond intervals)

  2 = 1 second units

  3 = 1 minute units

- code – an I*2 variable to accept a return code from the subroutine that indicates the disposition of the request as follows:

  1 = Request accepted

  2 = Delay too long (maximum delay = 1638.375 seconds)

  3 = Illegal unit specification

7.3.4 DIGITAL INPUT. The DIW subroutine allows the user to input sets of bits from the CRU interface and store those bits in a specified array. The form of the call to this subroutine is

```
CALL DIW (number, specification, target, code)
```

The required parameters for this call are defined as follows:

- number – an I*2 integer expression indicating the number of bit groups to be input from the CRU (must be between 1-16).
- specification – a 2 by "number" I*2 integer array. The first element of each doublet is the decimal CRU base as it would be used in Register 12, and the second element is the number of bits to transfer.
- target – an I*2 array which is "number" elements long into which bit group are store right justified.
- code – an I*2 variable to accept a return code from the subroutine that indicates the disposition of the request as follows:

  1 = All data collected

  2 = (This number is not used)

  3 = The "number" parameter is specified as zero

  4 = The CRU address is out of the valid range

  5 = The number of bits are not in the valid range of 1 to 16 bits.

7.3.5 LATCHED DIGITAL OUTPUT. The DOLW subroutine allows the user to output bits of information to the CRU. The bits can be latched in either the set or the reset condition. The form of the call to this subroutine is

```
CALL DOLW (number, specification, source, mask, code)
```

The required parameters for this call are defined as follows:

- number – an I*2 integer expression indicating the number of bit groups to be output to the CRU (must be between 1-16).
- specification – a 2 by "number" I*2 integer array. The first element of each doublet is the decimal CRU base as it would be used in Register 12, and the second element is the number of bits to transfer.
- source – an I*2 array from which the output bit groups are fetched. The data in the array must be right justified.
- mask – the library subroutine ignores this parameter. It must be provided for compatibility with other ISA libraries.

NOTE

CRU input and output data is bit-reversed. For a more detailed description of the CRU refer to paragraph 3.89.8 in *Model 990 Computer Assembly Language Programmer's Guide*, part number 943441-9701.

- code – an I*2 variable to accept a return code from the subroutine that indicates the disposition of the request as follows:

  1 = Request complete

  2 = (This number is not used)

  3 = The "number" parameter is specified as zero

  4 = The CRU address is out of the range

  5 = The number of bits are not in the valid range of 1 to 16 bits

7.3.6 MOMENTARY DIGITAL OUTPUT. The DOMW subroutine allows the user to output bit groups to the CRU. The bit groups consist of momentary digital output signals. All specified bits are reset after a delay of "time". The form of the call to this subroutine is

```
CALL DOMW (number, specification, source, time, code)
```

The required parameters, except "time" have the same definitions as those for the Latched Digital Output subroutine, DOLW. Time specifies a time delay before continuing the execution of a program sequence. The parameter time is a 3-dimensioned array whose elements correspond to the parameters of the WAIT routine (see paragraph 7.3.3).

7.3.7 OBTAIN DATE. The DATE subroutine allows the user to determine the correct calendar date. The form of the call to this subroutine is

```
CALL DATE (i)
```

The required argument i is an INTEGER*2 array containing three elements. The first element contains an integer representation of the year (A.D.), the second element contains an integer representation of the month, and the third element contains an integer representation of the day.

7.3.8 OBTAIN TIME. The TIME subroutine allows the user to determine the correct time of day (if the system time and date was initialized correctly). The form of the call to this subroutine is

```
CALL TIME (j)
```

The required argument j is an INTEGER*2 array containing three elements. The first element contains the hour, the second element contains the minute and the third element contains the second.

7.3.9 ANALOG DATA HANDLING. The 990 Computer family provides for analog data handling by an optimal analog-to-digital conversion module and an optimal digital-to-analog conversion module. These hardware modules can be controlled from FORTRAN by the following ISA subroutines:

Subroutine	Purpose
AISQW	Analog Input in Sequence
AIRDW	Analog Input in Random
AOW	Analog Output

The AISQW subroutine samples points in an order determined by the A/D hardware module. Each call samples consecutive channels beginning with channel 0 on a single A/D module. A maximum of 64 channels can be sampled since an A/D module contains no more than 64 channels. The subroutine samples a channel and stores a 12-bit integer in the least significant bits of the input vector; any scaling of values is left to the user. The choice of voltage or current range and code scheme (straight binary or two's complement) is a hardware switch selectable option left to the user.

The AIRDW subroutine samples points in an order determined by the user. Each call samples any number of channels from any A/D module sequence in any order. The user specifies the number of sample points desired and provides an I*2 vector to receive the converted values. In addition, each point must be tagged with a CRU A/D module address and a channel number. The subroutine samples the specified channel on the specified module and stores a 12-bit integer in the 12 least significant bits of the input vector; any scaling of values is left to the user. The choice of voltage or current range and code scheme (straight binary or two's complement) is a hardware switch option left to the user. If two's complement is switch selected, the converted value is not sign extended to 16 bits; it remains a 12-bit positive integer. To sign extend the value, the following statements can be used:

```
IF(IBTEST(IVALUE,11))IVALUE=IOR(IVALUE,4ZF000)
```

where IVALUE contains the 12-bit value.

The AOW subroutine converts digital values in an order specified by the user. The 990 Digital-To-Analog (D/A) hardware module can be ordered with 1 to 4 channels. The digital interface is common to all channels so the same digital value can be sent to any combination of the 4 channels. Each call converts any number of values through any combination of D/A modules using any combination of channels. The user specifies the number of values to convert and provides an I*2 vector containing the values. Each value must be tagged with a CRU D/A module address and a set of channel enable bits. The subroutine transfers the 12 least significant bits of a value to the D/A hardware module and channels. All scaling is left to the user. The choice of voltage or current range and code scheme is a hardware switch option left to the user.

All three subroutines have a similar calling sequence. The form of the call for the AISQW subroutine is:

```
CALL AISQW(points, CRU addr, input vector, code)
```

where:

points — is an I*2 expression for the number of conversions to perform.

CRU addr — is the CRU address for the chassis slot containing the A/D module. This is the value stamped on the chassis for the slot.

input vector — is an I*2 vector to receive the converted values.

code — is an I*2 variable to receive the completion code. Where the completion code:

1 – Normal completion

2 – Operation incomplete (hardware failed to respond)

3 – CRU address out of range

The form of the call for the AIRDW subroutine is:

CALL AIRDW (points, addr & chan # vector, input vector, code)

Where:

points is an I*2 expression for the number of conversions to perform.

addr & chan # vector are a two dimensional I*2 vector used to tag the source for each conversion. The dimensions should be 2 and the number of points. For each two word entry, the first word contains the A/D module CRU address and the record contains the channel number to use on the A/D module. In the example, point J, the entry is:

ADDR (1, J) = CRU address of A/D module
ADDR (2, J) = Channel number

Note that channel numbers are in the range 0 to 63.

input vector is an I*2 vector to receive the converted values.

code is an I*2 variable to receive the completion code. Where the completion code:

1 = Normal completion

2 = Operation incomplete (hardware failed to respond)

3 = CRU address out of range

4 = Channel number out of range

The form of the call for the AOW subroutine is:

CALL AOW (<points>, <addr & chan enable>, <output vector>, <code>)

where:

points is an I*2 expression for the number of conversions to perform.

addr & chan enable are a two dimensional I*2 vector used to tag the destination for each conversion. The dimensions should be 2 and the number of points. For each two word entry, the first word contains the D/A CRU module address and the second contains the channel enable mask. The channel enable mask is stored in the 4 least significant bits of the array word. The bits are defined as:

Bit Number	Purpose
15	1 = Enable conversion on channel 0 0 = Inhibit conversion on channel 0
14	1 = Enable conversion on channel 1 0 = Inhibit conversion on channel 1
13	1 = Enable conversion on channel 2 0 = Inhibit conversion on channel 2
12	1 = Enable conversion on channel 3 0 = Inhibit conversion on channel 3

output vector is an I*2 vector containing the digital values to convert.

code is an I*2 variable to receive the completion code. Where the completion code:

1 = Normal completion

2 = Operation incomplete

3 = CRU address out of range

7.3.10 **CFILW SUBROUTINE.** The CFILW subroutine allows the user to create a specific file. The form of this call is:

```
Call CFILW (CFILE, RECLNGTH, RECNUM, AB1)
```

The required parameters for this call are defined as follows:

- CFILE – An I*2 array that contains three items. The user must dimension this array such that its length will accommodate all three items.

  These items are:

  1) File type

  2) Number of characters in the pathname

  3) Pathname (2 characters per element)

The first element is the file type:

01 = Sequential file

02 = Relative record file

The second element contains the number of characters in the pathname, and the remaining elements contain the pathname.
Example:

```
Dimension FNAM(6), CFILE(8)
EQUIVALENCE (FNAM(1), CFILE(3))
DATA CFILE(1) /01/, CFILE(2) /12/
DATA FNAM /'SYS00312.TST'/
```

- RECLNGTH – Record length and even numbers greater than >1F, that specifies the number of characters in a record.
- RECNUM – An I*4 datum which indicates the maximum number of records in the file.
- AB1 – Error code:

  1 – Normal completion

  2 or greater – Signals an error. Proper reference is to the normal system error codes.

NOTE

When attempting to debug a program, it is important to remember that 990 FORTRAN generates its own internal ID's for the file control blocks to use when referencing LUNO's assigned in the ISA extensions.

7.3.11 DFILW SUBROUTINE. The DFILW subroutine allows the user to delete a specified file. The form of this call is:

CALL DFILW (DFILE, AB1)

The required parameters for this call are defined as follows:

- DFILE – is an I*2 array. The first element of the array contains the number of characters in the pathname, while the remaining elements contain the pathname.

  See 7.3.10 for an example.

- AB1 – Error code:

  1 – Normal completion

  2 or greater – Signals an error. Proper reference is to the normal system error codes.

7.3.12 OPENW/CLOSEW SUBROUTINE. The subroutine CALL OPENW/CALL CLOSEW allows for the opening and closing of a LUNO with an associated file name. A call to OPENW does not create a nonexistent file. Also, note that once you specify the parameters in the first call to the subroutines, the same number of parameters must be used in subsequent calls. The form of these calls are:

```
CALL OPENW (LUNO, FILENAME, ACCESS, AB1)
CALL CLOSEW (LUNO, AB1)
```

The required parameters for this call are defined as follows:

- LUNO – is the LUNO assigned.
- FILENAME – is an array name, the first word of which must contain the character length of the pathname or device name; the remainder of the array contains the pathname or device name itself.
- ACCESS – specifies the access method:

  1 – Read only

  2 – Shared

  3 – Exclusive write

  4 – Exclusive all
- AB1 – Error code:

  1 – Normal completion

  2 or greater – Signals an error. The error may be a normal system error code or one of the following special error codes:

  >40 – LUNO requested is invalid

  >41 – File previously opened and not closed

  >42 – FCB table full

  >43 – Access method requested is invalid

NOTE

The parameter AB1 may be omitted and automatic error termination with traceback will result if an error occurs. If AB1 is not omitted, it is the user's responsibility to code an error-handling routine to handle values returned in AB1. Note also that a file must have been opened previously with ISA OPENW before it can be closed or automatic error termination with traceback will result.

NOTE

When attempting to debug a program, it is important to remember that 990 FORTRAN generates its own internal ID's for the file control blocks to use when referencing LUNO's assigned in the ISA extensions.

7.3.13 RDRW/WRTRW SUBROUTINE. This subroutine allows for relative record read and write. The form for this call is:

CALL RDRW (LUN, RECNUM, BUF, WRDCNT, AB1)

CALL WRTRW (LUN, RECNUM, BUF, WRDCNT, AB1)

The required parameters for this call are defined as follows:

- LUN – is the LUNO assigned
- RECNUM – specifies the record number to be read/written (AN INTEGER*4 DATUM).
- BUF – designates the first variable into which information is to be read/written.
- WRDCNT – specifies the maximum number of integers to be read/written.
- AB1 – error code:

  1 – Normal completion

  2 or greater – Signals an error. The error may be a normal system error code or the following special error code:

  > 40 – LUNO requested invalid

NOTE

The parameter AB1 may be omitted and automatic error termination with traceback results if error occurs.

NOTE

The file to be read or written must have been previously opened by the ISA OPEN (OPENW) and must be closed by the ISA CLOSE (CLOSEW).

NOTE

When attempting to debug a program, it is important to remember that 990 FORTRAN generates its own internal ID's for the file control blocks to use when referencing LUNO's assigned in the ISA extensions.

7.3.14 SVCFUT SUBROUTINE.

NOTE

The SVCFUT subroutine is not supported on ~~TXDS or~~ DX10 2.X releases.

The SVCFUT subroutine allows the user to set up the SVC call block, with each SVC Block (I) equaling one word of the block. The form of this call is:

```
CALL SVCFUT (OPCDE, TYPE, LUN, VALID, RECLNGTH, FILANDISC, NUMREC, ALLOCATION, AB1)
```

The required parameters for this call are defined as follows:

- OPCDE – is the utility operation code
- TYPE – is the file type
- LUN – is the LUNO to be assigned or released
- VALID – is the validation identifier (not implemented in current operating system. The user must supply a dummy value.)
- RECLNGTH – is the record length. An even number greater than 1F (base 16) that specifies the number of characters in the record.
- FILANDISC – is the name of an INTEGER*2 array that holds the file pathname. The first element of this array contains the number of characters in the pathname, and the remaining elements contain the pathname.
- NUMERC – is an INTEGER*4 variable for the maximum number of records in the file.
- ALLOCATION – specifies the allocation. Contiguous (0) and Noncontiguous (1).
- AB1 – is an INTEGER*2 datum into which the RTND value of control byte AB1 (error code) is placed.

NOTE

When attempting to debug a program, it is important to remember that 990 FORTRAN generates its own internal ID's for the file control blocks to use when referencing LUNO's assigned in the ISA extensions.

7.3.15 MODAPW SUBROUTINE. The subroutine allows the user to change the access method of a previously opened file. The form of the call is:

CALL MODAPW (LUNO, ACCESS, AB1)

The required parameters for this call are defined as follows:

- LUNO is the LUNO assigned.
- ACCESS specifies the access privilege:

  1 = Read only

  2 = Shared

  3 = Exclusive write

  4 = Exclusive all

NOTE

When using a TX system, only SHARED and EXCLUSIVE ALL access methods are allowed. Therefore, ACCESS = 2 and ACCESS = 4 are allowed on TX systems.

- AB1 is the error code:

  1 – Normal completion

  2 or greater – signals an error. The error may be a normal system error code or the following 40– LUNO requested invalid

NOTE

The parameter AB1 may be omitted and automatic error termination with traceback results if an error occurs.

NOTE

When attempting to debug a program, it is important to remember that 990 FORTRAN generates its own internal ID's for the file control blocks to use when referencing LUNO's assigned in the ISA extensions.

7.4 990 FORTRAN CALLABLE SUBROUTINES

In addition to the standard FORTRAN library subroutines and the ISA extension subroutines, the 990 FORTRAN library provides subroutines for additional user convenience. These functions include a random number generator, additional program control calls, various forms of time and date information, and CRU I/O calls. The following paragraphs explain the function and format of these additional library members.

7.4.1 PSEUDO-RANDOM NUMBER. The library function, RANF, provides the user with the ability to access a number generated by a pseudo-random number generator algorithm. The function can be used in an assignment statement, such as

```
X = RANF (d)
```

The argument d is a dummy argument that forces the compiler to recognize RANF as a function. The result of this statement is the assignment of a pseudo-random number to the variable X. The random number provided is a REAL*4 number in the range of 0 to +1.

7.4.2 PRESET RANDOM NUMBER GENERATOR. The library subroutine, RANSET, allows the user to preset the seed value of the random number generator to any desired value. The form of the call for this subroutine is

```
CALL RANSET (x)
```

The argument x can be any integer value, and designates the value to preset the random number generator.

7.4.3 GENERATE SUPERVISOR CALL. The library subroutine, SVC, allows the user to generate a DX10 supervisor call by providing the SVC control block. The form of the call for this subroutine is

```
CALL SVC (svc block, ab1, auto abort flag, register block)
```

The required arguments are defined as follows:

- svc block – the name of the array containing the SVC control block.
- ab1 – an INTEGER*2 variable into which the SVC returns a control byte (usually an error code).
- auto abort flag – a LOGICAL input variable that when true allows errors to abort task. When the call is aborted, argument ab1 contains a nonzero value and the operating system initiates a traceback.
- register block – the name of a 10-word array that the SVC uses for pseudo-registers.

7.4.4 ABSOLUTE ADDRESS. The library function, LOC, allows the user to access the absolute address of a single argument. The function can be invoked with an assignment statement as follows

```
I = LOC (B)
```

The single argument, B, specifies the parameter whose absolute address will be assigned to the variable I. The returned value is INTEGER*2.

7.4.5 BID TASK. The library subroutine, BIDTSK, allows the user to issue a DX10 Bid Task SVC. The specified task must be installed nonreplicatable on the system program file. The form of the call to this subroutine is

CALL BIDTSK (task, abl)

The argument, task, is a 3-element INTEGER*2 array. The first element contains the task id; the other two elements contain task parameters. The argument, abl, is an INTEGER*2 variable into which the SVC returns the task state upon completion of the call.

7.4.6 DELAYED BID TASK. The library subroutine, DLYBID, allows the user to issue a DX10 Delayed Bid Task SVC. The specified task must be installed nonreplicatable on the system program file. The form of the call to this subroutine is

CALL DLYBID (task, date, abl)

The required arguments for this call are defined as follows:

- task – a 3-element INTEGER*2 array. The first element contains the task id; the other two elements contain task parameters.
- date – a 5-element INTEGER*2 array that contains the date and time at which to start the task. Element 1 contains the binary value of the year; element 2 contains the binary value of the day, element 3 contains the binary value of the hour, element 4 contains the binary value of the minute; element 5 contains the binary value of the second.
- abl – an INTEGER*2 variable that receives the completion code from the SVC upon completion.

7.4.7 CRU INPUT. The library function ICRU allows the user to read from one to 16 input CRU lines. The function can be used in an assignment statement, such as

I = ICRU (cru base, number lines)

The required arguments are defined as follows:

- cru base – the base address to be used in the data transfer as it should appear in workspace register 12.
- number lines – the number of contiguous CRU lines to be transferred.

The value read is then assigned to the variable I, right justified.

7.4.8 CRU OUTPUT. The library subroutine, OCRU, allows the user to output data to one to 16 output CRU lines. The subroutine call appears in the following form:

CALL OCRU (cru base, number lines, value)

The required arguments are defined as follows:

- cru base – the base address to be used in the data transfer as it should appear in workspace register 12.
- number lines – the number of contiguous CRU lines to be transferred.
- value – a literal value, expression or other legal argument whose value (right justified) will be output to the selected CRU lines.

7.4.9 OBTAIN DATE AND TIME. The library subroutine, DATIME, allows the user to fetch the binary values of the date and time parameters supplied by the operating system. The form of the call for this subroutine is

CALL DATIME (time)

The required argument, time, is a 5-element INTEGER*2 array. The first element receives the binary value for the year, the second element receives the binary value for the day, the third element receives the binary value for the hour, the fourth element receives the binary value for the minute, and the fifth element receives the binary value for the second.

7.4.10 OBTAIN ASCII DATE. The library subroutine ADATE allows the user to access the date information supplied by the operating system and store the values as ASCII characters. The form of the call for this subroutine is

CALL ADATE (I)

The required argument I is a 3-element INTEGER*2 array. The first element receives two ASCII numbers representing the month (01 - 12), the second element receives two ASCII numbers representing the day (01 - 31), and the third element receives two ASCII numbers representing the year.

7.4.11 OBTAIN ASCII TIME. The library subroutine ATIME allows the user to access the time information supplied by the operating system and store the values as ASCII characters. The form of the call for this subroutine is

CALL ATIME (J)

The required argument J is a 3-element INTEGER*2 array. The first element receives two ASCII numbers representing the hour (01 - 24), the second element receives two ASCII numbers representing the minute (00 - 59), and the third element receives two ASCII numbers representing the seconds (00 - 59).

7.4.12 OBTAIN MILITARY DATE. The library subroutine MDATE allows the user to access the date information supplied by the operating system and store the values in military date format, e.g., 04 JUL 76. The routine receives the information in integer format and reformats it into ASCII characters that correspond to the required format. The form of the call for this subroutine is

CALL MDATE (I)

The required argument I is a 4-element INTEGER*2 array. The first element receives two ASCII numbers representing the day (01 - 31), the second and third elements receive the three ASCII characters that are the first three letters of the month, and the fourth element receives two ASCII numbers representing the year.

7.5 FORTRAN-PROM PROGRAM SUBROUTINES

NOTE

> The package described in paragraph 7.5 is supported only by the DX10 3.X and TX990 2.2 releases.

The FORTRAN-PROM Programmer Subroutine package contains three FORTRAN callable assembly subroutines that provide software interface capability between a user written FORTRAN program, a FORTRAN accessible DX10 or TXDS file, and a PROM programmer module. This interface capability allows the FORTRAN user to write and read PROMs via the PROM programming module by calling the subroutines to perform the Input/Output functions that exceed standard FORTRAN capability. These routines may be used with either the erasable (EPROM) PROM device or bipolar devices.

7.5.1 LIMITATIONS. The FORTRAN-PROM Programmer Subroutine package is designed to transfer object programs to PROM or EPROM. It is 16 bit, record oriented, not 8 bit byte or 4 bit nibble oriented, and is not intended to be a general purpose PROM programmer. For example, the programmer subroutine, PRGROM, transfers no more than 8 bits from a memory array word to the PROM. When programming a data table with 8 bit entries in 2708 EPROM, every other byte in the memory image array may *not* be used since only 8 bits of a memory image array location are transferred to a corresponding EPROM location.

7.5.2 SUBROUTINE PACKAGE MODULES. Each of the three PROM programmer modules is a 990 assembly code subroutine and must be called using the standard FORTRAN linkage. Arguments required by these modules are described under user software interface (see paragraph 7.5.3). The PROM programming module is software driven via status lines and interrupts are not used.

7.5.2.1 IMGBLD Module. IMGBLD module takes the object file output by SDSMAC assembler and builds an absolute image of the object file by relocating all relocatable objects by the user specified bias. The absolute image is stored in a user defined array and each entry is a 16-bit word.

7.5.2.2 PRGROM Module. PRGROM module writes a bit string contained within a user selected memory image array element into the PROM at an address, passed by the user as a call argument, using PROM Programmer characteristics also passed by the user.

7.5.2.3 RDPROM Module. RDPROM module reads a PROM address defined by a caller argument using a physical word bit width also defined by the caller. The string is returned in a call argument designated memory image array element where it replaces bits starting with a bit position passed as an argument.

7.5.3 USER FORTRAN ROUTINE. The FORTRAN user calling the PROM Programmer Subroutines must provide storage allocation, PROM characteristics, and PROM and memory addressing capability. Additionally, he must provide FORTRAN input logic for object record input from the object file. The PROM physical characteristics are dependent on the specific PROM and the user normally uses those characteristic values furnished by the manufacturer.

7.5.3.1 Inputting the PROM Program From Mass Storage. The user's PROM program relocatable object must reside on a file that may be accessed with FORTRAN Input/Output statements. The user assigns a FORTRAN unit number to this file and reads it sequentially. As records are read by the FORTRAN main program, the subroutine IMGBLD is called, one call per standard object record, to build a biased memory image in the memory image array. The data flow is shown in figure 7-1. All call arguments are of the INTEGER*2 type.

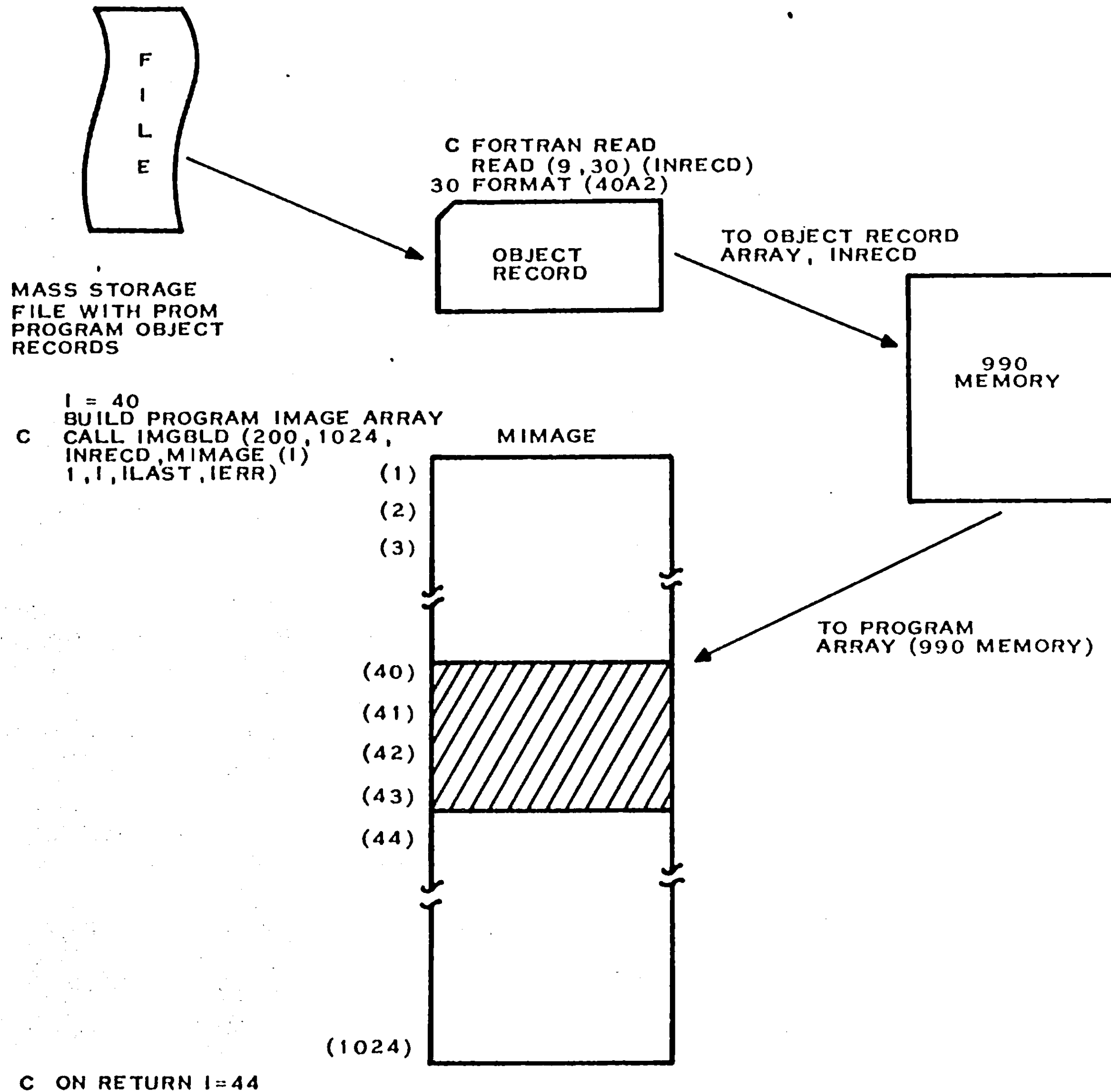


Figure 7-1. IMGBLD Data Flow

Example:

CALL IMGBLD (200,1024,INRECD,MIMAGE(I),I,ILAST,IERR)

where:

200 — is the PROM program load bias.

1024 — is the dimension value of the memory image array. If this value is less than the actual length of the PROM program, the user is responsible for array overlay management. Special care should be taken in that IMGBLD will terminate on array overflow conditions and data of the current record must be reprocessed once array space is available to prevent data loss. Also the array index must be reinitialized.

INRECD is the user integer array containing the object record to be loaded into the memory image array designated by MIMAGE(I).

MIMAGE(I) is the memory image array. It is a one dimensional integer array declared in the users FORTRAN program for the purpose of storing the loaded and biased PROM program in the 990 memory.

I is the memory image array index. When IMGBLD is called to load an object record the value of this argument gives the starting array element for the current record object. On return the value of argument 5 is the index for the next sequential unoccupied array element.

ILAST is a variable initialized by the caller to -1. Upon return, IMGBLD passes the final array index value when the end of file is encountered. This index value is the word length of the PROM program if the array has not been overlaid.

IERR contains an error code if IMGBLD encounters an error condition. See table 7-3 for error codes.

7.5.3.2 Writing the PROM. Once the user memory image array contains the biased data to be transmitted to the PROM, the user's FORTRAN program must successively select an array element and a bit string within the 16 bit array element, and pass this information with the appropriate PROM bit address to the subroutine PRGROM. The number of successive bits that can be physically written by any one call to PRGROM may vary from one physical PROM word to an entire PROM. The PROM physical characteristics must be passed as arguments with each call.

Table 7-3. Error Table

Code	Error Condition	Module
0	No error	All
2	Absolute module error	IMGBLD
3	Data array overflow error	IMGBLD
4	Checksum error	IMGBLD
5	Non-ASCII character error	IMGBLD
12	Hardware not online	PRGROM
13	Duty cycle out of range (1-100)	PRGROM
14	Pulse width index out of range (1-32)	PRGROM
15	EPROM parm 3 must equal parm 4	PRGROM
16	EPROM retry count must be 0	PRGROM
17	EPROM duty cycle must be 100	PRGROM
18	Number of bits (parm 3) out of range (1-8)	PRGROM
19	PROM Programmer hardware error	PRGROM
22	Hardware not online	RDPROM

Example:

CALL PRGROM (IERR, INCAUT, IPBITS, IPWORD 1, MIMAGE(I), MDISP, IPPDSP, LOWHI, IDUTY 2, IRETRY, ICRU, IPULSE)

where:

IERR	returns an error code if PRGROM encounters an error condition. See table 7-1 for error codes.
INCAUT	is initialized by the caller with the PROM physical word count to be written with the call. A count of one is automatic if the user passes a value less than or equal to zero. A count equal to the PROM word size will cause the complete PROM to be written with one subroutine call and reduce processing time per PROM significantly. In the event an error occurs before the count is completed, the subroutine returns the uncompleted count value in INCAUT and the error code in argument 1.
IPBITS	is the number of PROM bits that can concurrently be written. This value is defined by the manufacturer, but generally is equal to one for bipolar PROMs and is equal to the PROM physical word size for EPROMs.
IPWORD1	is the number of bits in the PROM physical word.
MIMAGE(I)	is the memory image array element (16 bits) from which a maximum of 8 consecutive data bits will be taken and then written into the PROM.
MDISP	is the integer bit number (0-15) indicating the start bit within the memory image, array element (MIMAGE(I)).
IPPDSP	is the PROM physical word address for the PROM write operation. If INCAUT is greater than one, this address will be automatically incremented and written until INCAUT is exhausted.
LOWHI	is an integer 0 for low PROM logic programming (i.e. PROM bits are all one value initially) and 1 for high PROM logic programming (i.e. PROM bits are all zero values initially).
IDUTY2	is the PROM duty cycle which is less than or equal to, the maximum duty cycle as given in the PROM characteristics with percent sign removed.
IRETRY	is the number of physical attempts allowed to write a PROM word in the event of an error condition. This argument must be zero for EPROMs because of the requirement that they be programmed sequentially and repetitively.

ICRU — is the integer CRU base address of the PROM Programming Module. This address is found on the CRU chassis.

IPULSE — is an integer index which determines the hardware write pulse width.

Pulse Index	Pulse Width (Millisec)
1	.5
2	1.
4	2.
8	4.
16	8.
32	16.

7.5.3.3 READING THE PROM

To retrieve data from PROM, the users FORTRAN program must successively select a physical PROM word address, a memory array element, and an element bit start address to be passed to the subroutine RDPPROM. The user's FORTRAN program must also provide any listing logic required after the information is transferred from the PROM to the integer memory image array.

Example:

```
Call RDPROM (IERR, INCAUT, IPWORD, MDISP, IPPDSP 1, ICRU, MIMAGE(I))
```

where:

IERR — contains an error code in the event of an error condition. The error codes and corresponding conditions are found in table 7-3.

INCAUT — contains the number of PROM physical words to be read with the current call. When greater than one, successive PROM locations are addressed and read until the count is exhausted or an error condition exists. The error condition code returned in IERR and INCAUT contains the number of unread words remaining.

IPWORD — is the number of bits in the PROM physical word.

MDISP — is the bit number (0-15) indicating the start bit within the memory array element for the PROM data read by the call.

IPPDSP 1 — is the PROM word physical address from which data is to be read.

ICRU — is the CRU base address of the PROM Programming Module.

MIMAGE(I) — is the integer memory image array element into which the read PROM data is placed, starting with the bit denoted in MDISP.

7.5.4 SAMPLE PROGRAM DESCRIPTIONS. Figures 7-2 through 7-5 contain four sample PROM program subroutines employed in routines designed to acquaint the user with their structure and application.

Figure 7-2 is a sample program for reading a 2708 EPROM into a FORTRAN array in main memory.

Figure 7-3 is a sample program for writing all zeros to a 2708 EPROM and then reading the EPROM to verify the write.

Figure 7-4 is a sample program for writing a bipolar S287 PROM to all ones and for reading the PROM after the write to verify the write.

Figure 7-5 is a sample program to read an object module, relocate it and program it into 4 bipolar S287 PROMs.

```
C
C READ ANY PROM ATTACHED TO THE PROM PROGRAMMER
C
      DIMENSION MIMAGE(1024)
C
C FOLLOWING ARE STATIC PARAMETERS USED TO READ A PROM
C
      IWDCNT=1024
      IPRMWD=8
      ISTBIT=8
      ICRU=160
C
C CLEAR MEMORY WHERE PROM IS TO WRITE FOR EASY CHECKING
C
      DO 10 I=1,1024
        MIMAGE(I)=0
10    CONTINUE
C
C READ ENTIRE PROM INTO MIMAGE BITS 8-15
C
      IBPROM=0
      I=1
      CALL RDPROM(IERR,IWDCNT,IPRMWD,ISTBIT,IBPROM,ICRU,
     *MIMAGE(I))
C
C DUMP MIMAGE TO VERIFY THAT READ WORKED, USE HEX OUTPUT
C
      IF(IERR.NE.0)GO TO 30
        WRITE(6,20)MIMAGE
20      FORMAT(8(2X,Z4))
        STOP
30      WRITE(6,40)IERR,IWDCNT
40      FORMAT(2X,'READ PROM ERROR = ',I3,' WORD CNT = ',I4)
        STOP
      END
```

Figure 7-2. Example: Read a 2708 EPROM

```
C TEST PROGRAM PRGROM BY WRITING ALL 1'S OR 0'S
C
      DIMENSION MIMAGE(1024)
C
C SET MIMAGE TO DESIRED BIT STREAM DEPENDING ON TEST
C
      DO 5 I=1,1024
        MIMAGE(I)=255
5     CONTINUE
C
C DEFINE PROM CHARACTERISTICS
C
      IPBITS=8
      IPWD=8
      LOWHI=0
      IDUTY=100
      IPULSE=2
      IRETRY=0
C
C DEFINE ALL REMAINING PARAMETERS FOR CALL OF PRGROM
C
      IWDCNT=1024
      I=1
      ISTBIT=8
      ICRU=160
C
C PROGRAM PROM. DO LOOP USED TO ACCUMULATE 100 MILLASECONDS
C OF PULSE NEED TO MAKE EPROM WRITE WORK
C
      DO 10 K=1,100
        IBPROM=0
        CALL PRGROM(IERR,IWDCNT,IPBITS,IPWD,MIMAGE(I),
     *  ISTBIT,IBPROM,LOWHI,IDUTY,IRETRY,ICRU,IPULSE)
        IF(IERR.NE.0)GO TO 20
10    CONTINUE
      GO TO 30
20        WRITE(6,25)IERR,IWDCNT
25        FORMAT('  WRITE PROM ERROR = ',I3,' WORD CNT = ',
     *    I4)
      STOP
C
C READ EPROM TO VERIFY WRITE
C
30    IBPROM=0
      I=1
      CALL RDPROM(IERR,IWDCNT,IPWD,ISTBIT,IBPROM,ICRU,
     *MIMAGE(I))
      IF(IERR.NE.0)GO TO 50
        WRITE(6,40)MIMAGE
40      FORMAT(8(2X,Z4))
        STOP
50      WRITE(6,60)IERR,IWDCNT
60      FORMAT(2X,'READ PROM ERROR = ',I3,' WORD CNT = ',I4)
```

Figure 7-3. Example: Write Zeros to a 2708 EPROM

```
C TEST PROGRAM PRGROM BY WRITING ALL 1'S,OR 0'S
C
      DIMENSION MIMAGE(256)
C
C SET MIMAGE TO DESIRED BIT STREAM DEPENDING ON TEST
C
      DO 5 I=1,256
        MIMAGE(I)=15
5     CONTINUE
C
C DEFINE PROM CHARACTERISTICS
C
      IPBITS=1
      IPWD=4
      LOWHI=0
      IDUTY=25
      IPULSE=2
      IRETRY=0
C
C DEFINE ALL REMAINING PARAMETERS FOR CALL OF PRGROM
C
      IWDCNT=256
      I=1
      ISTBIT=12
      ICRU=160
      IBPROM=0
      CALL PRGROM(IERR,IWDCNT,IPBITS,IPWD,MIMAGE(I),
     *ISTBIT,IBPROM,LOWHI,IDUTY,IRETRY,ICRU,IPULSE)
      IF(IERR.EQ.0)GO TO 20
        WRITE(6,10)IERR,IWDCNT
10      FORMAT('  WRITE PROM ERROR = ',I3,' WORD CNT = ',I4)
      STOP
C
C READ PROM AS CHECK OF WRITE
C
20    IBPROM=0
      I=1
      CALL RDPROM(IERR,IWDCNT,IPWD,ISTBIT,IBPROM,ICRU,
     *MIMAGE(I))
      IF(IERR.NE.0)GO TO 40
        WRITE(6,30)MIMAGE
30      FORMAT(8(2X,Z4))
        STOP
40      WRITE(6,50)IERR,IWDCNT
50      FORMAT(2X,'READ PROM ERROR = ',I3,' WORD CNT = ',I4)
        STOP
      END
```

Figure 7-4. Example: Write a Bipolar S287 PROM to Ones

```
C USE IMGBLD TO CREATE ABSOLUTE OBJECT CODE OF RELOCATABLE
C OBJECT FILE AND THEN PROGRAM 4 S287 PROMS WITH THE
C ABSOLUTE OBJECT CREATED BY IMGBLD.
C
      DIMENSION MIMAGE(256), INRECD(40)
      DO 1 I=1,256
        MIMAGE(I)=15
1     CONTINUE
      IBIAS=251
      J=1
      JLAST=-1
      REWIND 5
5     READ(5,10) INRECD
10    FORMAT(40A2)
C
C CONVERT TO ABSOLUTE OBJECT THE RECORD JUST READ
C
      CALL IMGBLD(IBIAS,256,INRECD,MIMAGE(J),J,JLAST,IERR)
C
C USE VALUE OF JLAST TO DETECT END OF FILE
C
      IF(IERR.NE.0) GO TO 70
        IF(JLAST.EQ.-1) GO TO 5
          IPBITS=1
          IPWD=4
          LOWHI=0
          IDUTY=25
          IPULSE=2
          IRETRY=2
          IWDCNT=256
          ISTBIT=0
          ICRU=160
          IBPROM=0
          J=1
          DO 40 I=1,4
            DISPLAY(2,20,LINE=5,POSITION=1)
20          FORMAT('LOAD BLANK PROM, ENTER DIGIT WHEN DONE')
            ACCEPT(3,30,LINE=6,POSITION=10,ECHO)ISW
30          FORMAT(I1)
            CALL PRGROM(IERR,IWDCNT,IPBITS,IPWD,MIMAGE(J),
     *      ISTBIT,IBPROM,LOWHI,IDUTY,IRETRY,ICRU,IPULSE)
            IF(IERR.NE.0) GO TO 50
              ISTBIT=ISTBIT+4
              IBPROM=0
40        CONTINUE
          STOP
50            WRITE(6,60) IERR,IWDCNT
60            FORMAT(' WRITE ERROR = ',I2,' WORD CNT = ',I3)
              STOP
70      WRITE(6,80) IERR,IWDCNT
80      FORMAT(' IMGBLD ERROR = ',I2)
        STOP
      END
```

Figure 7-5. Example: Read and Relocate an Object Module

7.6 5MT/6MT SERIAL INTERFACE MODULE SUBROUTINES

CAUTION

The following subroutines can be used only if the Device Service Routine (DSR) for the 5MT/6MT module has been included in the system during system generation.

The 990 computer family provides interface capability with up to 256 input/output lines from the 5MT/6MT modules through the serial interface module. This interface module monitors and controls up to 256 input/output lines of digital information either one line at a time or up to 16 lines at a time. The interface module operates in two modes: sequential or random.

In the sequential mode the user transfers data in 16-bit blocks. As many as 16 of these 16-bit blocks may be transferred with a single instruction (up to 256 bits). In the random mode the user transfers data a single bit at a time. As many as 16 single bit transfers may be transferred with a single instruction (up to 16 bits).

The 5MT/6MT serial interface module can be controlled from FORTRAN programs by using calls to seven different subroutines. Table 7-4 lists the subroutines; the following paragraphs define the requirements for using these subroutines in a FORTRAN program.

NOTE

In all of these subroutines, the CRU input and output data is bit reversed due to the nature of the CRU data transfer operation. Refer to the Assembly Language Programmer's Guide for an explanation of this concept.

Table 7-4. 5MT/6MT ISA Subroutines

Subroutine	Purpose
OPENMT	Open the device
CLOSMT	Close the device
RDSTS	Read status
RDMTR	Read random
RDMTS	Read sequential
WRMTR	Write random
WRMTS	Write sequential

7.6.1 OPENMT. The OPENMT subroutine call opens the module for data transfer. This operation is required because the module is a file-oriented device. The subroutine is called with the following sequence:

CALL OPENMT (luno, flag, error)

The required parameters for this call are defined as follows:

- luno – The FORTRAN unit number assigned to the 5MT/6MT module
- flag – The Integer *2 program variable that keeps track of the open or closed status of the device. After an OPENMT call, the subroutine places a 1 in this variable to indicate that the device has been opened. After a CLOSMT call, the subroutine places a 0 in this variable to indicate that the device has been closed.
- error – The Integer *2 program variable that contains a status code upon completion of any operation. The subroutine places the status code in this location. If no error occurs, a zero is returned.

Error Status Codes:

00 – Normal completion
02 – Illegal operation
04 – Record loss due to power failure
06 – Terminated abnormally due to some external event.

7.6.2 CLOSMT. The CLOSMT subroutine call closes the module for data transfer. This operation is required because the module is a file-oriented device. The subroutine is called with the following sequence:

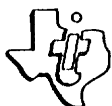
CALL CLOSMT (luno, flag, error)

The required parameters for this call are defined as follows:

- luno – The FORTRAN unit number assigned to the 5MT/6MT module
- flag – The Integer *2 program variable that keeps track of the open or closed status of the device. After an OPENMT call the subroutine places a 1 in the variable to indicate that the device has been opened. After a CLOSMT call, the subroutine places a 0 in this variable to indicate that the device has been closed.
- error – The Integer *2 program variable that contains a status code upon completion of any operation. The subroutine places the status code in this location.

Error Status Codes:

00 – Normal completion
02 – Illegal operation
04 – Record loss due to power failure
06 – Terminated abnormally due to some external event.



7.6.3 RDSTS. The RDSTS subroutine reads the present state of all 256 input lines and stores them in a 17 element program buffer. The last element in that buffer provides module status information such as output module power on and interrupt mask status. In operation, the information from the 5MT/6MT input modules, is read into an input RAM on the interface module. The subroutine reads the information from this RAM into the program buffer. The calling sequence for this subroutine is:

CALL RDSTS (luno, flag, buffer address, error)

The required parameters for this call are defined as follows:

- luno – The FORTRAN unit number assigned to the 5MT/6MT module
- flag – The Integer *2 program variable that keeps track of the open or closed status of the device. After an OPENMT call, the subroutine places a 1 in this variable to indicate that the device has been opened. After a CLOSMT call, the subroutine places a 0 in this variable to indicate that the device has been closed.
- buffer address – The location of the 17 element program buffer for storage of status information from the modules. This buffer must be declared Integer *2 in the FORTRAN program.
- error – The Integer *2 program variable that contains a status code upon completion of any operation. The subroutine places the status code in this location.

Error Status Codes

00 – Normal completion
02 – Illegal operation
04 – Record loss due to power failure
06 – Terminated abnormally due to some external event.

7.6.4 RDMTR. The RDMTR subroutine reads data from the input RAM, one bit at a time, and stores the information, right justified, in a one-element calling program buffer. Although the data is input one bit at a time, the user may specify that as many as 16 bits be input with one subroutine call. The user may also specify that a transmit/receive cycle will be initiated by setting the t/r flag. The t/r flag variable is set or reset by placing a 1 or a 0 in it. When a transmit/receive cycle is initiated, the contents of the serial interface module's output RAM is written to all of the 5MT/6MT output modules, and the current data from all the input modules is read into the RAM. If the t/r flag is reset, a transmit/receive cycle is initiated by the interface module to refresh the data in the input RAM with the present status of all 256 input lines. If the t/r flag is set, the data in the input RAM is not refreshed. The calling sequence for the subroutine is:

CALL RDMTR (luno, flag, buffer address, count, starting address, t/r flag, error)

The required parameters for this call are:

- luno – The FORTRAN unit number assigned to the 5MT/6MT module
- flag – The Integer *2 program variable that keeps track of the open or closed status of the device. After an OPENMT call, the subroutine places a 1 in the variable to indicate that the device has been opened. After a CLOSMT call, the subroutine places a 0 in this variable to indicate that the device has been closed.

- buffer address – The Integer *2 address of the program buffer. This buffer holds the incoming bit information in a right justified format.
- count – An Integer *2 value within the range of 1 through 16 that indicates the number of bits of information to retrieve from the input RAM.
- starting address – An Integer *2 value within the range of 1 through 256 that indicates the starting point in the input RAM for the input operation.
- t/r flag – When this flag is set (equal to a 1), it inhibits the start of a transmit/receive cycle on the interface module. When this flag is equal to a 0, it enables the start of a transmit/receive cycle on the interface module.
- error – The Integer *2 program variable that contains a status code upon completion of any operation. The subroutine places the status code in this location. If no error occurs, a zero is returned.

Error Status Codes:

00	–	Normal completion
02	–	Illegal operation
04	–	Record loss due to power failure
06	–	Terminated abnormally due to some external event.

7.6.5 RDMTS. The RDMTS subroutine reads data from the input RAM in bytes and stores the information in a calling program buffer. The user may specify up to 32 bytes for input with one call. The user may also specify the starting address in the input RAM for the start of the input operation. The input information is placed in locations in the calling program buffer corresponding to those in the input RAM. The subroutine is called with the following sequence:

CALL RDMTS (luno, flag, buffer address, count, starting address. t/r flag, error)

The required parameters for this call are as follows:

- luno – The FORTRAN unit number assigned to the 5MT/6MT module.
- flag – The Integer *2 program variable that keeps track of the open or closed status of the device. After an OPENMT call, the subroutine places a 1 in the variable to indicate that the device has been opened. After a CLOSMT call, the subroutine places a 0 in this variable to indicate that the device has been closed.
- buffer address – The address of the program buffer where input data is stored in locations corresponding to those of input RAM.
- count – An Integer *2 within the range of 1 through 32 that indicates the number of bytes to be retrieved from the input RAM.
- starting address – An Integer *2 ranging from 1 through 16 that indicates a word boundary address within both the input RAM and the calling program buffer. This address indicates the starting point in both areas for the transfer operation.

- t/r flag – When this Integer *2 flag is set (equal to a 1), it inhibits the start of a transmit/receive cycle on the interface module. When this flag is equal to a 0, it enables the start of a transmit/receive cycle on the interface module.

- error – The Integer *2 program variable that contains a status code upon completion of any operation. The subroutine places the status code in this location. If no error occurs, a zero is returned.

Error Status Codes:

00	–	Normal completion
02	–	Illegal operation
04	–	Record loss due to power failure
06	–	Terminated abnormally due to some external event.

7.6.6 WRMTR. The WRMTR subroutine writes data, one bit at a time, from a calling program buffer to the output RAM. The subroutine always starts the output operation with the least significant bit of the program buffer, regardless of the number of bits to be transferred. Therefore, the information in the program buffer must be previously right justified. The t/r flag for this subroutine is similar to the t/r flag for the RDMTR subroutine, except that the transmit/receive cycle begins after the information is written to the output RAM (if the flag is equal to a 0). When the t/r cycle is enabled, the data is first written to the output RAM and then the entire output RAM is written to the output 5MT/6MT modules. The subroutine is called with the following sequence:

CALL WRMTR (luno, flag, buffer address, count, starting address, t/r flag error)

The required parameters for this call are as follows:

- luno – The FORTRAN unit number assigned to the 5MT/6MT module.

- flag – The Integer *2 program variable that keeps track of the open or closed status of the device. After an OPENMT call, the subroutine places a 1 in the variable to indicate that the device has been opened. After a CLOSMT call, the subroutine places a 0 in this variable to indicate that the device has been closed.

- buffer address – The Integer *2 program buffer in which the information to be output to the output RAM is located.

- count – A value within the range of 1 through 16 that indicates the number of bits of information that are to be output from the program buffer.

- starting address – A value within the range of 1 through 256 that indicates the starting point in the output RAM for the write operation.

- t/r flag – When this Integer *2 flag is set (equal to a 1), it inhibits the start of a transmit/receive cycle on the interface module. When this flag is equal to a 0, it enables the start of a transmit/receive cycle on the interface module.

- error – The Integer *2 program variable that contains a status code upon completion of any operation. The subroutine places the status code in this location. If no error occurs, a zero is returned.

Error Status Codes:

00 – Normal completion
02 – Illegal operation
04 – Record loss due to power failure
06 – Terminated abnormally due to some external event.

7.6.7 WRMTS. The WRMTS subroutine writes bytes of data from a calling program buffer to the output RAM bytes. With one call sequence the user can output up to 32 bytes of data. The use of the t/r flag is the same for this call as it is for the WRMTR subroutine call. The subroutine is called with the following sequence:

CALL WRMTS (luno, flag, buffer address, count, starting address, t/r flag, error)

The required parameters for this call are as follows:

- luno – The FORTRAN unit number assigned to the 5MT/6MT module.
- flag – The Integer *2 program variable that keeps track of the open or closed status of the device. After an OPENMT call, the subroutine places a 1 in the variable to indicate that the device has been opened. After a CLOSMT call, the subroutine places a 0 in this variable to indicate that the device has been closed.
- buffer address – The program buffer where the output information is stored. Each element in the buffer is Integer *2.
- count – A value within the range of 1 through 32 that indicates the number of bytes to be transferred to the output RAM.
- starting address – An Integer *2 value within the range of 1 through 16 that indicates the word boundary for the start of the output transfer. The address applies to both the output RAM and the program buffer.
- t/r flag – When this Integer *2 flag is set (equal to a 1), it inhibits the start of a transmit/receives cycle on the interface module. When this flag is equal to a 0, it enables the start of a transmit/receive cycle on the interface module.
- error – The Integer *2 program variable that contains a status code upon completion of any operation. The subroutine places the status code in this location. If no error occurs, a zero is returned.

Error Status Codes:

00 – Normal completion
02 – Illegal operation
04 – Record loss due to power failure
06 – Terminated abnormally due to some external event.

7.7 32 INPUT/TRANSITION DETECTION MODULE SUBROUTINES

CAUTION

> The following subroutines can be used only if the Device Service Routine (DSR) for the 32 Input/Transition Detection module has been included in the system during system generation.

The 32 Input/Transition Detection module monitors 32 input lines for digital information. The module detects data in one of two ways: individual line monitoring or 32-bit block storage.

When operating in the individual monitoring mode, the module checks one of the 32 input lines for any transitions that may occur on that line (transition from high to low or from low to high). The line that is checked is determined by a mask that blocks inputs from other lines on the module. The user then enters a command to wait for a transition. When a transition occurs, on an enabled line, an interrupt is generated and the current value of the transition data, along with its corresponding address, is placed in a program buffer.

When operating in the block storage mode, the module reads the current value of all 32 input lines and places them in a two word program buffer.

NOTE

> In this mode the CRU input data is bit reversed in the calling program buffer due to the nature of the CRU data transfer operation. Refer to the Assembly Language Programmer's Guide for the explanation of the CRU data transfer.

The 32 IT module can be controlled from FORTRAN programs through five callable subroutines. Table 7-5 lists these subroutines; the following paragraphs define the requirements for using these subroutines in a FORTRAN program.

Table 7-5. 32 IT Module ISA Subroutines

Subroutine	Purpose
OPN32I	Open the device
CLS32I	Close the device
WRTMSK	Write mask
READ 32	Read 32 data lines
RDIBIT	Read interrupting bit

7.7.1 OPN32I. The OPN32I subroutine opens the module for data reception. The module must be opened since it is a file oriented device. The subroutine is called with the following sequence:

CALL OPN32I (luno, flag, error)

This required parameters for this call are as follows:

- luno – The FORTRAN unit number assigned to the 32 IT module
- flag – The Integer *2 program variable that keeps track of the open or closed status of the device. After an OPN32I call, the subroutine places a 1 in this variable to indicate that the device has been opened. After a CLS32I call, the subroutine places a 0 in this variable to indicate that the device has been closed.
- error – The Integer *2 program variable that contains a status code upon completion of any operation. The subroutine places the status code in this location.

Error Status Codes:

00 – Normal completion
02 – Illegal operation
04 – Record loss due to power failure
06 – Terminated abnormally due to some external event.

7.7.2 CLS32I. The CLS32I subroutine closes the module following data reception. The module must be closed since it is a file oriented device. The subroutine is called with the following sequence:

CALL CLS32I (<luno, flag, error>)

- luno – The FORTRAN unit number assigned to the 32 IT module
- flag – The Integer *2 program variable that keeps track of the open or closed status of the device. After an OPN32I call, the subroutine places a 1 in this variable to indicate that the device has been opened. After a CLS32I call, the subroutine places a 0 in this variable to indicate that the device has been closed.
- error – The Integer *2 program variable that contains a status code upon completion of any operation. The subroutine places the status code in this location.

Error Status Codes:

00 – Normal completion
02 – Illegal operation
04 – Record loss due to power failure
06 – Terminated abnormally due to some external event

7.7.3 WRTMSK. The WRTMSK subroutine accepts a hexadecimal mask from a two-element program buffer and writes it to the 32 IT module. Any line set to a 1 by the mask can then recognize a transition and produce an interrupt; any line set to a 0 by the mask is prevented from recognizing a transition and producing an interrupt. The subroutine is called with the following sequence:

CALL WRTMSK (luno, flag, data address, error)

The required parameters for this call are as follows:

- luno – The FORTRAN unit number assigned to the 32 IT module
- flag – The Integer *2 program variable that keeps track of the open or closed status of the device. After an OPN32I call, the subroutine places a 1 in this variable to indicate that the device has been opened. After a CLS32I call, the subroutine places a 0 in this variable to indicate that the device has been closed.
- data address – The address of a two-word calling program buffer that contains the interrupt mask. It should be typed Integer *4 in the FORTRAN program.
- error – The Integer *2 program variable that contains a status code upon completion of any operation. The subroutine places the status code in this location.

Error Status Codes:

00 – Normal completion
02 – Illegal operation
04 – Record loss due to power failure
06 – Terminated abnormally due to some external event

7.7.4 READ32. The READ32 subroutine call reads the current value of all 32 input lines and stores the data in a two-word calling program buffer. The subroutine is called with the following sequence:

CALL READ32 (luno, flag, data address, error)

The required parameters for this call are as follows:

- luno - The FORTRAN unit number assigned to the 32 IT module
- flag – The Integer *2 program variable that keeps track of the open or closed status of the device. After an OPN32I call, the subroutine places a 1 in this variable to indicate that the device has been opened. After a CLS32I call, the subroutine places a 0 in this variable to indicate that the device has been closed.
- data address – The address of a two-word calling program buffer for storing the input data from all 32 module input lines. It should be typed Integer *4 in the FORTRAN program.
- error – The address of a program variable that contains a status code upon completion of any operation. The subroutine places the status code inthis location. If no error occurs, a zero is returned.

Error Status Codes:

00 – Normal completion
02 – Illegal operation
04 – Record loss due to power failure
06 – Terminated abnormally due to some external event

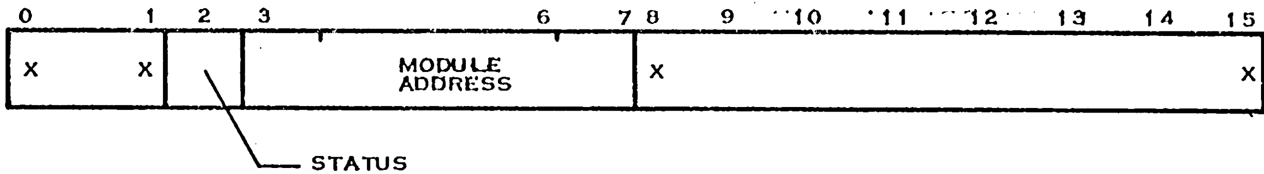


Figure 7-6. RDIBIT Calling Program Buffer

7.7.5 RDIBIT. The RDIBIT subroutine monitors all 32 of the module's input lines until an interrupt (generated by a transition on one of the lines) is detected. It then places the current value of the interrupting input line in a one-word calling program buffer along with the address of the interrupting input line. The data is placed in the left byte of the calling program buffer as illustrated in figure 7-6.

Before the module can generate an interrupt, it must have been previously enabled by a WRTMSK subroutine call. The module can generate an interrupt only if the input line corresponding to the detected transition has been enabled by a 1 value in the WRTMSK call. If the write mask contained a value of 0 for that input line, or if the WRTMSK call has not been executed, the module cannot generate an interrupt for the detected transition.

The module accepts up to 32 simultaneous transitions, one at each of the inputs to the module. However, it generates only one interrupt at a time. After the first interrupt is serviced, it generates an interrupt for another of the detected transitions until all of the detected transitions have been serviced. The RDIBIT subroutine services only one interrupt at a time. Therefore, a new call is required for each interrupt generated by the module. Each new call over-writes the data in the calling program buffer unless a new buffer address is generated for each new call.

The subroutine is called with the following sequence:

CALL RDIBIT (luno, flag, data address, error)

The required parameters for this call are as follows:

- luno – The FORTRAN unit number assigned to the module

- flag – The Integer *2 program variable that keeps track of the open or closed status of the device. After an OPN32I call, the subroutine places a 1 in this variable to indicate that the device has been opened. After a CLS32I call, the subroutine places a 0 in this variable to indicate that the device has been closed.

- data address – The address of a one-word calling program buffer that contains the current value of the interrupting input line, along with its address.

- error – The address of a program variable that contains a status code upon completion of any operation. The subroutine places the status code in this location. If no error occurs, a zero is returned.

Error Status Codes:

00 – Normal completion
02 – Illegal operation
04 – Record loss due to power failure
06 – Terminated abnormally due to some external event

7.8 MULTI-KEY INDEX FILE HANDLER

The Multi-Key Index File Handler is an assembly language routine which provides access to DX10 key index files from a FORTRAN program. There are 21 entry points to this routine, each of which performs a different function on key index files. Depending on the entry point called, different argument lists are required to be passed by the user. The following list describes the different types of arguments required by the key index file handler.

- BLOCK — This argument corresponds to an integer array dimensioned to have nine elements. The user must initialize the fifth element of this array with the value of the record length of the key index file. Also, note that a separate array is required for each key index file accessed in a program. This array is used to hold the key index file SVC call block.

  Example:
  ```
  INTEGER BLOCKA(9), BLOCKB(9)
  DATA BLOCKA(5)/80/, BLOCKB(5)/132/
  ```

- CURR — This argument corresponds to an integer array dimensioned to have ten elements. A separate array is required for each key in each key index file accessed in a program. The first element of this array must be initialized with the number of the key to which it corresponds. This array is used to hold the currency block for each key.

  Example:
  ```
  INTEGER CURRA1(10),CURRA2(10),CURRA3(10),
  CURRA4(10)
  INTEGER CURRB1(10),CURRB2(10)
  DATA   CURRA1(1)/1/,CURRA2(1)/2/,CURRA3(1)/3/)
  CURRA4(1)/4/
  DATA CURRB1(1)/1/,CURRB2(1)/2/
  ```

- PATH — This argument corresponds to an integer array large enough to hold the ASCII representation of the key index file pathname plus one word to hold the pathname length.

  Example:
  ```
  INTEGER PATHA(9),PATHB(12)
  DATA PATHA / 16, 'VOL.DIR.KEYFILE1' /
  DATA PATHB / 21, 'TRID32.DIREC.DATABASE'
  ```

- BUFFER — This argument corresponds to an array large enough to hold a logical record from the key index file.

  Example:
  ```
  INTEGER BUFFA(40), BUFFB(64)
  ```

- KEYBUF — This argument corresponds to an array large enough to hold the ASCII representation of the key value being used.

  Example:
  ```
  INTEGER KEYBUF(64)
  ```

- KEYSIZ — This argument corresponds to an integer value that represents the starting position of a truncated key value.

- ACCESS — This argument corresponds to an integer value that determines the access method to be used in communicating with a key index file. The set of possible values is:

  Exclusive Write = 0
  Exclusive All = 8
  Shared = 16
  Read-Only = 24

After each key index file operation, the operating system returns information that indicates the results of the operation. The status code is an 8-bit value that indicates a serious error in the key file operation. This value is found in the lower half of the first element in array BLOCK of the file being used. This value can be obtained on return from any key index file call with the following statements:

```
ISTAT = LAND(BLOCK(1),>FF)
BLOCK(1) = LAND(BLOCK(1),>FF00)
```

The first statement moves the status code into variable ISTAT, and the second statement clears the status code for the next operation. Refer to volume three of the *DX10 Operating System Release 3 Reference Manual* for a list of the possible status code values. Another code returned after an operation is the informative code. This code is an 8-bit value which describes conditions which can arise in normal key index file operations. This value is found in the upper half of the first element in array CURR of the key being used. This value can be obtained after a key index file call with the following statements:

```
INFORM = ISHFT(LAND(CURR(1),>FF00),-8)
CURR(1) = LAND(CURR(1),>FF)
```

The first statement moves the informative code into the lower half of variable INFORM, and the second statement clears the informative code for the next operation. The possible values returned in the informative code may be found in volume three of the *DX10 Operating System Release 3 Reference Manual.*

The following list of key index file operations corresponds exactly with the list of operations given in volume three of the *DX10 Operating System Release 3 Reference Manual.* The user should read carefully through this documentation before attempting to use key index files. FORTRAN checks subroutine calls for arguments that may cause problems if the user switches the type of a parameter in a call to the same key index file handler routine. For this reason it is recommended that the user work only with arrays of the type INTEGER.

1. Close file:

   ```
   CALL XCLOSE(BLOCK)
   ```

2. Read file characteristics:

   ```
   CALL XRDCHR(BLOCK,BUFFER)
   ```

3. Change access privileges:

   CALL XCNGAC(BLOCK,ACCESS)

4. Open random:

   CALL XOPNRN(BLOCK,ACCESS,PATH)

5. Read by key, unlocked:

   CALL URDKEY(BLOCK,BUFFER,CURR,KEYBUF)

6. Read by key, with lock:

   CALL LRDKEY(BLOCK,BUFFER,CURR,KEYBUF)

7. Read current, unlocked:

   CALL URDCUR(BLOCK,BUFFER,CURR)

8. Read current, with lock:

   CALL LRDCUR(BLOCK,BUFFER,CURR)

9. Read by primary key, unlocked:

   CALL URDPRM(BLOCK,BUFFER,CURR,KEYBUF)

10. Read by primary key, with lock:

    CALL LRDPRM(BLOCK,BUFFER,CURR,KEYBUF)

11. Read next, unlocked:

    CALL URDNXT(BLOCK,BUFFER,CURR)

12. Read next, with lock:

    CALL LRDNXT(BLOCK,BUFFER,CURR)

13. Insert record into file:

    CALL XINSRT(BLOCK,BUFFER,CURR)

14. Rewrite and leave locked:

    CALL LREWRT(BLOCK,BUFFER,CURR)

15. Rewrite with unlock:

    CALL UREWRT(BLOCK,BUFFER,CURR)

16. Delete by key:

    CALL KDLETE(BLOCK,CURR,KEYBUF)

17. Delete by currency:

    CALL CDLETE(BLOCK,CURR)

18. Unlock by currency:

    CALL CUNLOK(BLOCK,CURR)

19. Set currency equal:

    CALL SCUREQ(BLOCK,CURR,KEYBUF,KEYSIZ)

20. Set currency equal or greater:

    CALL SCUREG(BLOCK,CURR,KEYBUF,KEYSIZ)

21. Set currency greater:

    CALL SCURGT(BLOCK,CURR,KEYBUF,KEYSIZ)

SECTION VIII

PROGRAMMING TECHNIQUE

The FORTRAN language was the first of the higher-level languages still being used. No formalization of computer languages existed when the first FORTRAN compiler was implemented in 1954; no concept of "structured" programming had been evolved; and the FORTRAN syntax reflected (and still reflects) the architecture of the original target computer.

Presently the major version of the language, as measured by the number of available implementations, is FORTRAN IV. Although it is an extension of the original language, the structure and nucleus facilities are essentially as they were in the beginning.

Some of the better means of structuring programs can be achieved by imposing standards of coding stricter than those of the compiler. A well-constructed program debugs faster than one that is not, an aspect of programming that is rapidly becoming more important with rising software costs.

This section includes some recommended programming techniques.

1. *Make liberal use of comments.* It is important for the programmer to document his program thoroughly so that a user less familiar with it can read through and follow it easily. Even an experienced programmer will be able to read through a program more easily if it includes comments. One programmer's coding steps may not be clear to other programmers.

   FORTRAN is not one of the more readable languages, and is frequently used in highly technical, difficult program modules. Comments help the writer of the program think through his problem more clearly and speed up the debugging process during initial development.

   Comments can be read more easily if they are clearly delineated from program code, for example by a standard indentation or a border of asterisks.

2. *Plan statement number assignments carefully.* A coherent scheme of statement number assignments makes the program steps easier to follow. Do not number successive branch targets 1, 2, 3, ... Generally, during the normal coding and debugging process, a need for more branch points will arise after the first draft of the module.

   Allow increments of at least 10 in number assignments. Statements may then be inserted later and be assigned numbers that fit into the existing sequence. If statement numbers are related to their location in the listing, it is much easier for other programmers to locate the three branch points of an IF statement or locate any other statement that is not executed in program sequence.

Another possible convention is to use different digit positions in the statement number to indicate the function of the statement, as in the following scheme:

Statement Number	Meaning
000n0	Branch target
00n00	DO loop end
0n000	FORMAT statement
n0000	READ/WRITE statement

An intelligently planned scheme of statement numbering can be valuable as a documentation aid.

3. *Avoid branching into and out of DO loops.* Branches of this type are rarely necessary; a program with many of them can be difficult to read and modify. If the programmer avoids them, his programs will be clear and much more easily understood.

4. *Whenever feasible, use subroutine CALLs and function references in place of GO TO statements.* These substitutions should be made for the following reasons:

   a. The program will be more readable.

   b. The chances of endless loops in the program are reduced substantially.

   c. This practice encourages the programmer to approach the problem as a set of smaller tasks instead of one large program module.

   d. Debugging time will be reduced.

5. *The programmer may use features of 990 FORTRAN as documentation aids*, if he is willing to sacrifice compatibility with some other compilers. The two features are as follows:

   a. The program code need not begin in column 7; therefore, DO loops, DO loop nests, and other logical groupings may be indented to separate them visually and improve readability.

   b. Identifiers are not limited to six characters in length. This allows variable names to be meaningful and readable when a longer word may be spelled out. Because the compiler only looks at the first six characters of the label, however, duplications in these characters must be avoided.

6. *Leave as much set-up program code as possible outside the range of a DO.* By not including initialization or other set-up code in a DO, execution time may be reduced substantially.

7. Use the conditional compile option and D comment records to print program status information as a debug aid.

SECTION IX

FORTRAN COMPILER OUTPUT

9.1 GENERAL

The compiler program, used to translate programs from FORTRAN statements into executable machine language, produces two outputs. One is a listing of the FORTRAN program statements; the other is the linkable object code.

9.2 OUTPUT LISTING ELEMENTS

The output listing of the FORTRAN compiler consists of the following elements (if required).

- Source program, error messages, and the total number of errors.
- Common allocation – blank and/or labeled
- Required subroutines
- Equivalence allocation
- Program allocation

Program allocation lists all variables not appearing in COMMON or EQUIVALENCE statements.

9.3 ERROR DIAGNOSTICS AND MESSAGES

Error diagnostics and messages are listed and explained in the paragraphs that follow. The topics covered are:

- Statement error diagnostics
- Program error diagnostics
- Runtime error diagnostics
- Runtime error messages

9.3.1 STATEMENT ERROR DIAGNOSTICS. During compilation, statements which violate the syntactic or semantic rules of the language are recognized and error indications are printed. There are two levels of statement diagnostics: warnings and errors. Warnings are issued for minor infractions where the compiler can still determine what is to be done and compile the statement. Errors are severe violations of the language. In the case of errors, compilation proceeds as if the statement was never encountered. The statement label, if any, remains defined. If the erroneous statement is ever executed, it causes a link to a system routine which will halt execution of the program and notify the user that an attempt has been made to execute an erroneous statement. The name of the program and the line number of the statement are displayed.

One character of the statement is marked with a currency symbol "$" directly beneath the erroneous character, for example:

```
ZATA = X + Y * - A
               $
```

indicates that the character "-" is an error.

In the case of a syntax error, the marked character itself was unacceptable, as in the previous example. In the case of a semantic error, an identifier error or other construct error, the mark indicates the last character of the construct. For example, in the line:

```
COMMON ALPHA,BETA,ALPHA,GAMMA
                        $
```

the mark indicates that the identifier ALPHA is misused.

The compiler attempts all interpretations of statement type before discarding a statement. The marked position indicates the greatest amount of correct information found under the *most* logical assumption of statement type.

A comment specifying the reason for the failure is output directly after the marked line. There may be more then one diagnostic message per line. The diagnostic messages are listed left-to-right. Each message is followed by a sequence of characters and E*E*... or W*W*... indicating error or warning, respectively. An alphabetic list of possible statement diagnostic messages appears in table 9-1.

9.3.2 PROGRAM ERROR DIAGNOSTICS. After the source program has been listed, the compiler lists summary error messages pertaining to the entire program. Table 9-2 lists and defines these error messages.

9.3.3 RUNTIME ERROR DIAGNOSTICS. When the program is executed the runtime package produces a listing pertaining to any errors that may have occurred during execution. Tables 9-3 and 9-4 define these error messages.

9.4 OBJECT CODE

The object code produced by the FORTRAN compiler is in a relocatable format suitable for input to the link editor program.

The PROGRAM ID is taken from the name of the function or subroutine subprogram. The compiler allows six-character names for these subprograms. When the name is less than six characters long, the rightmost vacant characters are replaced with spaces. Thus, a FORTRAN program heading SUBROUTINE SORT (A,N) will yield a subprogram named SORTƀƀ where ƀ signifies space or blank. When the program name exceeds six characters, only the first six are used to identify the program; those programs which are not identified as functions, block data or subroutines are automatically named $MAIN. Since the programmer cannot generate a name with a leading $, this name is unique. The PROGRAM ID for a specification subprogram is always $BLOCK.

Labeled common names are derived in the same way. The common area which the programmer does not name is named $BLANK by the compiler.

For each program module or subprogram, the compiler also reserves a block of storage for storing workspace pointer, local variables, temporary variables and dummy parameters. The compiler labels each block, $DATA, and assigns the storage area directly following the area containing the program for which it was created. The $DATA segment is created for use by the link editor to provide relocatable storage for the program parameters.

Table 9-1. FORTRAN Compiler Statement Error Diagnostic Messages

Message	Meaning	Corrective Action
ARGUMENT CONVERTED W*W*W*	The type of the indicated parameter for an intrinsic function was converted to agree with the type required by the function.	Accept the conversion, select another intrinsic function or convert the parameter.
ARGUMENT COUNT E*E*E*	The number of parameters to a subprogram is wrong. The compiler detected the condition either because the function is an intrinsic function or because the same subprogram was called previously with a different number of parameters.	Supply correct parameters.
ARRAY SIZE E*E*E*	The program attempted to declare an array that has more elements than the maximum allowed.	Correct array declaration to 3 dimensions of less than available memory size in total elements.
BLOCK DATA ONLY E*E*E*	A DATA statement not in a BLOCK DATA subprogram attempted to initialize a variable in the COMMON area.	Move the offending DATA statement to a BLOCK DATA subprogram, or remove the variable from COMMON.
	–or–	
	An executable statement has been included in a BLOCK DATA subprogram.	Remove the indicated statement from the BLOCK DATA subprogram.
CONSTANT SIZE E*E*E*	The indicated constant falls outside the allowable range for constants (see paragraph 2.4).	Correct data to stay within allowable limits.
DATA TYPE E*E*E*	The type of a constant in a DATA statement does not agree with the type of the variables that it initializes.	Change type of indicated constant or change type of its corresponding variables.
DATA COUNT E*E*E*	The number of variables in a DATA statement does not agree with the number of constants.	Provide correct number of constants or variables.
DECLARATION CONFLICT E*E*E*	The program attempted to declare an identifier as a FORTRAN entity (simple variable, array, subprogram, or statement function name) and that identifier has been previously used for a different purpose.	Select a new identifier.
DIMENSION RANGE E*E*E*	An array dimension has been declared by the program that is outside the allowable range.	Correct the array dimension to prevent memory overflow.
DUPLICATE DUMMY E*E*E*	A dummy variable has been declared more than once in a statement function definition, FUNCTION, or SUBROUTINE statement.	Remove redundant declarations.
DUPLICATE OPTION E*E*E*	The program has stated one of the keyword options in an ACCEPT or DISPLAY statement more than once.	Remove redundant statements.
EXTRA COMMA W*W*W*	The compiler encountered more than one comma at a point where it expected only a single comma.	Remove extra commas.
FORMAT LABEL E*E*E*	The indicated statement number was declared in the label field of a FORMAT statement and was used in the program for other than a format reference.	Correct statement number reference or assign statement number to correct item.
ILLEGAL DO CLOSE W*W*W*	A DO loop was closed with an illegal statement.	Modify DO loop to include an executable statement in the last statement.

Table 9-1. FORTRAN Compiler Statement Error Diagnostic Messages (Continued)

Message	Meaning	Corrective Action
ILLEGAL LABEL E*E*E*	A statement number in the program is greater than the five digit maximum.	Reduce statement number to five digits.
ILLEGAL NUMBER E*E*E*	The indicated constant either overflowed or underflowed.	Correct the indicated constant to remain within allowable limits.
JUMP LABEL E*E*E*	The program used a statement number that is not a FORMAT statement to define a format.	Correct statement number reference to indicate the proper FORMAT statement.
LABEL MISSING W*W*W*	The indicated statement cannot be executed because it has no statement number.	Assign a valid, unique statement number to the indicated statement.
MISSING COMMA W*W*W*	The program is missing a comma at a point where the compiler expected to find one; however, the program compilation could continue.	Add a comma in the proper point.
MISUSED NAME E*E*E*	The program uses an identifier in the wrong context, such as: • A dummy variable within a DATA or EQUIVALENCE statement. • A variable dimension that is not a simple dummy variable • A subprogram name used without parameters in an expression.	Correct name.
MULTI DEFINED E*E*E*	A statement number has more than one statement assigned to it.	Ensure that all statements have unique statement numbers.
NOT ARRAY E*E*E*	The program used an identifier that is not an array name in an operation requiring an array.	Check identifier and insert the proper name, or correct operation so that an array is not required.
NOT INTEGER E*E*E*	A variable or expression that is not an integer type has been used where only an integer type is allowed.	Correct variable or expression to conform to the type required.
NUMBER OF SUBSCRIPTS E*E*E*	The number of subscripts in an array reference is incorrect.	Compare reference with array dimensions and resolve conflict.
RANGE E*E*E*	The first character in the declaration of an IMPLICIT range does not alphabetically precede the second character.	Reorder the range declaration in alphabetical order.
	–or–	
	The (scale) of a FIXED declaration is outside of the allowable range.	Adjust the (scale) specification to fall within -31 to +31.
	–or–	
	A constant subscript within an array reference exceeds the range of subscript values defined for that array.	Change the constant subscript to agree with the defined range.

Table 9-1. FORTRAN Compiler Statement Error Diagnostic Messages (Continued)

Message	Meaning	Corrective Action
STATEMENT NOT ALLOWED E*E*E*	A statement has been used in an illegal context.	Correct statement usage.
	–or–	
	The program uses an illegal secondary statement in a logical IF.	
	–or–	
	A statement appears in the wrong order in the program (i.e., a statement function definition occurring after executable statements).	
SYNTAX E*E*E*	Erroneous punctuation or illegally constructed arithmetic expression. The character marked in the output indicates how much of the statement that the compiler read before the statement could not be interpreted.	Correct punctuation or arithmetic expression.
TYPE CONFLICT E*E*E*	An IMPLICIT statement has declared a starting letter of an identifier to be of two different types.	Correct the IMPLICIT statement to define only one type for each starting character.
	–or–	
	The types of the operands of an arithmetic or logical operator are illegal.	Correct operands to agree with the requirements of the expression.
	–or–	
	The types of the right and left sides of an assignment are not compatible.	Correct the assignment so that both sides are of the same type.
UNDIMENSIONED E*E*E*	A simple variable in the program is followed by an open-paren, (, that the compiler interpreted to be an attempt to use the variable as an array reference.	Remove the open-paren from the statement, or dimension the simple variable as an array.
UNRECOGNIZABLE E*E*E*	The compiler could not recognize the entire statement.	Correct the statement.
UNSUCCESSFUL COPY E*E*E*	The compiler could not perform the requested COPY statement.	Ensure that the file name is correct; ensure that the file exists in the system library; ensure that an updated compiler is being used.

Table 9-2. FORTRAN Compiler Program Error Diagnostic Messages

Message	Meaning	Corrective Action
ALLOCATION ERRORS	Following this heading, the compiler lists the identifiers that the program incorrectly assigned memory locations. These errors occur in either COMMON or EQUIVALENCE statements in the program and are the result of errors such as: • Using an EQUIVALENCE statement to equate variables from different blocks of COMMON. • An attempt to extend a COMMON block backwards. • Attempting to perform an impossible EQUIVALENCE.	Examine identifiers in COMMON and EQUIVALENCE statements and correct erroneous condition.
CANNOT ALLOCATE TEMPORARY FILE	The disc space is not available to create one or both of the temporary files required during compilation.	Delete unnecessary files from disc, or otherwise create additional disc space.
COMPILER BUG COLLAPSE PHASE	The compiler was unable to complete the compilation due to a problem internal to the compiler.	System compiler errors require the attention of the system programmer.
COMPILER BUG GEN PHASE	The compiler was unable to begin the compilation due to a problem internal to the compiler.	System compiler errors require the attention of the system programmer.
COMPILER POINTER OVERFLOW	Insufficient memory space is available for the pointer. More than !023 of one type of item, i.e., labels, arrays, scalars, etc.	Decrease number of items of that type, which can only be determined by careful scrutiny of the source program. The best action may be to divide the program into smaller segments.
COMPILER PROGRAM OVERFLOW	Insufficient memory space is available for storing the program.	Increase requested memory size for the compilation, –or– Delete the X (variable cross-reference list) option.
COMPILER ROLL MEMORY OVERFLOW	Insufficient memory space is available for the roll memory used during compilation.	Increase requested memory size for the compilation, –or– Delete the X (variable cross-reference list) option.
FATAL ERROR, END VECTOR TAKEN CODE = X	The operating system has detected a fatal error and has terminated the compiler task.	Refer to the DX10 operating system documentation for the corresponding reason for error code X (task error code).

Table 9-2. FORTRAN Compiler Program Error Diagnostic Messages (Continued)

Message	Meaning	Corrective Action
FORTRAN COMPILER NORMAL COMPLETION	The compiler has successfully completed creating the object module from the supplied source program.	Continue with program preparation.
FUNCTION NAME NOT REFERENCED	The name of the FUNCTION subprogram that this message follows has not been referenced in the program.	Correct program to correctly reference the FUNCTION name, or delete the subprogram as unused.
LIBRARY I/O ERROR CODE = XXXX	A library error has occurred with one of the I/O devices; the code XXXX defines the type of error.	Consult volume 3 of the *DX10 Operating System Release 3 Reference Manual* to determine the cause associated with code XXXX.
OPEN DO LOOPS	Following this message, the compiler lists the DO loops in the program that do not provide for an exit from the loop. The list is in the form: statement number OPENED AT LINE line-number where line-number is the output listing line number.	Restructure DO loop to provide a termination point.
REGION NOT AVAILABLE	The memory resources of the system are not sufficient to provide the quantity of memory requested.	Reduce memory requested for compilation.
STORAGE OVERFLOW	This message indicates that a block of memory, either COMMON or noncommon, has become larger than the available memory in the system.	Restructure program to require a smaller block of memory.
TOO MANY NESTED COPIES	Copy statements have been nested deeper than five levels.	Reduce the level of nesting to five or less.
UNDEFINED LABELS	Following this message, the compiler lists the statement numbers of statements that have not been defined in the program. The list is in the form: statement number FIRST REFED AT LINE line-number where line-number is the output listing line number of the statement that first references the undefined statement.	Correct erroneous reference, or provide a statement corresponding to the label.
WARNING – INVALID COMPILER OPTION CODE = XXXX	One or more of the selected compiler option characters entered for this compilation is not a recognized option character. The code indicates the hexadecimal code (ASCII) for the invalid character.	Correct invalid option character (compiler will function properly without correction, but will not act on the invalid option).

Table 9-3. FORTRAN Runtime Package Diagnostic Messages

Message	Meaning	Corrective Action
INSUFFICIENT PRB SPACE	The system table space of the operating system is inadequate for the number of Physical Record Blocks (PRBs) required for the I/O units assigned in the program.	Reduce the number of FORTRAN units assigned. -or - Regenerate the operating system with an expanded table space reserved for the FORTRAN runtime package.*
LIBRARY I/O ERROR CODE = XXXX	A library error has occurred with one of the I/O devices; the code XXXX defines the type of error.	Consult the DX10 operating system documentation to determine the cause associated with code XXXX.
NORMAL PROGRAM COMPLETION	The program has successfully completed execution.	None required.
TOO MANY UNIT ASSIGNMENTS	The system table space of the operating system is inadequate for the number of units assigned in the FORTRAN program.	Reduce the number of FORTRAN units assigned. –or– Expand the table space reserved for the FORTRAN runtime package.*
WARNING – BACKSPACE AFTER REWIND UNIT = unit number**	The program includes a series of I/O commands that requests the indicated unit to perform a backspace operation after it has been rewound to the beginning of the file. The unit number parameter is the FORTRAN unit number assigned to the device in the program.	Restructure I/O commands to eliminate erroneous condition.
WARNING – DEFINED LOGICAL RECORD LENGTH > CREATED LOGICAL RECORD LENGTH. TRUNCATED	The logical record length specified upon creation is less than the logical record length specified by the DEFINE FILE statement.	Recreate the file with a sufficiently large logical record length.
WARNING – LOGICAL RECORD LENGTH > MAX BUFFER SIZE. TRUNCATED UNIT = unit number**	The logical record length specified for the indicated FORTRAN I/O unit is greater than the maximum allowable buffer size of 135 characters; characters in excess of 135 will not be recognized or printed.	Reformat logical records to remain within the maximum buffer size of 135 characters.
WARNING - REDUNDANT BUFFER I/O STATUS CHECK UNIT = unit number**	The program includes a series of buffer I/O routines that checks the status of a unit twice without initiating a BUFIN or a BUFOUT call.	Restructure I/O sequence or delete redundant IUNIT status check.

*Expanding the table space requires a copy of the source code for the FORTRAN runtime package. See paragraph E.4.

**Unit number is the hexadecimal value of the logical unit number.

9.5 RUNTIME ERROR TRACEBACK INFORMATION

The standard subprogram linkage creates a linked list of information concerning active subprogram calls. This list provides debugging information to the FORTRAN runtime package. When it detects an error, the runtime package prints the following information on the LIST device to aid in debugging:

- An error message to indicate the type of error that occurred
- The FORTRAN routine that produced the error
- The location of the CALL for the routine
- The location of other higher level nested CALLs

For example, the runtime error message:

```
FLOATING POINT, DIVIDE BY ZERO ERROR AT >001A TRACE3
                                 CALLED AT >0014 IN TRACE2
                                 CALLED AT >0014 IN $MAIN
```

indicates that an error occurred in statement location $001A_{16}$ in subroutine TRACE3, that TRACE3 was called at statement location 0014_{16} in TRACE2, and that TRACE2 was called at statement location 0014_{16} in the main program $MAIN. Runtime error messages produce the relative address and the module where the error occurred. Table 9-4 contains a list of Runtime Errors.

The data structures that provide the traceback facility are:

1. The BLWP instruction automatically threads the calls by storing the previous workspace pointer and program counter in registers 13 and 14 of each workspace.

2. The 16 words of each workspace are immediately followed by 3 words of traceback information:

   a. A pointer to a 6-character program name

   b. The entry address of the first instruction to be executed in the program

   c. The type of the program indicated by the following values:

      0 = Main program (top level)

      1 = Nonreentrant subprogram

      2 = Reentrant subprogram

      3 = FORTRAN runtime routine (skipped in traceback listing)

Table 9-4. FORTRAN Runtime Error Messages

Error		Type
BACKSPACE END FILE REWIND	*	End of file File previously open Illegal operation Illegal unit I/O
BUFIN BUFOUT IUNIT	*	Illegal unit Illegal operation File previously open Incorrect number of arguments
DEFINE FILE FIND	*	Illegal unit Illegal operation End of file File previously open I/O
END VECTOR TAKEN		–
FLOATING POINT		Divide by zero Illegal code Illegal instruction Overflow Underflow
FORMATTED		End of file File previously opened Illegal count Illegal format character Illegal input character Illegal operation Illegal repeat count Illegal unit Integer input overflow I/O No conversion Numeric/Logical Paren mismatch Paren nesting >3 Real input overflow
SVC		Illegal SVC call
SUBPROGRAM		Incorrect number of arguments

*Any of these error messages could result from the specified types of errors.

Table 9-4. FORTRAN Runtime Error Messages (Continued)

Error	Type
UNFORMATTED	Illegal unit Illegal operation End of file File previously open I/O
FUNCTION NAME	Argument too large Both arguments zero Division by zero Incorrect number of arguments Negative argument Overflow Singularity

TASK TERMINATED
One of the fifteen task errors listed below.

WP=xxxx PC=xxxx <PC>=xxxx ST=xxxx

WORKSPACE REGISTERS

0 - 7 xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
8 - 15 xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx

TASK ERRORS:

1. Nonrecoverable memory parity error
2. Tried to execute an undefined instruction
3. Accessed an illegal tiline address
4. Illegal supervisor call code
5. Tried to access an address outside of the task memory area
6. Tried to execute a privileged instruction
7. Terminated by a kill task
8. Installed memory configuration is too small to load task
9. Map segment not present in memory
10. Execute protection violation
11. Wrote to a write protected segment
12. Stack overflow
13. Hardware breakpoint address error
14. Time out error
15. Overflow protection violation

REENTRANT	Data area overflow

APPENDIX A

FORTRAN SOURCE PROGRAM CHARACTER SET

The source program character set for 990 FORTRAN is shown in table A-1. The table shows the hexadecimal value and decimal value of the ASCII code for each character, the character itself, and the corresponding Hollerith (punched card) code.

Table A-1. Character Set

Hexadecimal Value	Decimal Value	Character	Hollerith (Punched Card) Code
20	32	Space	Blank
21	33	!	12-8-7
22	34	"	8-7
23	35	#	8-3
24	36	$	11-8-3
25	37	%	0-8-4
26	38	&	12
27	39	'	8-5
28	40	(	12-8-5
29	41	)	11-8-5
2A	42	*	11-8-4
2B	43	+	12-8-6
2C	44	,	0-8-3
2D	45	-	11
2E	46	.	12-8-3
2F	47	/	0-1
30	48	0	0
31	49	1	1
32	50	2	2
33	51	3	3
34	52	4	4
35	53	5	5
36	54	6	6
37	55	7	7
38	56	8	8
39	57	9	9
3A	58	:	8-2
3B	59	;	11-8-6
3C	60	<	12-8-4
3D	61	=	8-6
3E	62	>	0-8-6
40	64	@	8-4
41	65	A	12-1
42	66	B	12-2
43	67	C	12-3
44	68	D	12-4
45	69	E	12-5
46	70	F	12-6

Table A-1. Character Set (Continued)

Hexadecimal Value	Decimal Value	Character	Hollerith (Punched Card) Code
47	71	G	12-7
48	72	H	12-8
49	73	I	12-9
4A	74	J	11-1
4B	75	K	11-2
4C	76	L	11-3
4D	77	M	11-4
4E	78	N	11-5
4F	79	O	11-6
50	80	P	11-7
51	81	Q	11-8
52	82	R	11-9
53	83	S	0-2
54	84	T	0-3
55	85	U	0-4
56	86	V	0-5
57	87	W	0-6
58	88	X	0-7
59	89	Y	0-8
5A	90	Z	0-9
5B	91	[	12-8-2
5C	92	\	0-8-2
5D	93	]	11-8-2
5E	94	∧(↑)	11-8-7
5F	95	_	0-8-5

APPENDIX B

CALLING SEQUENCES BETWEEN FORTRAN AND ASSEMBLY LANGUAGE PROGRAMS

B.1 GENERAL

FORTRAN programs and subroutines employ standard calling sequences. By adhering to these standard conventions, the programmer may write FORTRAN programs to call assembly language subroutines. This may be accomplished by following the calling sequences provided in paragraphs B.2 through B.4. The remaining paragraphs describe a set of macros that allow the programmer to write assembly language routines that may be called by FORTRAN programs or that call FORTRAN subprograms. Macros are supported only by the DX10 assembler.

B.2 NONREENTRANT ROUTINES

The compiler generates the following calling sequence for linkage to a nonreentrant subprogram.

Source:

CALL SUB (A_1,A_2, . . . ,A_n)

Generated Code:

```
BLWP    @SUB
DATA    n
DATA    A1
DATA    A2
  .
  .
  .
DATA    An
```

The parameters are passed as a vector of 16-bit addresses following the linkage to the subprogram. If the low order bit of the parameter word is zero, the parameter word is the word address of the parameter value. If the low order bit is one, the parameter is the word address of a 16-bit pointer that indicates the location of the parameter value.

Example:

CALL JOE (I,K(J))

The system first computes the address of K(J), stores the address in a temporary word and passes a pointer (with the low order bit set) to the temporary location TEMP:

```
MOV     @J,R7
AI      R7,K-2
MOV     R7,@TEMP
BLWP    @JOE
DATA    2
DATA    I
DATA    *TEMP
```

where * is "indirect" (low order bit = 1).

The receiving sequence for nonreentrant subroutines is as follows:

```
         IDT      'SUB111'      SUBROUTINE NAME MUST CONTAIN 6
                                CHARACTERS
         DEF      SUB111
         REF      F$RGMY
SUB111   DATA     WS            WORKSPACE ADDRESS
         DATA     START         ENTRY POINT ADDRESS
START    BL       @F$RGMY
         DATA     N             NUMBER OF ARGUMENTS
         DATA     T             POINTER OF ARGUMENT ADDRESS LIST
NAME     TEXT     'SUB111'      NAME OF SUBROUTINE
         .
         .
         USER SUPPLIED ASSEMBLY LANGUAGE CODE
         .
         .
         RTWP                   RETURN TO FORTRAN PROGRAM
WS       BSS      32            WORKSPACE
         DATA     NAME          PROGRAM NAME ADDRESS
         DATA     START         SUBROUTINE ENTRY ADDRESS
         DATA     1             NONREENTRANT FLAG
T        DATA     0             ADDRESS OF ARGUMENT A1
         DATA     0             ADDRESS OF ARGUMENT A2
         DATA     0             ADDRESS OF ARGUMENT An
         .
         .
         ADDITIONAL USER SUPPLIED DATA MAY BE INSERTED HERE
         .
         .
         END
```

@F$RGMY is a FORTRAN routine that transfers the parameter addresses from the calling sequence to a series of words beginning at T. Location T contains the address (no indirect) of A_1. T+2 contains the address of A_2, and T+2 (N-1) contains the address of A_n. The user must allocate the data area required for these addresses. But the user does not have to be concerned with whether or not the low order bit is set. @F$RGMY handles the conversion to direct addresses.

The name of the subroutine is included as part of the debugging trace back information. This name must be six characters in length, left-justified and blank-filled. The three words immediately following each workspace area will be modified by @F$RGMY to contain the address of the subroutine's name, entry point address, and the reentrant flag. The reentrant flag is set to one for nonreentrant subroutines and to zero for reentrant subroutines.

To further assist in the debugging of the program, the location of the arguments relative to the main program origin may be determined by the following method: add the location of the variable (see the allocation section of the FORTRAN compiler listing) to the origin of the $DATA module (the $DATA module is found in the Load Map Listing of the Link Editor).

B.3 REENTRANT ROUTINES

Subprograms which are to be used recursively have a receiving sequence which differs from non-reentrant subprograms. This receiving sequence is generated by compiling the subprogram using the FORTRAN R option. The code generated for such a subprogram is:

```
*       FUNCTION FACTOR(N)
                   IDT       'FACTOR'
                   DEF       FACTOR
                   REF       F$RGMY,F$XRER
FACTOR             DATA      >0038           NUMBER OF DATA BYTES NEEDED
                   DATA      START           STARTING ADDRESS
START              BL        @F$RGMY
                   DATA      >8002           NUMBER OF ARGUMENTS + >8000
                   DATA      T#0             POINTER(RELATIVE TO R10) TO
                                             ARGUMENT ADDRESS LIST
                   TEXT      'FACTOR'        SUBPROGRAM NAME TEXT STRING
        .
        .
        .
*       END
                   BL        @F$XRER         RETURN FROM REENTRANT CALL
```

Refer to Appendix E, paragraph E.3.6 for information describing the use of recursive FORTRAN subprograms.

B.4 PROGRAM LINK EDIT

The link control file should conform with link edit standard format for FORTRAN. The following example gives the order of the INCLUDE statements in the Link Edit Control File.

```
INCLUDE (ACCESS NAME OF FORTRAN OBJECT OUTPUT)
INCLUDE (ACCESS NAME OF SDSMAC OBJECT OUTPUT)
```

NOTE

The FORTRAN object must be the first include in the control file.

See Appendix H for additional DX10 3.X link editing information and Appendix G for TXDS link editing information.

B.5 CREATING THE MACROS

Macros are provided to aid in producing the proper receiving code sequences when subroutines called by FORTRAN are written in assembly language and the DX10 Macro Assembler is used. In addition, assembly language subroutines that call FORTRAN subroutines can use these macros to produce the proper calling sequences.

Before using these macros, the user must create a file in which to place the desired macros. Macros are not provided with the FORTRAN package. The macros that are required must be entered in the macro file by the user from the macro formats contained in paragraphs B.5.1 and B.5.2.

If any macro listed in the MACROs column is to be used, the user must create a file that contains this macro. In addition the file must contain the macros listed in the corresponding CALLED MACROs column.

Macros	Called Macros
FTNINT	UNIQUE, PADSTR
FTNSUB	FPMEQU, UNIQUE, FPMGEN, PADSTR
FTNCAL	FPMDAT
RSUB	PADSTR
RCALL	FPMDAT
FTNRET	

B.5.1 USER CALLABLE MACROS. The following paragraphs describe macros that implement the FORTRAN call structure.

B.5.1.1 FTNCAL Macro. The FTNCAL macro generates a FORTRAN subroutine call. FTNCAL support macros are found in paragraph B.5.2.1.

Format:

```
FTNCAL $MACRO S,P
       REF    :S:
       BLWP   :S:
       UNL
       $VAR NP
       $IF P.A&$PCALL            P PRESENT ?
       $IF P.A&$POPL
         $ASG P.V TO NP .V
       $ELSE
         $ASG 1 TO NP .V
       $ENDIF

  $ENDIF
  LIST
  DATA    :NP .V:
  UNL
  FPMDAT :P:
  LIST
  $END
```

B.5.1.2 FTNINT Macro. The FTNINT macro is used to set the FORTRAN runtime initialization whenever FORTRAN subprograms are to be used and a FORTRAN main program is not included. FTNINT support macros are found in paragraphs B.5.2.2 and B.5.2.3.

Format:

```
FTNINT        $MACRO S                SET  FORTRN  R/T  INITIALIZATION
              REF F$XPRE,F$REVP
              $VAR T,U,WP,PC,N
              $ASG'UNIQUE' TO U.S
              UNIQUE
              $ASG :U.SS: TO PC.S
              UNIQUE
              $ASG :U.SS:  TO WP.S
              DATA :WP.S:
              DATA :PC.S:
              DATA F$REVP
```

```
:WP.S:          BSS 32
                UNIQUE
                $ASG :U.SS: TO N.S
                DATA :N.S:
                DATA :PC.S:
                DATA 0
                $ASG "":S.S:"" TO T.S
:PC.S:          BL @F$XPRE
:N.S:           TEXT :T.S:
                UNL
                PADSTR :T.L:
                LIST
                $END
```

B.5.1.3 FTNRET Macro. The FTNRET macro generates a reentrant return.

Format:

```
FTNRET $MACRO
       REF      F$XRER          GENERATES REENTRANT RETURN
       BL       @F$XRER
       $END
```

B.5.1.4 FTNSUB Macro. The FTNSUB macro generates a FORTRAN callable subprogram. FTNSUB support macros are found in paragraphs B.5.2.2, B.5.2.3, B.5.2.4, and B.5.2.5.

Format:

```
FTNSUB  $MACRO S,P
        UNL
        $VAR U,WS,TX,PC,WK
        $ASG 'UNIQUE' TO U.S
        $ASG 'FTN;WS' TO WK.S        WORKSPACE SYM SAVE
*
  $IF   WK.SV=0
        UNIQUE
        $ASG :U.SS: TO WS.S          LABEL FOR WORKSPACE
        $ASG :U.SS: TO WK.SS         FTNENTRY MACRO
   $ELSE
        $ASG :WK.SS: TO WS.S         USE PREVIOUSLY CREATED WKSP
   $ENDIF
        UNIQUE
        $ASG :U.SS: TO TX.S          LABEL FOR TEXT STRING
        UNIQUE
        $ASG :U.SS: TO PC.S          LABEL FOR INITIAL PC
        LIST
        DEF :S:
:S:     DATA :WS.S:,:PC.S:           ENTRY VECTOR
*
        $IF WK.SV=0                  IF FIRST INVOCATION
:WS.S:  BSS 32                       WORKSPACE
            DATA :TX.S:,:S:,1        TRACEBACK VECTOR
            $ASG 1 TO WK.SV          SET FLAG
        $ENDIF
```

```
*
         $VAR NP,CT,MP                GET NUMBER OF CALLING PARAMETERS
         $ASG 'COUNT;' TO CT.S
      $IF P.A&$PCALL                  P PRESENT ?
         $IF P.A&$POPL
           $ASG P.V TO NP.V
         $ELSE
           $ASG P.V TO NP.V
         $ENDIF
      $ENDIF
*
 $IF     NP.V>MP.SV                   IF MORE PARAMETERS THIS TIME THAN
                                      LAST
         $ASG NP.V TO MP.SV
         UNL
         UNIQUE
         $ASG :U.SS: TO MP.SS
:MP.SS:     EQU $
         FPMGEN :P:
   $ELSE                              USE THE PREVIOUSLY DEFINED AREA
         $ASG 0 TO CT.SV
*        UNL
         FPMEQU :P:
   $ENDIF
*
         LIST
         REF F$RGMY
:PC.S:   BL @F$RGMY                   CRACK ARGUMENT ADDRESSES
         DATA :NP.V:
*
   $IF   NP.V=0                       IF NO PARAMETERS
         DATA 0
   $ELSE
         DATA :MP.SS:
   $ENDIF
*
   $VAR T
   $IF S.L<7                          IF SUB NAME IN RANGE
   $ASG "":S.S:"" TO T.S
:TX.S:   TEXT :T.S:
         UNL
         PADSTR :T.L:
         LIST
   $ELSE
         TEXT 'SUBERR'                SUBROUTINE NAME TOO LONG
   $ENDIF
*
         $END
FTN; WS  EQU 0
COUNT;   EQU 0
MAX;;;   EQU 0
```

B.5.1.5 **RSUB Macro.** The RSUB macro generates a reentrant FORTRAN callable subroutine entry. RSUB support macros are found in paragraph B.5.2.3.

Format:

```
RSUB    $MACRO  S,NB,NP,DP
*
        DEF   :S:
:S:     DATA :NB:
        DATA $+2
        REF  F$RGMY
        BL @F$RGMY
*
   $IF NP.A&$PCALL                  IF NUMBER PARMS PRESENT EQUALS
        DATA :NP.V:++ 8000          NUMBER PARMS + REENTRANT FLAG
   $ELSE
        DATA 8000                   NO PARAMETERS SPECIFIED
   $ENDIF
*
   $IF DP.A&$PCALL                  IF DISPLACEMENT FOR PARMS PRESENT
   $IF  DP. SA &$REL
        DATA :DP:                   WARNING – SYMBOL RELOCATABLE
   $ELSE
        DATA :DP:
   $ENDIF
   $ELSE
        DATA 0
   $ENDIF
*
        $VAR  T
        $IF S.L<7                   IF SUB NAME LESS EQUAL 6 CHARS
        $ASG "":S.S:"" TO T.S
          TEXT  :T.S:
          UNL
          PADSTR  :T.L:
          LIST
        $ELSE
          TEXT  'SUBERR'            SUBROUTINE NAME TOO LONG
        $ENDIF
        $END
```

B.5.1.6 **RCALL Macro.** The RCALL macro generates a reentrant subroutine call. RCALL support macros are found in paragraph B.5.2.1.

Format:

```
RCALL   $MACRO S,P
        REF F$XREC,  :S:
        BL  @F$XREC
        DATA  :S:
        UNL
        $VAR NP
```

```
$IF P.A&$PCALL
  $IF P.A&$POPL
    $ASG 1 TO NP.V
  $ENDIF
$ENDIF
LIST
DATA :NP.V:
UNL
FPMDAT :P:
LIST
$END
```

B.5.2 **SUPPORT MACROS.** The following support macros are called by the user-called macros to simplify certain specific functions.

B.5.2.1 **FPMDAT** Macro. The FPMDAT macro generates a data statement.

Format:

```
FPMDAT $MACRO P1, P2                     GENERATE DATA STATEMENTS
*
          $IF P1.A&$P CALL               IF PARAMETER NON-NULL
            LIST
            DATA :P1.S:
            UNL
            FPMDAT :P2:
          $ENDIF
*
          $END
```

B.5.2.2 **UNIQUE** Macro. The UNIQUE macro generates unique variable names in the string of the symbols table.

Format:

```
UNIQUE $MACRO D
       $VAR N,V
       $ASG 'UNIQUE' TO N.S              SYMBOL "UNIQUE"
       $ASG 'N;;VAL' TO V.S
       $ASG 'N;;':V.SV: TO N.SS
       $ASG V.SV+1 TO V.SV
       $END
N;;VAL EQU 0
```

B.5.2.3 **PADSTR** Macro. The PADSTR macro generates blanks to pad a string of characters to a maximum of six characters plus two quotes.

```
PADSTR $MACRO ST                         GENERATES BLANKS TO PAD STRING
```

```
*
          $IF ST.V<8                     SIX CHARACTERS + 2 QUOTES
           LIST
           TEXT ' '
           UNL
           PADSTR :ST.V:+1
          $ENDIF
*
          $END
```

B.5.2.4 **FPMEQU Macro.** The FPMEQU macro generates EQUS for the FORTRAN parameters.

Format:

```
FPMEQU $MACRO  P1,P2
*
          $IF P1.A&$PCALL                IF PARAMETER ONE PRESENT
           $VAR CT,MP
           $ASG 'COUNT;' TO CT.S
           $ASG 'MAX;;;' TO MP.S
           LIST
:P1:       EQU :MP.SS:+:CT.SV:           ADDRESS OF PARAMETER :P1:
           UNL
           $ASG CT.SV+2 TO CT.SV
           FPMEQU :P2:
          $ENDIF
*
          $END
```

B.5.2.5 **FPMGEN Macro.** The FPMGEN macro generates a BSS instruction for a parameter list.

Format:

```
FPMGEN $MACRO  P1, P2
*
          $IF P1.A&$PCALL                IF CALLED WITH NON-NULL PARAMETERS
           LIST
:P1:       BSS 2                         ADDRESS OF PARAMETER (P1)
           UNL
           FPMGEN :P2:
          $ENDIF
*
          $END
```

B.6 USING THE MACROS

An assembly language routine can call a FORTRAN subprogram or can be called by a FORTRAN program. The following paragraphs describe the macros that handle the calling sequences for these tasks. The macros described below are another method of implementing the calling and receiving sequences discussed in paragraphs B.2 through B.4.

B.6.1 NONREENTRANT SUBROUTINES. The following paragraphs discuss the macros used for linkage to nonreentrant subprograms.

B.6.1.1 FTNINT. The FTNINT macro generates the FORTRAN runtime initialization necessary to call a FORTRAN subroutine from an assembly language program. Note that this macro (or its equivalent coding) is required whenever a FORTRAN main program will not be used.

The format of the call is:

FTNINT S

where:

S is the name of the FORTRAN subroutine (see paragraphs B.6.1.3 and B.6.2.2 for examples).

NOTE

This macro must be invoked before the first call to a FORTRAN subroutine is made.

B.6.1.2 FTNSUB. The FTNSUB macro generates a FORTRAN callable subprogram. The macro is used at the beginning of the assembly language subprogram that is to be called by a FORTRAN program. FTNSUB builds an entry vector consisting of a workspace, entry point and end action for the subprogram.

The format of the call is:

FTNSUB S, P1, ... , Pn

where:

S is the label for the entry point into the assembly language subprogram

P1-Pn are parameters.

The FORTRAN program shown in figure B-1 calls an assembly subroutine Test shown in figure B-2.

FTNSUB is a macro that passes the entry point and two parameters to the assembly language subroutine. The entry point name from the macro FTNSUB must be the name used in the FORTRAN call.

The entry point TEST is not seen in the assembly language subroutine since it is built in the macro processing and the label is used there. The subroutine TEST opens ST01, sends the value of the first parameter to ST01, and closes ST01.

```
IPAR1=11
IPAR2=22
CALL TEST (IPAR1,IPAR2)
END
```

Figure B-1. FORTRAN Program

```
        UNL
        COPY     MACRO
        LIST
        IDT      'TEST'
*
        DXOP     SVC,15
*
        FTNSUB   TEST,IPAR11,IPAR22
        MOVB     @OPEN,@OPCODE
        SVC      @PRB                 OPEN
        MOV      @IPAR11,R1
        MOV      *R1,R0
        SVC      @CONVRT              CONVERT PARAMETER FROM BINARY TO
                                      DECIMAL
        MOV      @HEX,@BUFFER         PUT DECIMAL NUMBER IN BUFFER
        MOV      @WRITE,@OPCODE
        MOV      @ADDRS,@ADDR
        SVC      @PRB                 ASCII, WRITE
        MOVB     @CLOSE,@OPCODE
        SVC      @PRB                 CLOSE
        RTWP
PRB     DATA     0
OPCODE  BYTE     0,0
        DATA     0
ADDR    DATA     0
        DATA     0
        DATA     10
        BYTE     0,0
        DATA     0
        BYTE     0,0
        DATA     0,0
        DATA     NAME
NAME    BYTE     4
        TEXT     'ST01'
OPEN    BYTE     >80
WRITE   BYTE     0B
CLOSE   BYTE     >81
ADDR    DATA     BUFFER
BUFFER  DATA     IPAR11
CONVRT  BYTE     A,0                  BINARY TO DECIMAL CONVERSION
HEX     BYTE     0,0,0,0,0,0
        END
```

Figure B-2. Assembly Language Subroutine TEST

B.6.1.3 FTNCAL. The FTNCAL macro generates the call to the FORTRAN subroutine. The format of the call is:

FTNCAL S, P1, . . . , Pn

where:

S is the name of the FORTRAN subroutine

P1 - Pn are parameters.

Figure B-3 is an assembly language program TEST1 that calls a FORTRAN subroutine TEST2, figure B-4. The macro FTNINT is used to pass the subroutine name and sets up the FORTRAN runtime initialization for the subroutine. The macro FTNCAL calls the FORTRAN subroutine and passes the name and two parameters to the FORTRAN subroutine. The subroutine TEST2 writes the value of the two parameters and 'TEST COMPLETE' on FORTRAN UNIT17.

B.6.2 REENTRANT SUBROUTINES. For the linkage to reentrant subprograms, the macros discussed in the following paragraphs are used.

NOTE

See Appendix E, paragraph E.3.6 for information about interface routines F$XREC and F$XRER.

```
        UNL
        COPY MACRO
        LIST
        IDT  'TEST1'
        DXOP SVC,15
TEST1   FTNINT TEST1
        FTNCAL TEST2,(IPAR1,IPAR2)
        SVC    @ENDTSK
ENDTSK  BYTE   04
IPAR1   DATA   1
IPAR2   DATA   2
        END
```

Figure B-3. TEST1. Example of FTNINT, FTNCAL in an Assembly Language Program

```
        SUBROUTINE TEST2 (IPAR11,IPAR22)
        WRITE (17,90) IPAR11,IPAR22
90      FORMAT (1X, I4, 1X, I4)
        WRITE (17,100)
100     FORMAT (1X, 'TEST COMPLETE')
        RETURN
        END
```

Figure B-4. TEST2. A FORTRAN Subroutine

B.6.2.1 RSUB. The RSUB macro generates a reentrant FORTRAN callable subroutine. It is placed at the beginning of the called assembly language program.

The format of the call is:

```
RSUB S,NB,NP,DP
```

where:

S is the entry point in the assembly language program

NB is the number of bytes needed for temporary data and workspace in the assembly language program

NP is the number of parameters to be passed

DP is the displacement of external references specifically to access the parameters.

The FORTRAN program in figure B-5 calls the reentrant assembly language subroutine TEST. RSUB passes the entry point of the assembly language subroutine, the number of bytes needed for temporary data, the number of parameters, and the displacement of the external references (DISPLC=R10) in the subroutine (figure B-6). In the FORTRAN subroutine, the name used in the call must be the same as the name of the entry point in RSUB. The macro FTNRET causes control to be returned to the calling program. Subroutine TEST opens the ST01 writes the value of the first parameter to the ST01 and closes the ST01.

NOTE

See Appendix E, paragraph E.3.6 for information about interface routines F$XREC and F$XRER.

NOTE

The REF directive must be used in order to access the parameters.

```
REENTRANT TEST
IPAR1=11
IPAR2=22
CALL TEST (IPAR1, IPAR2)
END
```

Figure B-5. FORTRAN Program

```
        UNL
        COPY    MACRO
        LIST
*       IDT     'TEST'

        REF     IPAR1,IPAR2
        DXOP    SVC,15
*
        RSUB    TEST,50,2,DISPLC
        MOV     @DISPLC,R10
        MOVB    @OPEN,@OPCODE
        SVC     @PRB                    OPEN
        MOV     @IPAR1(R10),R1
        MOV     *R1,R0
        SVC     @CONVRT                 CONVERT PARM FROM BINARY TO DEC
        MOV     @HEX,@BUFFER            PUT DEC NUMBER IN BUFFER
        MOVB    @WRITE,@OPCODE
        MOV     @ADDRS,@ADDR
        SVC     @PRB                    ASCII WRITE
        MOVB    @CLOSE,@OPCODE
        SVC     @PRB                    CLOSE
        FTNRET
DISPLC  DATA    2000
PRB     DATA    0
OPCODE  BYTE    0,0
        DATA    0
ADDR    DATA    0
        DATA    0
        DATA    10
        BYTE    0,0
        DATA    0
        BYTE    0,0
        DATA    0,0
        DATA    NAME
NAME    BYTE    4
        TEXT    'ST01'
OPEN    BYTE    >80
WRITE   BYTE    0B
CLOSE   BYTE    >81
ADDRS   DATA    BUFFER
BUFFER  DATA    IPAR1
CONVRT  BYTE    A,0                     BIN TO DEC CONVERSION
HEX     BYTE    0,0,0,0,0,0
        END
```

Figure B-6. Assembly Language Subroutine TEST Using RSUB and FTNRET

B.6.2.2 RCALL. RCALL is used to call a reentrant FORTRAN subprogram compiled using the R (reentrant) option. FTNINT must be called before RCALL is used to generate a reentrant subroutine call.

The format of the call is:

RCALL S,P1,. . ., Pn

where:

S is the name of FORTRAN subroutine

P1-Pn are parameters.

Figure B-7 is an assembly language program TEST1 that calls the reentrant (compiled with the R option) FORTRAN subroutine TEST2 in figure B-8. The macro FTNINT passes the subroutine name and sets up FORTRAN runtime initialization that is used by the reentrant FORTRAN subroutine. The macro RCALL is used in making the call to the reentrant FORTRAN subroutine; with this macro, the subroutine name and two parameters are passed to the subroutine.

```
         UNL
         COPY    MACRO
         LIST
         IDT     'TEST1'
         DATA    WS,ENTRY+38,0
         BSS     32
         DXOP    SVC,15
ENTRY    FTNINT  TEST2
         RCALL   TEST2,IPAR1,IPAR2
         SVC     @ENDTSK
ENDTSK   BYTE    04
IPAR1    DATA    1
IPAR2    DATA    2
         END
```

Figure B-7. Assembly Language Program TEST1

The subroutine TEST1, figure B-8, writes the value of the two passed parameters, and displays 'TEST COMPLETE' to FORTRAN UNIT17.

```
        SUBROUTINE TEST 2 (IPAR11,IPAR22)
        WRITE (17,90) IPAR11,IPAR22
90      FORMAT (1X,14,1X,14)
        WRITE (17,100)
100     FORMAT (1X, 'TEST COMPLETE')
        RETURN
        END
```

Figure B-8. FORTRAN Subroutine TEST2

NOTE

See Appendix E, paragraph E.3.6 for information about interface routines F$XREC and F$XRER.

B.6.2.3 **FTNRET.** FTNRET (see paragraph B.6.2.1) is used to generate a reentrant return from an assembly language subprogram. The form of the call is:

FTNRET

B.6.3 **ASSEMBLING.** Macros are supported only by SDSMAC. If a macro is used in a program, the copy statement must be invoked to include the macro definition.

The format of this statement is:

COPY pathname

where:

pathname is the pathname of the file containing the macro definitions.

APPENDIX C

FLOATING POINT ARITHMETIC PACKAGE

C.1 INTRODUCTION

The 990 FORTRAN floating point arithmetic package consists of 8 modules that implement the following functions:

- Convert integer to floating point.
- Convert floating point to integer.
- Single precision floating point addition and subtraction.
- Double precision floating point addition and subtraction and double precision fraction addition.
- Single precision floating point multiplication.
- Double precision floating point multiplication.
- Single precision floating point division.
- Double precision floating point division.

In general, the calling sequence for these routines consists of the following code:

BL @(subroutine name)

error return (usually a JMP instruction)

normal return

Parameters usually consisting of argument addresses are passed in the workspace registers. If exception conditions occur (such as overflow, or underflow), a code is returned in register 8.

Error codes include the following:

Code	Meaning
8000	Exponent overflow (number too large)
4000	Exponent underflow (number too small)
2000	Attempt to divide by zero

The following paragraphs discuss the 8 modules in the floating point package and the routines in each module.

C.2 INTEGER TO FLOATING POINT CONVERSION MODULE

This module occupies 82 (52_{16}) words of memory and contains routines to convert from either integer or extended (32 bit) integer to either single precision or double precision floating point. The routines are:

Routine Name	Function
F$XCIR	Integer to single precision
F$XCID	Integer to double precision
F$XCER	Extended integer to single precision
F$XCED	Extended integer to double precision

The routines do not return an error. The calling sequence is:

```
                BL      @F$XCIR/F$XCID/F$XCER/F$XCED
error return    NOP
normal return
```

Source address: R9

Destination address: R10

Registers used: R0 – R10

The address of the integer to be converted is passed in register 9. The address of the floating point output is passed in register 10. Registers 0 – 8 are used for intermediate storage and their original contents destroyed.

C.3 FLOATING POINT TO INTEGER CONVERSION MODULE

The floating point to integer conversion module occupies 98 (62_{16}) words of memory, and contains routines to convert from single precision or double precision floating point to either integer or extended (32 bit) integer. The routines are:

Routine Name	Conversion Function
F$XCRI	Single precision to integer
F$XCRE	Single precision to extended integer
F$XCDI	Double precision to integer
F$XCDE	Double precision to extended integer

If the integer part of the floating point number is too large to fit in 15 bits (for integer conversion) or 31 bits (for extended conversions), then an overflow exception condition exists and the error return is taken. Otherwise, the conversion proceeds normally and the normal return is taken. Thus, the calling sequence for these routines is:

```
                BL      @F$XCRI/F$XCRE/F$XCDI/F$XCDE
error return    JMP     exception processor
normal return
```

Registers used: R8 – R10

The address of the floating point number to be converted is passed in register 9. The address of the integer output is passed in register 10. Register 8 contains the exception code (if an exceptional condition occurs).

C.4 SINGLE PRECISION FLOATING POINT ADDITION AND SUBTRACTION MODULE

This module occupies 178 ($B2_{16}$) words of memory and contains routines to add and subtract single precision floating point numbers. The routines are:

Routine Name	Function
F$XAR	Single precision addition
F$XSR	Single precision subtraction

The calling sequence for addition is:

```
                   BL      @F$XAR
error return       JMP     exception processor
normal return

Registers used:    R0 – R10, R12
```

Register 9 (R9) is used to pass the address of the addend. Register 10 (R10) is used to pass the address of the augend. After addition, the sum replaces the augend.

	augend	address in R10
+	addend	address in R9
	sum	replaces augend (address in R10)

Register R0 - R9, and R12 are used for intermediate results and their previous contents are destroyed. The sum is normalized. Nonnormalized operands may be used if the radix point for both operands is correctly aligned. A sum fraction of zero causes a return of true zero.

The calling sequence for subtraction is:

```
                   BL      @F$XSR
error return       JMP     exception processor
normal return

Registers used     R0 - R10, R12
```

Register 9 (R9) is used to pass the address of the subtrahend. Register 10 (R10) is used to pass the address of the minuend. After subtraction, the difference replaces the minuend.

minuend	address in R10
subtrahend	address in R9
difference	replaces minuend (address in R10)

Registers R0 - R8, and R12 are used for intermediate results and their previous contents destroyed. The difference is normalized. Nonnormalized operands may be used if the radix point for both operands is aligned. A difference fraction of zero, causes a return of true zero.

C.5 DOUBLE PRECISION ADDITION, SUBTRACTION, AND FRACTION ADDITION MODULE

This module occupies 147 ($F7_{16}$) words of memory and contains routines to add and subtract double precision floating point numbers and a routine to add double precision fractions. The routines are:

Routine Name	Function
F$XAD	Double precision addition
F$XSD	Double precision subtraction
ADDFRC	Double precision fraction addition

The calling sequence for addition is:

```
                 BL      @F$XAD
error return     JMP     exception processor
normal return
```

Registers used: R0 - R10, R12

Register 9 (R9) is used to pass the address of the addend. Register 10 (R10) is used to pass the address of the augend. After addition, the sum replaces the augend.

```
      augend      address in R10
  +   addend      address in R9
      sum         replaces augend (address in R10)
```

Registers R0 - R9, and R12 are used for intermediate results and their previous contents destroyed. The sum is normalized. Nonnormalized operands may be used if the radix point for both operands is correctly aligned. A sum fraction of zero causes a return of true zero.

The calling sequence for subtraction is:

```
                 BL      @F$XSD
error return     JMP     exception processor
normal return
```

Registers used: R0 - R10, R12

Register 9 (R9) is used to pass the address of the subtrahend. Register 10 (R10) is used to pass the address of the minuend. After subtraction, the difference replaces the minuend.

```
      minuend      address in R10
      subtrahend   address in R9
      difference   replaces minuend (address in R10)
```

Registers R0 - R8, and R12 are used for intermediate results and their previous contents destroyed. The difference is normalized. Nonnormalized operands may be used if the radix point for both operands is aligned. A difference fraction of zero causes a return of true zero.

The calling sequence for fraction addition is:

```
                BL      @ADDFRC
```

Registers used: R0 - R4, R10

As input, registers R0 - R3 contain the addend fraction. As output, registers R0 - R3 contain the sum fraction. R10 contains the address of the augend. R4 is used for the carry when output occurs.

```
            addend         - contained in R0 - R3
   +        augend         - address in R10
            carry      sum
   contained in R4     contained in R0 - R3
```

C.6 SINGLE PRECISION FLOATING POINT MULTIPLICATION MODULE

This module is 82 (52_{16}) words in length and contains a routine to multiply single precision floating point numbers. The routine is:

Routine Name	Function
F$XMR	Single precision multiplication

The calling sequence for multiplication is:

```
                      BL      @F$XMR
error return          JMP     exception processor
normal return
```

Registers used: R0 - R10, R12

Register R9 is used to pass the address of the multiplier. Register R10 is used to pass the address of the multiplicand. After multiplication, the product replaces the multiplicand.

```
            multiplicand   - address in R10
   X        multiplier     - address in R9
            product        - replaces multiplicand (address in R10)
```

Registers R0 - R8, and R12 are used for intermediate results and their previous contents destroyed. The product is normalized. The factors must be normalized.

This is a 32-bit operation. The factors are both 32 bits and the product is truncated to 32 bits.

C.7 DOUBLE PRECISION FLOATING POINT MULTIPLICATION MODULE

This module is 152 (98_{16}) words in length and contains a routine to multiply double precision floating point numbers. The routine is:

Routine Name	Function
F$XMD	Double precision multiplication

The calling sequence for multiplication is:

	BL	@F$XMD
error return	JMP	exception processor
normal return		

Registers used: R0 - R10, R12

Register R9 is used to pass the address of the multiplier. Register R10 is used to pass the address of the multiplicand. After multiplication, the product replaces the multiplicand.

	multiplicand	- address in R10
X	multiplier	- address in R9
	product	- replaces multiplicand (address in R10)

Registers R0 - R8, and R12 are used for intermediate results and their previous contents destroyed. The product is normalized. The factors must be normalized.

This is a 64-bit operation. The factors are both 64 bits and the product is truncated to 64 bits.

C.8 SINGLE PRECISION DIVISION MODULE

This module occupies 172 (AC_{16}) words of memory and contains a routine to divide single precision floating point operands. The routine is:

Routine Name	Function
F$XDR	Single precision division

The calling sequence for division is:

	BL	@F$XDR
error return	JMP	exception processor
normal return		

Registers used: R0 - R10, and R12

Register 9 is used to pass the address of the divisor. Register 10 is used to pass the address of the dividend. After division, the quotient replaces the dividend:

quotient replaces dividend (address in R10)

address in R9 divisor

dividend address in R10

Register R7 must contain the address of a two word work area.

Registers R0 - R8, and R12 are used for intermediate results and their previous contents destroyed. The input operands must be normalized, and the output quotient is normalized.

C.9 DOUBLE PRECISION DIVISION MODULE

This module occupies 266 ($10A_{16}$) words of memory and contains a routine to divide double precision floating point operands.

The routine is:

Routine Name	Function
F$XDD	Double precision division

The calling sequence for division is:

```
                    BL       @F$XDD
error return        JMP      exception processor
normal return

Registers used:     R0 - R10, R12
```

Register 9 is used to pass the address of the divisor. Register 10 is used to pass the address of the dividend. After division, the quotient replaces the dividend:

quotient   replaces dividend (address in R10)

address in R9 divisor

dividend   address in R10

Register R7 must contain the address of a five word work area.

Registers R0 - R8, and R12 are used for intermediate results and their previous contents destroyed. The input operands must be normalized, and the output quotient is normalized.

APPENDIX D

CHARACTER STRING MANIPULATION

D.1 INTRODUCTION

It is often desirable to handle character data in blocks rather than one word at a time. A user would like, for example, to be able to move a part of a card image without worrying about word boundaries or data types. There are many other uses; the following list is presented not as a limit to what the character string package can do, but as an aid for the user in formulating his own ideas.

- Intermixing character strings.
- Editing character data for human error.
- Searching for a particular character set.
- Arranging character data in ascending or descending order.

The approach to handling characters as long contiguous data strings produces a more readable and less error-prone code than handling them one word at a time.

This package was written with the intent of allowing the user as much freedom as possible in moving, comparing and validating character strings. With this in mind, the following conditions should be remembered by the user.

NOTE

The bounds of input and output arrays may be exceeded if care is not exercised in selecting and maintaining the indices for these arrays.

NOTE

Although this character string package allows input data of any type, the user must be aware that the system remembers the original data type of the parameters for functions. Errors will occur during compilation if subsequent function parameters are of a different type than the original.

D.2 SUBSTR SUBROUTINE

The SUBSTR routine gives the user capability of transferring data from one storage location to another in units of 8 bits (byte). Although this is most useful in the manipulation of character strings, the user may apply it to other data types as well.

The form of the subroutine call is:

```
CALL SUBSTR(STRA,IPOSA,LENA,STRB,IPOSB,LENB,IERR)
```

The required parameters for this call are defined as follows:

- STRA – The variable or array name that contains the data.
- IPOSA – The number of the first byte to be moved.
- LENA – The number of bytes to move from STRA.
- STRB – The variable or array name in which to store the data.
- IPOSB – The number of the first byte in which to store.
- LENB – The number of bytes to store in STRB. If LENB > LENA, the LENA bytes from STRA will be followed by LENB – LENA bytes containing the blank character (20 hexadecimal). If LENB < LENA, only LENB bytes will be moved.
- IERR – An error flag that returns the following values:

  0 - Normal completion.
  -1 - Invalid argument(s) passed to SUBSTR.

All parameters must be of the type INTEGER*2, except STRA and STRB, which may be of any data type and dimension. Parameters IPOSA and IPOSB must be in the following ranges:

1 <= IPOSA <= BYTE(STRA) , 1 <= IPOSB < BYTE(STRB)

where BYTE(STRA) is the length of STRA in bytes. Parameters LENA and LENB must be in the following ranges:

0 <= LENA <= BYTE(STRA) - IPOSA + 1
0 <= LENB <= BYTE(STRB) - IPOSB + 1

SUBSTR can only check to see that the lower bounds on these limits are satisfied. If a parameter is passed with a value less than its lower bound, SUBSTR will take the following action:

- The message INVALID ARGUMENT PASSED TO SUBSTR will be given the first time an error is found. This message will not be given for subsequent errors in the same task.
- The parameter IERR will be set to minus one and then control will return to the calling program without any further action.

It is the responsibility of the user to ensure that no parameters are passed whose values exceed their upper bound. Execution of a SUBSTR call with parameters that exceed their upper bound will lead to unpredictable results and possible termination of the task by the operating system.

Example:

```
C     THIS PROGRAM INSERTS THE CORRECT DATE INTO A MESSAGE
C
            INTEGER*2 A(4), B(9), E
            DATA A/'09/11/78'/
            DATA B/'TODAY IS ?????????'/
            CALL SUBSTR( A, 1, 8, B, 10, 9 ,E )
```

```
C
C    ARRAY B NOW CONTAINS THE STRING 'TODAY IS 09/11/78'
C    ARRAY A IS UNALTERED
C    VARIABLE E CONTAINS THE VALUE ZERO
C
          STOP
          END
```

Example:

```
C    THIS PROGRAM MOVES ELEMENTS 10 THROUGH 19 OF ARRAY SOURCE
C    INTO ELEMENTS 20 THROUGH 29 OF ARRAY TARGET
C
          REAL*4 SOURCE(50), TARGET(100)
          REAL (5,10)SOURCE,TARGET
     10   FORMAT ( 15(10F8.5) )
C
C    THE BYTE NUMBER OF ELEMENT I IN ANY ARRAY WHERE EACH ELEMENT
C    IS J BYTE LONG IS: J*(I-1)+1
C
          CALL SUBSTR( SOURCE, 37, 40, TARGET, 77, 40, IERR )
C
C    ELEMENTS 20 THROUGH 29 OF ARRAY TARGET HAVE NOW BEEN REPLACED
C    WITH THE VALUES OF ELEMENTS 10 THROUGH 19 OF ARRAY SOURCE.
C    IERR CONTAINS A ZERO INDICATING NORMAL COMPLETION
C
          STOP
          END
```

D.3 KOMSTR FUNCTION

The integer function KOMSTR allows the user to compare two character strings. Since each character occupies one byte (8 bits) of storage, a logical comparison is made byte-by-byte on the ASCII value of each character.

The form of the call is:

```
I = KOMSTR(STRA,IPOSA,LEN,STRB,IPOSB,IERR)
```

The required parameters for this call are defined as follows:

- STRA – The first character string may be any data type and can be a dimensioned variable.
- IPOSA – The INTEGER*2 variable or constant is the starting character position in the first character string.
- LEN – The INTEGER*2 variable or constant is the number of characters to compare.
- STRB – The second character string may be any data type and can be a dimensioned variable.
- IPOSB – The INTEGER*2 variable or constant is the starting character position in the second character string.

- IERR – The INTEGER*2 variable is the return error code:

  0 - Normal completion
  -1 - Invalid input parameter(s)

The input parameters are checked for positive values. If a parameter is negative or zero, IERR is set to -1, and the function return value is set to zero. The message INVALID ARGUMENT PASSED TO KOMSTR will be given the first time an error is found. If the input parameters STRA and STRB have an address outside your program space, the end vector will be taken. This error may be the result of the input parameters being out of sequence. Also note that error checking is not performed for input parameters IPOSA, LEN and IPOSB exceeding variables boundaries.

Return values:

STRA > STRB, KOMSTR = +1
STRA = STRB, KOMSTR = 0
STRA < STRB, KOMSTR = -1

Example:

```
          DIMENSION B(3)
          DATA B/'123456TEST12'/
          A = 'TEST'
C
C    COMPARE STRINGS
C
          LEN = 4
          I = KOMSTR(A,1,LEN,B,7,IERR)
C
C    CHECK FOR ARGUMENT ERROR
C
          IF (IERR.LT.O) GO TO 100
C
C    MAKE USE OF ARITHMETIC IF
C
          IF (I) 30,40,50
             .
             .
             .
          STOP
C
C    ERROR HANDLING
C
     100
             .
             .
             .
          STOP
          END
```

This example will compare the first four characters of A with four characters of B starting with character 7. The return value will be zero, indicating that the strings are equal. Also, the error flag will be zero.

Example:

```
          DIMENSION NAME(4),INNAME(6)
          DATA NAME/'RAFFERTY'/
C
C    READ IN EMPLOYEE NAME AND SALARY TO DATE
C
     10 READ(5,50,END=998) (INNAME(I), I=1, 6), SALARY
     50 FORMAT (6A2,F10,2)
C
C    IS IT RAFFERTY?
C
          J = KOMSTR(NAME,1,8,INNAME,1,KERR)
          IF (KERR.EQ.-1) GO TO 999
          IF (J.EQ.0) GO TO 100
          GO TO 10
C
C    HANDLE THE RAFFERTY CASE
C
     100
              .
              .
              .
          STOP
C
C    HANDLE ERRORS
C
     998
              .
              .
     999      .
              .
          STOP
          END
```

This example will search input data for an employee with the name RAFFERTY.

D.4 INDEX FUNCTION

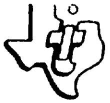
The function INDEX searches a portion of one string for an exact match to a portion of another string. When a match is found, the character number at which the matching substring begins is determined. This number is returned as the function value. If no match is found, 0 is returned.

The form of the call is:

```
M = INDEX(STRA,IPOSA,LENA,STRB,IPOSB,LENB)
```

The required parameters for this call are defined as follows:

- STRA – The variable or array containing the string to be searched.
- IPOSA – An INTEGER*2 variable or constant containing the character number within A at which the search begins.
- LENA – An INTEGER*2 variable or constant containing the length of the substring within A to be searched.



- STRB – The variable or array containing the string to be matched.
- IPOSB – An INTEGER*2 variable or constant containing the character number within B at which the substring to be matched begins.
- LENB – An INTEGER*2 variable or constant containing the length of the substring within B to be matched. If LENB>LENA, the value 0 will be returned.

The input parameters IPOSA, LENA, IPOSB and LENB are checked for positive values. Upon detection of a negative or zero value, the function returns a value of -1. The message INVALID ARGUMENT PASSED TO INDEX will be given the first time this occurs. It is the user's responsibility to check the parameters being passed against their upper bounds. Execution will continue with values that are out of range and may lead to unexpected results.

Example:

```
          INTEGER STRA(7),STRB(2)
          DATA STRA/'ABCDEFGHIJKLMN'/,STRB/'FGHI'/
C
C     SEARCH FOR A MATCH
C
          M = INDEX(STRA,3,10,STRB,2,3)
C
C     CHECK FOR ARGUMENT ERROR
C
          IF (M.EQ.-1) GO TO 100
                .
                .
                .
          STOP
100             .
                .  (Error handling)
                .
          STOP
          END
```

The above program will search 10 characters of STRA beginning at position 3 for a match to the three characters in STRB which begin at position 2. The substring GHI is matched, returning the value 7 to M.

Example:

```
          DIMENSION A(2)
          DATA A/'12345678'/,B/'456X'/
C
C     SEARCH FOR A MATCH
C
      M = INDEX(A,5,4,B,1,3)
C
C     CHECK FOR ARGUMENT ERROR
C
```

```
      IF (M.EQ.-1) GO TO 100
         •
         •
         •
      STOP
 100     •
         •    (Error handling)
         •
      STOP
      END
```

The above program will search four characters of A beginning at position 5 for a match to the first three characters in B. No match to the substring 456 is found. The value 0 is returned to M.

D.5 IVERFY FUNCTION

The IVERFY portion of the character string handling package is designed to compare a character string of data with a user-supplied character string composed of all permissible characters. Each character is compared with the list of valid (acceptable) characters. If an invalid character is located within the string, a pointer value is set that is the character number within the string found to be invalid. If no invalid characters are found, the pointer value will contain a zero.

The form of the call is:

```
M = IVERFY (STRA,IPOSA,LENA,STRB,IPOSB,LENB)
```

The required parameters for this call are defined as follows:

- STRA – Any variable or array containing the character string to be verified.
- IPOSA – An INTEGER*2 variable or constant representing the beginning character within the first string.
- LENA – An INTEGER*2 variable or constant representing the number of characters within the first string to be verified.
- STRB – Any variable or array containing the character string against which the first string will be checked.
- IPOSB – An INTEGER*2 variable or constant that represents the beginning character within the second string.
- LENB – An INTEGER*2 variable or constant representing the number of character within the second string against which the first string will be checked.

If an invalid argument (a negative or zero parameter) is passed for any of the parameters IPOSA, LENA, IPOSB or LENB, the IVERFY routine will output the error message INVALID ARGUMENT PASSED TO IVERFY. IVERFY will subsequently return a -1 to the calling program as the value of the function.

Example:

```
      DIMENSION ABLE(3), BAKER(3)
      DATA ABLE/' 12345.I7890'/
      DATA BAKER/' 0123456789.'/
```

```
C
C    VERIFY STRING
C
          M = IVERFY ( ABLE,1,10,BAKER,1,12 )
C
C    CHECK FOR ERROR RETURN
C
          IF ( M .EQ. -1 ) GO TO 900
C
C    CONTINUE WITH PROGRAM
C
                .
                .
                .
          STOP
C
C    ERROR HANDLING
C
     900
                .
                .
                .
          STOP
          END
```

In this example, the IVERFY routine would return an 8 for the value of the variable M, indicating that an invalid character was detected at the eighth location string ABLE. Note that a blank was considered a valid character.

Example:

```
          INTEGER*2 MIKE(4), VALID(6)
          DATA MIKE /'6020164 '/
          DATA VALID /' 0123456789.'/
C
C    VERIFY DATA
C
          M = (MIKE, 5, 4, VALID, 1, 12)
C
C    CHECK FOR INVALID PARAMETER
C
          IF ( M .EQ. -1) GO TO 999
C
C    CHECK FOR INVALID CHARACTER
C
          IF ( M .NE. 0 ) GO TO 500
C
C    NO ERRORS. PROCESS DATA.
C
          WRITE . . . .
                .
                .
```

```
C
C   ERROR HANDLING ROUTINES
C
    500       .
              .
    900       .
              .
        STOP
        END
```

In this example, the user wants to verify only the last four characters contained in the string MIKE. The IVERFY routine will return a 0 for the value of M, indicating that the characters verified without error.

D.6 MFLD FUNCTION

The integer function MFLD allows the user to select a bit, or a group of consecutive bits, from a 16-bit word and to store these bits right-adjusted into the value of the function.

The form of the call is:

```
I = MFLD(IPOS,NUM,WORD,IERR)
```

The required parameters for this call are defined as follows:

- IPOS – INTEGER*2 variable or constant. The starting bit position. This position is counted from zero. The right-most bit of the word will be the zero position. The left-most bit of the word is bit 15.

  The following example shows the bit positions within a word:

  (15––––––9876543210)
  To select the desired starting bit, count from right to left.
  Example: User wants bits 4 thru 7;
  J will equal 4 (not 7).

- NUM – INTEGER*2 variable or constant. Number of consecutive bits to be selected. One bit is the smallest number of bits that can be selected and 16 is the largest number.

- WORD – Any 16-bit variable or constant. The 16-bit word from which the bits will be selected.

- IERR – INTEGER*2 variable. A returned value of zero indicates an error-free frunction call. A minus one (-1) flags an input argument error.

The error flag (IERR) is set to -1 when any of the three following conditions is not met:

```
0 <= IPOS <= 15
1 <= NUM <= 16
IPOS + NUM <= 16
```

The error message INVALID ARGUMENT PASSED TO MFLD is given the first time such an error is found.

Example 1:

```
IWORD='AB'
MASK=MFLD(0,8,IWORD,IERR)
```

On return from the function, the right-most portion of the variable MASK will contain B and the remaining left portion will contain zeros.

Example 2:

```
IWORD='AB'
MASK=MFLD(8,8,IWORD,IERR)
```

On return from the function, the right-most portion of the variable MASK will contain A and the remaining left portion will contain zeros.

Example 3:

If the variable IWORD contained the hexadecimal value EF and the user wanted 3 bits starting with bit 4, the following would be true.

```
I=MFLD(4,3,IWORD,IERR)
```

On return from the function; I=6

D.7 LENGTH FUNCTION

The integer function LENGTH allows the user to find the length of a character string. The storage area of the character string is scanned from right to left until the first nonblank character is found. This is the last character in the string. The length of the character string is returned as the function value.

The form of the call is:

```
I = LENGTH(STRING,MAXLEN)
```

The required parameters for this call are defined as follows:

- STRING -- The character string may be any data type and can be a dimensioned variable.
- MAXLEN – The INTEGER*2 variable or constant is the maximum size of the character string.

The input parameters are checked for certain boundary conditions. If an input parameter address is less than or equal to zero or the value of MAXLEN is negative, the function return value is set to -1. The message INVALID ARGUMENT PASSED TO LENGTH will be given the first time an error is found. If the input parameters STRA and MAXLEN have an address outside your program space, the end vector will be taken. This error may be the result of the input parameters being reversed. Also note that error checking is not performed for input parameter MAXLEN exceeding variable boundaries.

Example:

```
INTEGER*2 INBUFF(10)
DATA INBUFF/' LENGTH TEST          '/
```

```
C
C     FIND THE LENGTH OF THE STRING
C
        LEN = LENGTH(INBUFF,20)
C
C     CHECK FOR PARAMETER ERROR
C
        IF (LEN.LT.0) GO TO 100
C
C     WRITE OUT THE LENGTH
C
        WRITE(6,50)LEN
  50 FORMAT(' THE STRING LENGTH = ',I2)
          .
          .
          .
        STOP
C
C     ERROR HANDLING
C
  100
          .
          .
          .
        STOP
        END
```

This example will give the length of the character string in INBUFF. The function return value is 13.

Example:

```
        REAL*4 BUFFER
        LB = 4
C
C     PROMPT FOR COMMAND FROM THE TERMINAL
C
  10 DISPLAY (5,50,LINE=5,POSITION=5,ERASE)
  50 FORMAT('ENTER COMMAND: ')
     ACCEPT (5,75,LINE=5,POSITION=21,PROMPT)BUFFER
  75 FORMAT (A4)
C
C     BRANCH TO HANDLE COMMANDS BASED ON THEIR LENGTH
C
        LENCMD = LENGTH (BUFFER,LB)
        GO TO (100,200,300,400),LENCMD
C
C     HANDLE COMMANDS
C
  100
          .
          .
          .
        STOP
```

```
200
      .
      .
      .
      GO TO 10
300
      .
      .
      .
      GO TO 10
400
      .
      .
      .
      GO TO 10
      END
```

This example is a command processor. The LENGTH function is used to break the commands down by length before processing them.

D.8 DNCASE SUBROUTINE

The DNCASE subroutine converts a character string to lowercase letters. The character string is scanned from left to right replacing uppercase letters with lowercase letters. All other symbols in the string remain unchanged.

The form of the call is:

```
CALL DNCASE(STRING,LENGTH,IERR)
```

The required parameters for this call are defined as follows:

- STRING – The character string to be converted may be any data type and can be a dimensioned variable.
- LENGTH – The INTEGER*2 variable or constant is the number of characters in the character string.
- IERR – The INTEGER*2 variable is the error code:
  - 0 – Normal completion
  - -1 – Invalid input parameter(s)

The input parameters are checked for certain boundary conditions. If the value of the input parameter LENGTH is negative, IERR is set to -1. The message 'INVALID ARGUMENT PASSED TO DNCASE' is given the first time an error is found. If the input parameters have an address outside the program space, the end vector is taken. This error may be the result of the input parameters being out of sequence. Also, note that error checking is not performed for input parameter LENGTH exceeding variable boundaries.

Example:

```
      INTEGER*2 MSG(10)
      DATA MSG/'MESSAGE NUMBER ONE. '/
C
C  FIND THE LENGTH OF THE STRING
C
      LEN = LENGTH(MSG,20)
C
C  CONVERT TO LOWER CASE LETTERS
C
      CALL DNCASE(MSG,LEN,NERR)
C
C  CHECK FOR PARAMETER ERROR
C
      IF (NERR.LT.0) GO TO 200
C
C  WRITE OUT THE CONVERTED STRING
C
      WRITE(6,100)MSG
  100 FORMAT(' THE MESSAGE IS ',10A2)
                .
                .
                .
      STOP
C
C  ERROR HANDLING
C
  200           .
                .
                .
      STOP
      END
```

This example finds the length of the character string in MSG and converts the character string to lowercase letters.

Example:

```
      INTEGER*2 BUFFER(40)
C
C  READ IN A LINE OF TEXT
C
   10 READ(5,50,END=100)BUFFER
   50 FORMAT(40A2)
C
C  CONVERT TEXT TO LOWER CASE
C
      CALL DNCASE(BUFFER,80,IERR)
C
C  WRITE TEXT TO TEMP FILE
C
      WRITE(4,50)BUFFER
```

```
      GO TO 10
C
C  READ TEMP FILE BACK IN AND CONVERT
C  THE FIRST LETTER OF EACH SENTENCE TO
C  UPPER CASE.
C
  100 ENDFILE 4
      REWIND 4
      REWIND 5
      FLAG = 1
  150 READ(4,50,END=999)BUFFER
C
C  FIND FIRST NON BLANK IN THE LINE
C
      ISTART = 1
      LEN = LENGTH(BUFFER,80)
      IF (LEN.EQ.0) GO TO 400
  160 J = INDEX(BUFFER,ISTART,1,' ',1,1)
      IF (J.EQ.0) GO TO 165
      ISTART = ISTART + 1
      GO TO 160
C
C  CAPITALIZE FIRST LETTER OF A LINE
C
  165 IF (FLAG.EQ.0) GO TO 250
      CALL UPCASE(BUFFER,ISTART,IERR)
      FLAG = 0
C
C  FIND END OF SENTENCE
C
  250 I = INDEX(BUFFER,ISTART,81-ISTART,'.  ',1,3)
      IF (I.EQ.0) GO TO 400
      I = I + 3
      IF (I.LE.LEN) GO TO 300
      FLAG = 1
      GO TO 400
C
C     CAPITALIZE FIRST LETTER OF A SENTENCE
C
  300 N = I / 2
      L = 2 - (I - N * 2)
      IF (L.EQ.1) N = N + 1
      CALL UPCASE(BUFFER(N),L,IERR)
      ISTART = I + 1
      IF (ISTART.GT.80) GO TO 400
      LEN = LEN - I
```

```
      GO TO 250
C
C  WRITE UPDATED FILE
C
  400 WRITE(5,50)BUFFER
      GO TO 150
  999 ENDFILE 5
      STOP
      END
```

The example first converts the uppercase text in an input file to lowercase text. Then the first character in each sentence of the text is converted to uppercase. Note that this example uses several character string functions.

D.9 UPCASE SUBROUTINE

The UPCASE subroutine converts a character string to uppercase letters. The character string is scanned from left to right replacing lowercase letters with uppercase letters. All other symbols in the string remain unchanged.

The form of the call is:

CALL UPCASE(STRING,LENGTH,IERR)

The required parameters for this call are defined as follows:

- STRING – The character string to be converted may be any data type and can be a dimensioned variable.
- LENGTH – The INTEGER*2 variable or constant is the number of characters in the character string.
- IERR – The INTEGER*2 variable is the error code:
  0 – Normal completion
  -1 – Invalid input parameter(s)

The input parameters are checked for certain boundary conditions. If the value of the input parameter LENGTH is negative, IERR is set to -1. The message 'INVALID ARGUMENT PASSED TO UPCASE' is given the first time an error is found. If the input parameters have an address outside the program space, the end vector is taken. This error may be the result of the input parameters being out of sequence. Also, note that error checking is not performed for input parameter LENGTH exceeding variable boundaries.

Example:

```
      INTEGER*2 MSG(10)
      DATA MSG/'message entry #1    '/
C
C  FIND THE LENGTH OF THE STRING
C
      LEN = LENGTH(MSG,20)
C
C  CONVERT TO UPPER CASE LETTERS
C
      CALL UPCASE(MSG,LEN,IERR)
```

```
C
C   CHECK FOR PARAMETER ERROR
C
        IF (IERR.EQ.-1) GO TO 100
C
C   WRITE OUT THE CONVERTED STRING
C
        WRITE(6,50)MSG
   50 FORMAT(' THE MESSAGE IS ',10A2)
              .
              .
              .
        STOP
C
C   ERROR HANDLING
C
  100         .
              .
              .
       STOP
       END
```

This example finds the length of the character string in MSG and converts the character string to uppercase letters.

Example:

```
        REAL*4 BUFF(20)
        LB = 80
C
C   READ IN A LINE OF TEXT
C
   10 READ(5,50,END=999)BUFF
   50 FORMAT(20A4)
C
C   CONVERT TEXT TO UPPER CASE
C
        CALL UPCASE(BUFF,LB,IERR)
C
C   WRITE TEXT TO TEMP FILE
C
        WRITE(4,50)BUFF
        GO TO 10
              .
              .
              .
  999 STOP
      END
```

This example converts the lowercase text in an input file to uppercase text and stores the results in another file.

D.10 TRANS SUBROUTINE

The TRANS subroutine translates a character string. The original character string is scanned from left to right comparing each character with the first translate string. If a match is found, the corresponding character in the second translate string is used to replace the character in the original character string.

The form of the call is:

CALL TRANS(STRING,LENGTH,TSTRA,TLEN,TSTRB,IERR)

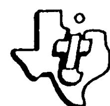
The required parameters for this call are defined as follows:

- STRING – The character string to be translated may be any data type and can be a dimensioned variable.
- LENGTH – The INTEGER*2 variable or constant is the number of characters in the character string being translated.
- TSTRA – The comparison translate character string may be any data type and can be a dimensioned variable.
- TLEN – The INTEGER*2 variable or constant is the number of characters in the translation character strings.
- TSTRB – The replacement translate character string may be any data type and can be a dimensioned variable.
- IERR – The INTEGER*2 variable is the error code:
  0 – Normal completion
  -1 – Invalid input parameter(s)

The input parameters are checked for certain boundary conditions. If the value of the input parameter LENGTH or TLEN is negative, IERR is set to -1. The message 'INVALID ARGUMENT PASSED TO TRANS' is given the first time an error is found. If the input parameters have an address outside the program space, the end vector is taken. This error may be the result of the input parameters being out of sequence. Also, note that error checking is not performed for input parameter LENGTH or TLEN exceeding variable boundaries.

Example:

```
          INTEGER*2 BUFFER(15),TA(2),TB(2)
          DATA BUFFER/'THE PROGRAM [LEVELA] IS ''OK''! '/
          DATA TA/'[]''!'/
          DATA TB/'()".'/
C
C         WRITE OUT INITAL BUFFER
C
          WRITE(6,50)BUFFER
       50 FORMAT('1THE INITIAL BUFFER IS:  ',15A2)
C
C         CHANGE [ TO (
C                ] TO )
C                ' TO "
C                ! TO .
```




```
C
      CALL TRANS(BUFFER,30,TA,4,TB,IERR)
C
C     WRITE OUT THE TRANSLATED BUFFER
C
      WRITE(6,100)BUFFER,IERR
  100 FORMAT('OTHE NEW BUFFER IS:  ',15A2,/,' IERR = ',I2)
      STOP
      END
```

This example translates the character string in BUFFER according to the translate strings TA and TB.

Example:

```
      DIMENSION TEXT(20),ASCII(64),EBCDIC(64),BUFFER(64)
      INTEGER*4 TEMP,VALUE
      INTEGER*4 NAME(34)
C
C     CONTROL NAMES
C
      DATA   NAME/'NUL ','SOH ','STX ','ETX ',
     +            'EOT ','ENQ ','ACK ','BEL ',
     +            'BS  ','HT  ','LF  ','VT  ',
     +            'FF  ','CR  ','SO  ','SI  ',
     +            'DLE ','DC1 ','DC2 ','DC3 ',
     +            'DC4 ','NAK ','SYN ','ETB ',
     +            'CAN ','EM  ','SUB ','ESC ',
     +            'FS  ','GS  ','RS  ','US  ',
     +            'SP  ','DEL '/
C
C     EBCDIC (INPUT) TRANSLATE TABLE
C     IBM STANDARD U.S. BIT PATTERN
C
      DATA EBCDIC/>00010203,>04050607,>08090A0B,>0C0D0E0F,
     1           >10111213,>14151617,>18191A1B,>1C1D1E1F,
     2           >20212223,>24252627,>28292A2B,>2C2D2E2F,
     3           >30313233,>34353637,>38393A3B,>3C3D3E3F,
     4           >40414243,>44454647,>48494A4B,>4C4D4E4F,
     5           >50515253,>54555657,>58595A5B,>5C5D5E5F,
     6           >60616263,>64656667,>68696A6B,>6C6D6E6F,
     7           >70717273,>74757677,>78797A7B,>7C7D7E7F,
     8           >80818283,>84858687,>88898A8B,>8C8D8E8F,
     9           >90919293,>94959697,>98999A9B,>9C9D9E9F,
     A           >A0A1A2A3,>A4A5A6A7,>A8A9AAAB,>ACADAEAF,
     B           >B0B1B2B3,>B4B5B6B7,>B8B9BABB,>BCBDBEBF,
     C           >C0C1C2C3,>C4C5C6C7,>C8C9CACB,>CCCDCECF,
     D           >D0D1D2D3,>D4D5D6D7,>D8D9DADB,>DCDDDEDF,
     E           >E0E1E2E3,>E4E5E6E7,>E8E9EAEB,>ECEDEEEF,
     F           >F0F1F2F3,>F4F5F6F7,>F8F9FAFB,>FCFDFEFF/
```

```
C
C     ASCII (OUTPUT) TRANSLATE TABLE
C     CODES NOT PRODUCED IN TRANSLATION ARE:
C     >13, >1D, >1F, >5B, >5D, AND >5E
C
      DATA  ASCII/>00010203,>0009007F,>0000000B,>0C0D0E0F,
     1              >10111200,>00000800,>18190000,>00000000,
     2              >00001C00,>000A171B,>00000000,>00050607,
     3              >00001600,>001E0004,>00000000,>1415001A,
     4              >20000000,>00000000,>0000002E,>3C282B00,
     5              >26000000,>00000000,>00002124,>2A293B00,
     6              >2D2F0000,>00000000,>00007C2C,>255F3E3F,
     7              >00000000,>00000000,>00603A23,>40273D22,
     8              >00616263,>64656667,>68690000,>00000000,
     9              >006A6B6C,>6D6E6F70,>71720000,>00000000,
     A              >007E7374,>75767778,>797A0000,>00000000,
     B              >00000000,>00000000,>00000000,>00000000,
     C              >7B414243,>44454647,>48490000,>00000000,
     D              >7D4A4B4C,>4D4E4F50,>51520000,>00000000,
     E              >5C005354,>55565758,>595A0000,>00000000,
     F              >30313233,>34353637,>38390000,>00000000/
C
C     CONVERT EBCDIC TO ASCII
C     UNTRANSLATABLE CHARACTERS ARE CONVERTED TO NULLS
C
      DO 10 I=1,64
   10 BUFFER(I) = EBCDIC(I)
      LEN = 256
      CALL TRANS(BUFFER,LEN,EBCDIC,256,ASCII,IERR)
C
C     CHECK FOR PARAMETER ERROR
C
      IF (IERR.EQ.-1) GO TO 500
C
C     WRITE OUT RESULTS IN THE FORM OF A TABLE
C
      K = 0
      TEMP = 0
      DO 300 I=1,64
      IF (MOD(K,32).GT.0) GO TO 50
      WRITE(6,25)
   25 FORMAT('1    EBCDIC CODE  ASCII CODE  ASCII VALUE')
      WRITE(6,35)
   35 FORMAT('+    -----------  ----------  -----------',/)
   50 DO 200 J=1,4
      CALL SUBSTR(BUFFER(I),J,1,TEMP,4,1,IERR)
      IF (IERR.EQ.-1) GO TO 550
      VALUE = TEMP
```

```
C
C       ADD CONTROL NAMES
C
        IF (TEMP.EQ.127) VALUE = NAME(34)
        IF (TEMP.GT.32) GO TO 75
        II = TEMP+1
        VALUE = NAME(II)
   75   WRITE(6,100)K,TEMP,VALUE
  100   FORMAT(5X,Z2,11X,Z2,10X,A4)
  200   K = K+1
        WRITE(6,150)
  150   FORMAT(5X,'----------------------------------------')
  300   CONTINUE
        STOP
C
C       ERROR HANDLING
C
  500   WRITE(6,600)
  600   FORMAT(' INVALID INPUT TO TRANS')
        STOP
  550   WRITE(6,650)
  650   FORMAT(' INVALID INPUT TO SUBSTR')
        STOP
        END
```

This example converts a buffer of EBCDIC codes to ASCII codes and produces a table of the translation results. The translate tables shown in this example could be used in an application to convert EBCDIC data to ASCII.

APPENDIX E

FORTRAN RUNTIME DESCRIPTION

E.1 INTRODUCTION TO FORTRAN RUNTIME

Many of the functions available to the FORTRAN user are provided in the form of callable subprograms that must be linked with the object code output of the compiler. These include not only the user explicitly called functions (such as the intrinsic functions), but also many that the user never explicitly calls, but are automatically called by compiler generated calls. All of these routines are included in the FORTRAN runtime directories.

For DX10 3.X releases there are five directories (OSLOBJ, STLOBJ, DXLOBJ, TXLOBJ and SALOBJ) available for linking with compiler generated object code. For TXDS releases there are 2 directories (TXLOBJ and SALOBJ) available. Linking with the different directories provides linked object modules suitable for running in different operating environments. Linked object modules can be generated for DX10 3.X, TX990, and standalone (no operating system).

For DX10 3.X environments, standard FORTRAN runtime routines are contained in two directories. The first directory (OSLOBJ) contains operating system specific routines. OSLOBJ cannot be used for standalone FORTRAN. All other FORTRAN routines are contained in the second standard directory (STLOBJ). Linking with a special directory (DXLOBJ) for DX10 applications and standard directories (OSLOBJ and STLOBJ) produces linked object modules to run under DX10, using global LUNOs for Input/Output. Linking with SALOBJ for standalone applications and one of the standard directories (STLOBJ) produces linked object modules that must require no operating system support for execution. Linking with TXLOBJ and the standard directories (OSLOBJ and STLOBJ) produce linked object modules to run under TX990.

For TXDS development environments, all FORTRAN runtime routines are contained in one directory TXLOBJ. Linking with the additional directory SALOBJ for standalone applications produces linked object modules that must not require operating system support for execution.

E.2 OPERATING CHARACTERISTICS

The following paragraphs briefly describe the methods of I/O unit correspondence, message and error logging and program termination provided by each directory.

E.2.1 THE DX10 3.X STANDARD DIRECTORIES. The standard FORTRAN Runtime is made up of two directories. One directory provides the support functions that are operating system specific, such as Input/Output functions (READ, WRITE, ACCEPT, DISPLAY, REWIND, ENDFILE, etc.). The second directory consists of nonoperating system specific functions and modules that are independent of the operating system environment (for example, the real arithmetic support).

I/O Unit Correspondence is provided through synonym assignment of the DX10 System Command Interpreter. Messages and errors are logged on a terminal local file. The terminal local file is automatically output at the end of execution.

E.2.2 DXLOBJ DIRECTORY. The special directory for programs that are linked for execution under DX10 using global LUNOs contains only those routines that are environment dependent. All other routines are obtained from the standard directories. I/O unit correspondence is provided by the assignment of a global or task local LUNO to the device or file used in the FORTRAN program. Messages and errors are logged on the system log.

E.2.3 TXLOBJ DIRECTORY. The TXLOBJ directory on DX10 contains those modules required by TXDS that differ from DX10. The directory on TXDS contains all FORTRAN runtime routines. I/O unit correspondence is provided by the assignment of a GLOBAL LUNO through Operator Communications Package (OCP) to the device or file used in the FORTRAN program. Messages and errors are logged on the system console.

E.2.4 THE STANDALONE DIRECTORY. The special standalone directory contains only those environment dependent routines which are required in every link edit of a FORTRAN program. Other routines are obtained from the standard directory containing nonoperating system dependent support functions, STLOBJ for DX10 and TXLOBJ for TXDS. Functions requiring operating system support (refer to table E-1) may not be used. Use of an operating system supported function requires that the support routines be supplied by the user.

Messages and errors are logged by leaving an error code in word 0 of F$XLWS. An error code of 0 implies normal termination. A nonzero error code implies an error has occurred. The code is the address of a message buffer that contains a message describing the error. The first word in the buffer specifies the number of ASCII characters in the message and is followed by the ASCII characters of the message.

Table E-1. Functions Which Are O.S. Dependent

All I/O except CRU.	READ, WRITE, ACCEPT, DISPLAY, BACKSPACE, DEFINE FILE,ENCODE,DECODE
ISA File Contention Routines	RDRW, WRTRW, CFILW, DFILW, OPENW, CLOSEW, SVCFUT, MODAPW
ISA Extensions Making O.S. Supervisor Calls	SVC, DATIME, WAIT, TRNON, START, TIME, BIDTSK, DLYBID, ATIME, ADATE, DATE, MDATE
Initialization	
Error Logging	
Termination	

Note: In DX10 3.X these functions are contained in the standard directory OSLOBJ.

E.3 REPLACING STANDARD MODULES

The user can modify certain characteristics of the FORTRAN runtime package by supplying to the link editor customized modules that replace the standard modules. The characteristics that can be altered are:

- Table space allocated for file control blocks and physical record blocks that determines the number of I/O units that may be used.
- Space allocated for the I/O buffer that determines the maximum length of records that may be read or written
- Termination processing
- End vector processing

- Error logging procedure
- Memory management for reentrant subprograms.

The following is documentation for the modules that may be replaced to alter the standard operational characteristics of the FORTRAN runtime directories.

E.3.1 NUMBER OF I/O UNITS ALLOWED. The modules F$XFCB and F$XTBL control the maximum number of units that may be used in a FORTRAN program. If one is altered the other should be checked to ensure that compatibility is maintained.

F$XFCB is the file control file block built at runtime. This table should have one entry for each Logical Unit Number (LUNO) that the FORTRAN program uses for I/O. The default table size allows I/O to 20 LUNOs. The number of LUNOs allowed may be altered by the user by changing the value of MAXLUN in this module. MAXLUN is equated to the maximum number of LUNOs allowed.

NOTE

> If the maximum number of LUNOs allowed is increased, F$XTBL should be examined to ensure that there is enough Physical Record Block (PRB) table space available for the extra LUNOs.

Figure E-1. is a listing of F$XFCB.

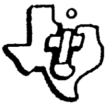
F$XTBL contains the table space for the physical record blocks (PRB). For each I/O unit there must be space for a PRB. The PRB area is defined by two labels. F$XTBL defines the start of the PRB area and F$XTBE defines the end of the PRB area.

For the F$XTBL module for use on DX10, the size of the PRB area is specified by the label TBLSIZ. This area may be increased by changing the value of TBLSIZ but it must not be less than 360_{16} because the area is initially used to read in the TCA record. The space for each PRB is 38 bytes. Therefore, for a maximum of 20 LUNOs, 760 ($2F8_{16}$) bytes must be available. Since this is less than 360_{16}, TBLSIZ must be set to 360_{16}.

Figure E-2 is a listing of F$XTBL for use on DX10.

For the F$XTBL module used on TX990/TXDS, the size of the PRB area is specified by the label MAXLUN. This label specifies the maximum number of I/O units. By increasing this value, the PRB area is increased. The space for each TX990/TXDS PRB is 26 bytes. Therefore, for a maximum of 25 LUNOs, MAXLUN would be set to 25 to reserve 650 ($28A_{16}$) bytes for the total PRB space.

Figure E-3 is a listing of F$XTBL for use on TX990/TXDS.




```
        IDT   'F$XFCB'
*
MAXLUN EQU   20                    ALLOW I/O TO 20 LUNS
*
*
*       DEFINITION OF AN FCB
*
*
        DORG 0
LUN     BSS   1                    LUN NUMBER FOR THIS FCB ENTRY
LSTATS BSS   1                     LAST STATUS ON FILE
PRB     BSS   2                    ADDR OF PRB TO ACCESS LUN
STATE   BSS   1                    STATE OF LUN AFTER LAST I/O
FLAGS   BSS   1                    MISC FLAG BYTE
ASSCAD BSS   2                     ASSOCIATED VARIABLED ADDRESS
FINAME BSS   2                     ADDRESS OF FILE NAME
        EVEN
ENDFCB EQU   $                     LENGTH OF FCB STRUCTURE


*
*
*
        DEF   F$XFCB,F$XFCE
*
        RORG
        DSEG
F$XFCB EQU   $                     START OF FCB TABLE
        BSS   MAXLUN*ENDFCB
F$XFCE EQU   $                     END OF FCB TABLE + 1
        DEND
        END
```

Figure E-1. F$XFCB

```
        IDT   'F$XTBL'
*
        DEF   F$XTBL,F$XTBE
        DEF   F$RPRM,F$RASN,F$RPRB,F$ROP,F$RNUM
        DEF   F$RTCA,F$RTID,F$RCAL,F$RMON
        DEF   F$XTID,F$XCAL,F$XMON
*
        DSEG
TBLSIZ  EQU   >360

        EVEN
F$XTBL  EQU   $                 START OF TCA RECORD BUFFER
        BSS   TBLSIZ
F$XTBE  EQU   $                 END OF TCA RECORD BUFFER + 1.
*
        PAGE
*  SVC CALL BLOCKS USED BY F$RGET TO READ TCA RECORD
        EVEN
F$XPRB  BYTE  0                 I/O CALL
        BYTE  0                 I/O SVC ERROR CODE
F$XOP   BYTE  0                 INITIALLY 'OPEN' OP
        BYTE  >F                LUNO OF TCA FILE
        BYTE  0                 SYSTEM FLAGS
        BYTE  0                 USER FLAGS (NO SHARE)
F$TCA   DATA  0                 ADDR OF TCA BUFFER
        DATA  >360              LENGTH OF TCA (CHARACTERS)
        DATA  0                 CHARACTER COUNT
        DATA  0                 NOT USED HERE
        BYTE  0                 NOT USED HERE
F$XNUM  BYTE  0                 REC NUMBER (ID)
*
        EVEN
F$XPRM  BYTE  >17,0
AB2     BYTE  0                 DUMMY
F$XTID  BYTE  0-0               CONTAINS TERMINAL ID
AB4     BYTE  0                 DUMMY
F$XCAL  BYTE  0-0               CONTAINS CALLED TASK ID
*
*
        EVEN
F$XASN  BYTE  >15               ASSIGN TCA FILE
IGNERR  BYTE  0
        BYTE  1,2               OP CODE=ASSIGN,FLAGS=RR
        BYTE  >F,0              LUNO 15
        DATA  >360              RECORD LENGTH



        DATA  'TC'
        DATA  'AF'
        DATA  'IL'
        DATA  'DS'
        DATA  'C '
        DATA  0                 NUMBER OF RECORDS
        DATA  0                 NUMBER OF RECORDS
*
        EVEN
F$XMON  BYTE  0-0               MONITOR ID FOR THIS TERM.
*
        PAGE
F$RPRM  EQU   F$XPRM-F$XTBL
F$RASN  EQU   F$XASN-F$XTBL
F$RPRB  EQU   F$XPRB-F$XTBL
F$ROP   EQU   F$XOP-F$XTBL
F$RNUM  EQU   F$XNUM-F$XTBL
F$RTCA  EQU   F$TCA-F$XTBL
F$RTID  EQU   F$XTID-F$XTBL
F$RCAL  EQU   F$XCAL-F$XTBL
F$RMON  EQU   F$XMON-F$XTBL
        DEND
*
        END
```

Figure E-2. F$XTBL Used on DX10

```
         IDT    'F$XTBLTX'
*
         DEF    F$XTBL,F$XTBE
*
         DSEG
MAXLUN   EQU    20                     MAXIMUM NUMBER OF LUNOS
         EVEN
F$XTBL   EQU    $                      START OF LUN RECORD BUFFER
         BSS    MAXLUN*24
F$XTBE   EQU    $                      END OF LUN RECORD BUFFER + 1.
*
         DEND
         END
```

Figure E-3. F$XTBL Used on TXDS

E.3.2 I/O BUFFER SIZE. The size of the I/O buffer is controlled in module F$RBUF. F$RBUF is a data module that contains a buffer for the format editor. The standard length of the buffer is ~~144~~ 148 characters; four of these are reserved for carriage control. No formatted I/O may be performed with record sizes greater than the length of this buffer.

The user may change the length of the I/O buffer by changing the value of F$XBFS in this module. F$XBFS must be equated to the desired character length.

F$XBFS is referenced in F$XVFB. If F$XVFB is linked into a reentrant procedure, then all tasks using that procedure must be linked with the same version of F$RBUF.

The following is a listing of the existing F$RBUF module:

```
         IDT      'F$RBUF'
*
         DEF      F$RBUF, F$XBFS
*
F$XBFS   EQU      ~~138~~ 144       TOTAL BUFFER LENGTH
         DSEG
         BSS      4                 CARRIAGE CONTROL CHARS
F$RBUF   BSS      F$XBFS            I/O BUFFER
         DEND
*
         END
```

E.3.3 TERMINATION PROCESSING. A FORTRAN program always terminates in module F$XFTL.

F$XFTL terminates the user task and returns to the operating system. All termination modes including normal, detected error, and end vector terminations are performed here.

Special termination processing can be supplied by replacing F$XFTL with a user written module that has F$XFTL as the entry point name.

NOTE

No processing that depends upon using a particular workspace should be performed. Also, F$XFTL is a reentrant subroutine.

E.3.4 END VECTOR PROCESSING. If an error in a FORTRAN program occurs that causes the end vector to be taken, control is transferred to module F$REVP.

F$REVP contains the end vector routine, which should never be reached. It logs a message and then branches to F$XFTL, the task termination routine.

Special end vector processing can be supplied by replacing F$REVP with a user written module that has F$REVP as the entry point name.

NOTE

No processing should be performed that depends upon using a particular workspace.

E.3.5 MESSAGE AND ERROR LOGGING. All messages and errors are logged by the module F$XLOG.

The error logger F$XLOG, writes a message to the terminal local file (DX10) or the system console (TXDS). The type of message is encoded into the message as follows:

```
<MESSAGE NAME> BYTE<CODE>
               BYTE<LENGTH>
               TEXT'<MESSAGE>'
```

where:

<CODE> = 0 If not additional information required

<CODE> = 1 If an error code is present. If specified the message code = XXXX is added to the basic message.

<CODE> = 2 If a unit number is present. If specified the message unit = XXXX is added to the basic message.

(CODE) = 3 Error code and unit number

<LENGTH> = Length of basic message in bytes

<MESSAGE> = Basic error message

Calling Sequence:

```
LI    RO, <MESSAGE NAME>
LI    R1, <ERROR CODE>      OPTIONAL
LI    R2, <UNIT NUMBER>     OPTIONAL
~~BL    @F$XLOG~~
BLWP @F$RLOG(R10)           R10 = F$RWRK
or  BLWP @F$ILOG
```

NOTE

This routine is reentrant. Its workspace, temporary variables, and buffer are in F$XLWS. It is called by a BLWP through a branch vector in F$RWRK.

E.3.6 REENTRANT SUBPROGRAM MEMORY MANAGEMENT. The FORTRAN runtime library contains two routines that are used to coordinate the calling of recursive subprograms. The first of these is F$XREC which is used to call a recursive subprogram. F$XREC allocates a portion of the reentrant data area stack to be used for the current invocation of a recursive subprogram. When the subprogram terminates, it calls F$XRER which releases the data area for that invocation of the subprogram. A listing of the F$XREC routine and an example of it follow.

```
         TITL 'REENTRANT SUBPROGRAM LINKAGE'
*
*  THE FOLLOWING ROUTINE PERFORMS THE NECESSARY STEPS
*  TO CALL RECURSIVE SUBPROGRAMS, AND ALLOCATE THEIR DATA
*  AREA. THE USER NEEDS TO SUPPLY A MODULE CONTAINING
*  THE NECESSARY DATA STORAGE AREA FOR ALL HIS RECURSIVE
*  FORTRAN SUBPROGRAMS. THIS MODULE SHOULD BE INCLUDED IN
*  THE TASK SEGMENT OF THE PROGRAM LINK EDIT. THE STORAGE
*  AREA MODULE SHOULD HAVE THE FOLLOWING FORM:
*
*         DEF  F$STOR,F$STND
*         DSEG
* F$STOR  BSS  X     ( WHERE X IS THE NUMBER OF BYTES NEEDED )
* F$STND  BSS  32
*         END
*
*  THE VALUE OF X MUST BE LARGE ENOUGH TO HOLD THE DATA AREA
*  STORAGE FOR THE LONGEST LEVEL OF RECURSIVE DESCENT WHICH MAY
*  OCCUR IN THE TASK. EACH INVOCATION OF A RECURSIVE SUBPROGRAM
*  REQUIRES A NEW COPY OF ITS DATA AREA, SO IF A SUBPROGRAM WITH
*  A DATA AREA SIZE OF >50 BYTES DESCENDS THROUGH FOUR LEVELS OF
*  RECURSION, IT REQUIRES 4 * >50 = >140 BYTES OF DATA STORAGE.
*  THE 32 BYTES RESERVED AT LABEL F$STND ARE NOT COUNTED IN THE
*  ABOVE COMPUTATION.
*
*
*
*
        IDT   'F$XREC'
        DEF   F$XREC,F$XRER
        REF   F$STOR,F$STND      REFERENCE DATA AREA
        REF   F$ERRS,F$RWRK      REFERENCE ERROR ROUTINE,WORKSPACE
*
F$XREC  EQU   $
        BLWP  @FXREC             SIMULATE BLWP
        DSEG
FSTEND  DATA  F$STND             ADDRESS OF END OF DATA AREA
FXREC   DATA  F$STOR             ADDRESS OF DATA AREA
        DATA  ENTRY
        DEND
ENTRY   MOV   @22(13),R14        REPLACE PC POINTER
        MOV   *R14+,R0           GET SUBPROGRAM POINTER
        A     *R0+,@FXREC        ADD DATA AREA LENGTH
        C     @FXREC,@FSTEND     CHECK FOR OVERFLOW
        JH    OVFL
        STWP  R10                SET R10 TO WORKSPACE
        MOV   *R0,R0             GET SUBPROGRAM START ADDRESS
        B     *R0                BRANCH TO SUBPROGRAM
*
OVFL    LI    R0,RNTMSG          SET OVERFLOW MESSAGE
        LI    R1,OVLMSG
        LI    R10,F$RWRK         SET I/O WORKSPACE ADDRESS
        B     F$ERRS
*
F$XRER  MOV   R10,@FXREC         DEALLOCATE DATA AREA
        RTWP                     RETURN FROM SUBPROGRAM

RNTMSG  BYTE  9
        TEXT  'REENTRANT'
OVLMSG  BYTE  18
        TEXT  'DATA AREA OVERFLOW'
        END
```

Example:

The following example shows the user how to code a recursive FORTRAN function to compute the factorial of an integer. The factorial function can be defined recursively as:

```
FACTORIAL(0) = 1
FACTORIAL(N) = N * FACTORIAL(N-1)
```

A FORTRAN implementation of this function might look like:

```
      INTEGER FUNCTION FACTORIAL(N)
      IF (N .GT. 1) GOTO 1
      FACTORIAL = 1
      RETURN
1     FACTORIAL = N * FACTORIAL(N-1)
      END
```

The above function would have to be compiled using the FORTRAN 'R' option. The user would then have to create an assembly language module defining the stack to be used to store the local data for the function. The size of the stack must be large enough to hold a separate copy of the function's data area for each level of recursion that may occur. The above function has a data area size of >36 bytes (this information is printed with the error summary at the end of the compilation listing). The largest factorial which fits into an INTEGER*2 variable is the factorial of seven. Calling the above function with an argument of seven results in exactly seven levels of recursion. Therefore, the amount of storage needed is 7 * >36 = 378 bytes of storage. The following assembly language module allocates the necessary stack storage:

```
        IDT     'F$STOR'
        DEF     F$STOR,F$STND
        DSEG
F$STOR  BSS     378
F$STND  BSS     32
        END
```

If the function tried to recurse beyond seven levels, the FORTRAN runtime would issue the following message: REENTRANT, DATA AREA OVERFLOW ERROR followed by traceback information listing all the recursive calls leading to the error.

E.3.7 LINKING WITH USER-SUPPLIED MODULES ON DX10. The modules that the user supplies should be maintained in a directory to link with a FORTRAN Program. The modules are included by a library search rather than by an explicit INCLUDE command during the link edit procedure. Thus, both the specified and standard modules cannot be included in the link together.

NOTE

The special user directory must have aliases added for each entry point if the member names are different from the entry point names.

E.3.8 LINKING WITH USER SUPPLIED MODULES ON TXDS. The modules that the user supplies are included by an explicit INCLUDE command during the link edit procedure.

E.4 I/O INTERFACE

FORTRAN programs that run in the standalone mode cannot reference any runtime routines that depend on the services of an operating system. In DX10 3.X the dependent routines are included in a separate FORTRAN runtime directory. Standalone FORTRAN does not support FORTRAN language statements that must make a supervisor call during processing. The FORTRAN statements can be applied with user written I/O support routines. The following sections document the software interface between the compiler generated object code and the runtime I/O support routines.

E.4.1 READ/WRITE CALLING SEQUENCE. The READ and WRITE statements have the following general calling sequence:

```
BLWP   @<I/O set up>                 ENTRY POINT FOR A GIVEN TYPE OF I/O.
DATA   <arg 1>                       ADDRESS OF THE ENTRY POINT ARGU-
                                     MENTS
  .
  .
  .
DATA   <arg n>
BLWP   @<list element 1 handler>     HANDLES EACH ELEMENT IN THE I/O
                                     STATEMENT LIST.
DATA   <list element 1>
  .
  .
  .
BLWP   @<list element n handler>     ENTRY POINT NAME
DATA   <list element n>              ADDRESS OF THE LIST ELEMENT
BLWP   @<finish I/O>                 TERMINATES I/O OPERATION
```

The name of the entry point depends upon whether formatted or unformatted I/O is being performed. This routine has no arguments. All arguments are passed by address except those specially noted.

E.4.1.1 Formatted READ/WRITE. The calling sequence for each variation of formatted READ/WRITE statements follows:

```
READ (unit, format) list
    BLWP     @F$RRF
    DATA     unit
    DATA     format
      .
      .
      .
    (calls to list handlers)
      .
      .
      .
    BLWP     @F$RSIO
```

```
READ (unit, format, END = label 1, ERR = label 2) list
     BLWP      @F$RRFB
     DATA      unit
     DATA      format
     DATA      label 1
     DATA      label 2
       .
       .
       .
     (calls to list handlers)
       .
       .
       .
     BLWP      @F$RSIO

READ (unit, format, END = label 1) list
     BLWP      @F$RRFD
     DATA      unit
     DATA      format
     DATA      label 1
       .
       .
       .
     (calls to list handlers)
       .
       .
       .
     BLWP      @FRSIO

READ (unit, format ERR = label 1) list
     BLWP      @F$RRFR
     DATA      unit
     DATA      format
     DATA      label 1
       .
       .
       .
     (calls to list handlers)
       .
       .
       .
     BLWP      @F$RSIO

WRITE (unit, format) list
     BLWP      @F$RWF
     DATA      unit
     DATA      format
       .
       .
       .
     (calls to list handlers)
       .
       .
       .
     BLWP      @R$RSIO
```

```
WRITE (unit, format, ERR = label 1) list
      BLWP      @F$RWFR
      DATA      unit
      DATA      format
      DATA      label 1
        .
        .
        .
      (calls to list handlers)
        .
        .
        .
      BLWP      @F$RSIO
```

E.4.1.2 Unformatted READ/WRITE. The calling sequence for each variation of unformatted READ/WRITE statements follows:

```
READ (unit) list
      BLWP      @F$RRU
      DATA      unit
        .
        .
        .
      (calls to list handlers)
        .
        .
        .
      BLWP      @F$RSIP

READ (unit, END = label 1, ERR = label 2) list
      BLWP      @FRRUB
      DATA      unit
      DATA      label 1
      DATA      label 2
        .
        .
        .
      (calls to list handlers)
        .
        .
        .
      BLWP      @F$RSIP

READ (unit, END = label 1) list
      BLWP      @F$RRUD
      DATA      unit
      DATA      label 1
```

```
    (calls to list handlers)
        .
        .
        .
    BLWP      F$RSIP

READ (unit, ERR = label 1) list
    BLWP      @F$RRUR
    DATA      unit
    DATA      label 1
        .
        .
        .
    (calls to list handlers)
        .
        .
        .
    BLWP      @F$RSIP

WRITE (unit) list
    BLWP      @F$RWU
    DATA      unit
        .
        .
        .
    (calls to list handlers)
        .
        .
        .
    BLWP      @F$RSIP

WRITE (unit, ERR = label 1) list
    BLWP      @F$RWUR
    DATA      unit
    DATA      label 1
        .
        .
        .
    (calls to list handlers)
        .
        .
        .
    BLWP      @F$RSIP
```

E.4.1.3 Formatted Direct Access READ/WRITE. The calling sequence for each variation of formatted direct access READ/WRITE statements follows:

```
READ (unit'record, format) list
    BLWP      @F$RRE
    DATA      unit
    DATA      format
    DATA      record
        .
        .
        .
```

```
        (calls to list handlers)
              .
              .
              .
        BLWP       @F$RSIO

READ (unit'record, format END = label 1, ERR = label 2) list
        BLWP       @FR$RREB
        DATA       unit
        DATA       format
        DATA       record
        DATA       label 1
        DATA       label 2
              .
              .
              .
        (calls to list handlers)
              .
              .
              .
        BLWP       @R$RSIO

READ (unit'record, format, END = label 1) list
        BLWP       @F$RRED
        DATA       unit
        DATA       format
        DATA       record
        DATA       label 1
              .
              .
              .
        (calls to list handlers)
              .
              .
              .
        BLWP       @F$RSIO

READ (unit'record, format, ERR = label 1) list
        BLWP       @F$RRER
        DATA       unit
        DATA       format
        DATA       record
        DATA       label 1
              .
              .
              .
        (calls to list handlers)
              .
              .
              .
        BLLOP      @F$RSIO
```

```
WRITE (unit'record, format) list
      BLWP    @F$RWE
      DATA    unit
      DATA    format
      DATA    record
       .
       .
       .
      (calls to data handlers)
       .
       .
       .
      BLWP    @F$RSIO

WRITE (unit'record, format ERR = label 1) list
      BLWP    @F$RWER
      DATA    unit
      DATA    format
      DATA    record
      DATA    label 1
       .
       .
       .
      (calls to list handlers)
       .
       .
       .
      BLWP    @F$RSIO
```

E.4.1.4 Unformatted Direct Access READ/WRITE. The calling sequence for each variation of unformatted direct access READ/WRITE statements follows:

```
READ (unit'record) list
      BLWP    @F$RRL
      DATA    unit
      DATA    record
       .
       .
       .
      (calls to list handlers)
       .
       .
       .
      BLWP    @F$RSIP

READ (unit'record, END = label 1, ERR = label 2) list
      BLWP    @F$RRLB
      DATA    unit
      DATA    record
      DATA    label 1
      DATA    label 2
       .
       .
       .
```

```
     (calls to list handlers)
       .
       .
       .
     BLWP      @F$RSIP

READ (unit'record, END = label 1) list
     BLWP      @F$RRLD
     DATA      unit
     DATA      record
     DATA      label 1
       .
       .
       .
     (calls to list handlers)
       .
       .
       .
     BLWP      @F$RSIP

READ (unit'format, ERR = label 1) list
     BLWP      @F$RRLR
     DATA      unit
     DATA      record
     DATA      label 1
       .
       .
       .
     (calls to list handlers)
       .
       .
       .
     BLWP      @F$RSIP

WRITE (unit'record) list
     BLWP      @F$RWL
     DATA      unit
     DATA      record
       .
       .
       .
     (calls to list handlers)
       .
       .
       .
     BLWP      @F$RSIP

WRITE (unit'record, ERR = label 1) list
     BLWP      @F$RWLR
     DATA      unit
     DATA      record
     DATA      label 1
       .
       .
       .
```

```
(calls to list handlers)
      .
      .
      .
BLWP    @F$RSIP
```

E.4.1.5 I/O List Element Handlers. Each element in the I/O list of a READ or WRITE statement generates a call. The entry point depends upon the elements. If the element is an unsubscripted array name, the general format of the call follows:

```
BLWP    @<entry point>
DATA    <element name>
DATA    <array size name>  (only present for fixed array)
```

The following entry point names apply:

F$RIUS – integer array, formatted I/O
F$RIUT – integer array, unformated I/O
F$REUS – extended integer array, formatted I/O
F$REUT – extended integer array, unformatted I/O
F$RFUS – fixed array, formatted I/O
F$RFUT – fixed array, unformatted I/O
F$RRUS – real array, formatted I/O
F$RRUT – real array, unformatted I/O
F$RDUS – double precision array, formatted I/O
R$RDIT – double precision array, unformatted I/O
F$RCUS – complex array, formatted I/O
F$RCUT – complex array, unformatted I/O
F$RLUS – logical array, formatted I/O
F$RLUT – logical array, unformatted I/O

The first argument following the BLWP instruction is the address of the list element. The second argument is the address of the location containing the size of the array. The third argument is the scale factor of a fixed array and is present only when the entry point is F$RFUS or F$RFUT. The third argument is not an address.

If the element is a scalar variable, constant, or array element, the general format of the call follows:

```
BLWP    @<entry point>
DATA    <element name>
DATA    <scale factor>  (present only for fixed type)
```

The following entry point names apply:

F$RIOL – integer, formatted I/O
F$RIOM – integer, unformatted I/O
F$REOL – extended integer, formatted I/O
F$REOM – extended integer, unformatted I/O
F$RFOL – fixed, formatted I/O
F$RFOM – fixed, unformatted I/O
F$RROL – real, formatted I/O
F$RROM – real, unformatted I/O
F$RDOL – double precision, formatted I/O
F$RDOM – double precision, unformatted I/O

F$RCOL – complex, formatted I/O
F$RCOM – complex, unformatted I/O
F$RLOL – logical, formatted I/O
F$RLOM – logical, unformatted I/O

The first word following the BLWP instruction contains the address of the list element. The second word contains the scale factor, but is only present when the entry point is F$RFOL and F$RFOM. The second word is not an address.

E.4.2 DIRECT ACCESS FILE DEFINITION. The calling sequence for defining direct access files and for locating a record is as follows:

```
DEFINE FILE unit (number of records, rec size, control, associated variable)
    BLWP   @F$RDEF
    DATA   unit                  Location address of unit number
    DATA   number of records     Location address of number of records
    DATA   rec size              Location address of rec size
    DATA   control               Location address of the code for the type
                                 of format control*
    DATA   associated variable   Address of the associated variable
```

*The codes are:

0 = format control
1 = no format control
2 - either format or no format control

```
FIND (unit'record)
    BLWP   @F$RFND
    DATA   unit
    DATA   record                Location address of the unit number
```

E.4.3 BUFIN/BUFOUT SUBROUTINES. The subroutines for transferring physical records between I/O devices and memory buffers are called as follows:

```
CALL BUFIN (unit, mode, buffer, charcount)
    BLWP   @FRFIN
    DATA   4
    DATA   unit
    DATA   mode
    DATA   buffer
    DATA   charcount

CALL BUFOUT (unit, mode, buffer, charcount)
    BLWP   @BUFOUT
    DATA   4                     Number of arguments in the source
                                 statements
    DATA   unit                  Location address of the unit number
    DATA   mode                  Location address for the mode of operation
                                 (0 = ASCII, 1 = binary)
    DATA   buffer                Location address for memory buffer area
    DATA   charcount             Location address of the number of characters
                                 to be transferred
```

E.4.4 ACCEPT/DISPLAY. The calling sequences for transferring data to and from the screen of cathode ray tube I/O devices (CRTs) are as follows:

```
ACCEPT (unit, format, line=label 1, position=label 2, erase, prompt, echo, ERR=label3) list
        BLWP    @F$RACC
        DATA    unit
        DATA    format
        DATA    label 1
        DATA    label 2
        DATA    flag (|   |EC |PR |ER|   |)
          .
          .
          .
        (calls to list handles)
          .
          .
          .
        BLWP    @F$RSIO

DISPLAY (unit, format, line-label 1, position=label 2, ERR=label 3) list
        BLWP    @F$RDIS
        DATA    unit
        DATA    format
        DATA    label 1
        DATA    label 2
        DATA    Erase flag (0=no, 1=erase)
        DATA    label 3
          .
          .
          .
        (calls to list handlers)
          .
          .
          .
        BLWP    @F$RSIO
```

E.4.5 MASS STORAGE FILE INPUT/OUTPUT STATEMENTS. The calling sequences for rewinding a file, backspacing a record on a file, and placing an end-of-file on a file follow:

```
REWIND unit
        BLWP    @F$RREW
        DATA    unit

BACKSPACE unit
        BLWP    @F$RBSP
        DATA    unit

ENDFILE unit
        BLWP    @F$REND
        DATA    unit
```

E.4.6 ENCODE/DECODE. The calling sequences for transferring data from one area of computer memory to another are as follows:

```
ENCODE (number of characters per record, format, buffer, number of characters processed) list
        BLWP    @F$RENN
        DATA    number of characters per record
        DATA    number of characters processed
        DATA    format
        DATA    buffer
          •
          •
          •
        (calls to list handlers)
          •
          •
          •
        BLWP    @F$RSIO

DECODE (number of characters per record, format, buffer, number of characters processed) list
        BLWP    @F$RDEN
        DATA    number of characters per record
        DATA    number of characters processed
        DATA    format
        DATA    buffer
          •
          •
          •
        (calls to list handlers)
          •
          •
          •
        BLWP    @F$RSIO
```

APPENDIX F

FORTRAN INSTALLATION ON TXDS

F.1 COMPILER

The FORTRAN compiler, TXFTN/SYS, is supplied installed on diskette. To compile under TXDS, the diskette is loaded into a system operational diskette drive.

F.2 LINK EDITOR

To link FORTRAN runtime routines with FORTRAN programs, the link editor, TXSLNK/SYS, must be copied onto the diskette containing the FORTRAN runtime libraries. In addition, to use the automatic overlay feature of the link editor with FORTRAN programs, the automatic overlay supervisor module must also be copied to the diskette containing the runtime libraries. Perform the following steps to copy the information onto the diskette:

1. Insert the TXDS system diskette (911 or 913 version) in diskette drive 1 (left) and the FORTRAN runtime diskette in diskette drive 2 (right).

2. Boot load the system as described in the TXDS Programmer's Guide.

3. After the system diskette was loaded, perform the following steps to bid the control program:

   a. Enter an exclamation point (!) to bid the control program.

4. TXDS control program responds as follows when bid:

   PROGRAM:

5. Unload the system diskette from diskette drive 1 and load the Link Editor diskette in drive 1.

6. Enter the following responses to the TXDS prompts:

   CAUTION

   FORTRAN runtime diskette shipped with Write Protect.

7. Enter the following responses to the TXCS prompts:

```
 PROGRAM:   :TXCCAT/SYS
   INPUT:   DSC:TXSLNK/SYS
  OUTPUT:   DSC2:TXSLNK/SYS
 OPTIONS:
```

   These entries cause the copy/concatenate utility program to be executed. It copies the link editor to the FORTRAN runtime diskette.

8. Enter the following responses to the TXCS prompts to copy the automatic overlay supervisor module to the runtime diskette:

PROGRAM:	>10
INPUT:	DSC:TXLOVM/SYS
OUTPUT:	DSC2:TXLOVM/SYS
OPTIONS:	

F.3 FORTRAN VERIFICATION

To verify that the FORTRAN compiler and linkage editor are working properly, a test program called :TICTAC/FTN is compiled, linked and executed. The line printer (LP) is used as the print device. If the configuration does not include a line printer, the listing may be sent to the log device. Perform the following procedure to process the test program:

1. Insert the TXDS system diskette (911 or 913 version) in diskette drive 1 (left) and the FORTRAN compiler diskette in diskette drive 2 (right).

CAUTION

FORTRAN compiler diskette shipped Write Protected.

2. Boot load the system as described in the TXDS Programmer's Guide.

3. Enter the following responses to the TXDS control program prompts:

PROGRAM:	:TXFTN/SYS
INPUT:	DSC2:TICTAC/FTN
OUTPUT:	:TICTAC/OBJ,LP,DSC
OPTIONS:	M3500

These entries load and execute the FORTRAN compiler. After the test program has been compiled, control returns to the TXDS control program.

4. Remove the FORTRAN compiler diskette from drive 2 and replace it with the FORTRAN runtime diskette. The link editor must previously have been copied to the runtime diskette as described previously in this appendix.

5. Enter the following responses to the TXDS control program prompts:

PROGRAM:	:TXSLNK/SYS
INPUT:	DSC2:TICTAC/CTL
OUTPUT:	:TICTAC/GO, LP
OPTIONS:	M8000

6. Enter the following responses to the TXDS control program prompts to execute the linked program:

PROGRAM: :TICTAC/GO*

The program is loaded and executed. This program does not require LUNO assignments. It uses LUNOs that are preassigned to the system log device.

APPENDIX G

FORTRAN EXECUTION ON TXDS

G.1 GENERAL

The 990 FORTRAN compiler executes under control of TXDS and outputs linkable object modules. The compiler output must be linked with the appropriate FORTRAN directory using the link editor. Depending on the directory selection, the linked object may then be executed either under TXDS, using either OCP commands or TXDS commands, or in a standalone environment. Compiler errors are documented in Section IX of this manual.

G.2 PROGRAM GENERATION STEPS

Execution of a FORTRAN program requires the following steps:

- Program Compilation
- Program Link Edit
- Program Loading and Execution

The following paragraphs provide instructions for performing these steps using TXDS. Standalone FORTRAN is described in paragraph G.3.

G.2.1 PROGRAM COMPILATION. The FORTRAN Compiler may be run under TXDS. Refer to the *Terminal Executive Development System (TXDS) Programmer's Guide*, Part Number 946258-9701, for information about activation of TXDS.

To activate the FORTRAN Compiler, place the FORTRAN diskette in either diskette drive, input :TXFTN/SYS on the system log using the following procedure:

PROGRAM: :TXFTN/SYS (enter RETURN/NEW LINE)

The following message prompt is displayed, information is requested by the system from the user:

INPUT:	source file pathname
OUTPUT:	output file pathnames
OPTIONS:	memory specifications and compiler options

The following paragraphs discuss these inputs.

NOTE

Most TXDS utilities can be rebid after execution by entering >10 in response to the PROGRAM: prompt. However, the FORTRAN compiler cannot be rebid in this manner. Instead, reenter :TXFTN/SYS to rebid the compiler.

G.2.1.1 Input – Source File Pathname. The source file pathname can be either a device or a sequential file, the pathname defaults are listed below.

Source File Defaults

Field	Default
DEV	Default Disc name
FILE	None
EXT	FTN

Refer to the *Terminal Executive Development System (TXDS) Programmer's Guide*, Part Number 946258-9701), for a complete explanation of the pathname format.

The source file must be predefined at compilation. If the format is illegal or if the file does not exist, error messages are printed and the compiler terminates. The input file is rewound upon opening.

G.2.1.2 Output. There are three output files from TXFTN: the compiled object module, the listing, and the diskette drive on which the work files (temporary) are created. The object and listing outputs may be directed to any file or device on the system. The work file must be on diskette. If the device is illegal, a message is printed and the program terminates. If an output file does not exist, it is created with the pathname given. The pathnames must be specified in the following order: object pathname, listing pathname and work files drive name.

Object Output. The compiled object may be written to any relative record file or device capable of supporting object format data and backspace operations. Pathname defaults are listed below:

Object File Defaults

Field	Default
DEV	Default Disc Name
FILE	Input File
EXT	OBJ

Listing Output. This listing may be written to any file or device on the system. Pathname defaults are listed below:

Listing File Defaults

Field	Defaults
DEV	Default Disc Name
FILE	Input File
EXT	LST

If the pathname is null the system printer as defined at system generation time is used.

Work File Drive Name. This field is an optional field used to specify an alternate diskette drive name (i.e., DSC, DSC2) to which the temporary work files are created. The device field is the only field that is used. If this field is null (normal case), the work files are created on the diskette from which the compiler was loaded. The temporary work files are deleted upon program termination.

Example:

```
DEV:FILE/EXT,DEV:FILE/EXT,DSC or DSC2
```

G.2.1.3 Options. All options are specified by a single alphabetic character. The option specified may be followed in some cases by a numeric field. Input format is free-form such that delimiters for options may be commas, blanks or no delimiters. The options may be entered in any order. Listed below are the options recognized by TXFTN:

Option	Description
C	Conditional compilation
D	Debug trace compilation
F	Free format
O	List generated object
R	Reentrant object
S	Assembly language source
X	Symbol cross reference
Mnnnnn (n=decimal digit)	Override default symbol table size The default is 4K bytes

Conditional Compilation (C). Entering the letter C as an option specifies that all program input records having a letter D in column 1 are to be compiled with the rest of the program statements. If the letter C is not entered as an option, these input records are treated as comments.

Debug Trace Compilation (D). Entering the letter D as an option instructs the system to produce a trace listing of the program during execution. The listing shows control flow of the program by printing a message when subprograms are entered and exited, and when labeled statements are executed. Each trace line indicates the execution of a FORTRAN statement and gives the line number (as assigned in the compiler generated listing) of the statement. The name of the program that contains the statement is also printed. A trace message appears when any of the following conditions occur:

- When the task enters a subprogram, the system prints a trace line in the form:

  LINE line-number IN name, ENTRY

  where:

  line number is the line number in the output list of the FUNCTION or SUBROUTINE statement, and name is the name of the subprogram that was entered.

- When the task exits a subprogram, the system prints a trace line in the form:

  LINE line-number IN name, RETURN

  where:

  line number is the line number in the output list of the RETURN statement, and name is the name of the subprogram that was exited.

- When a labeled statement is executed, the system prints a trace line in one of the two following forms:

  LINE line-number IN name
  LINE line-number

where:

line number is the line number in the output list of the labeled statement. The parameter name indicates the name of the subprogram containing the labeled statement.

This parameter is omitted if the previous trace line contained a statement from the same subprogram.

Free Format (F). Entering the Free Format option specifies to the compiler that the input program is not in the standard format of columns 1-5, column 6 and columns 7 through 80. Instead, the compiler scans the input program according to the following rules:

- If the first nonblank character (possibly following a D during a conditional compilation) is an ampersand, the line is a continuation line.
- If the first nonblank character is a digit (0 through 9), the line is a statement.
- If the character is a C, the line is a comment.
- If the first character is a D, the line is handled by the rules of conditional compilation.

Object Code Listing (O). Entering the letter O as an option specifies to the compiler that it should print a listing of the generated object code on the device indicated as the LIST FILE: in the compiler prompting message. The object code listing option prints a large number of machine instructions for each FORTRAN statement. This option should not be selected until the FORTRAN source code is debugged.

Reentrant Object (R). Entering the letter R as an option specifies that the compiler generate object code for subprograms which use a base relative addressing method, and can therefore be used in recursive applications. The R option is not required to have shared FORTRAN subprograms under DX10 (see Appendix H, paragraph H.2.2.1). Appendix E, paragraph E.3.6 describes recursive FORTRAN applications and the use of the R option.

Assembly Source Code (S). Entering the letter S as an option specifies that the compiler should generate assembly language source code instead of object code. The source code is placed on the file designated in response to the OUTPUT: prompting message for the compiler. The source code may then be used as input to the assembler to generate an object module.

NOTE

The compiler produces optimized source code. Use care when modifying this code, since changes to register contents may produce undesireable side effects.

Variable Cross Reference List (X). Entering the letter X as an option specifies that the compiler print a listing of the program variables on the device indicated as the LIST FILE: in the compiler prompting message. Selecting the X option uses memory space to generate the list. Memory is calculated at eight words of memory per symbol and programs that overflow memory space, when this option is selected, may be able to be successfully executed by deleting X from the OPTIONS: list.

Memory Option (M). The memory option (the letter M followed by a numeric field) is used to override the default memory size. The default is 4K bytes. The syntax of the option is as follows:

Mnnnnn (where: n is a decimal digit.)

Calculating available memory for a FORTRAN compile is accomplished by reducing the size of the dynamic task area (printed out as the AVAILABLE: <parameter> when the TX990/TXDS system is loaded) by 11,661 or the size in words of the FORTRAN compiler. The result is the memory in words available for a FORTRAN compile. Multiply available words by two, thus obtaining the number of bytes available.

G.2.2 PROGRAM LINK EDIT. After the program has been compiled, the resulting object code must be linked before executing. To perform the link, a control file must be created. To activate the link editor, respond to the PROGRAM prompt with the following:

PROGRAM: :TXSLNK/SYS (Carriage RETURN/NEW LINE)

The following information is then requested:

INPUT: control file pathname
OUTPUT: output file pathnames
OPTIONS: memory specification

To activate link edit, load the FORTRAN runtime diskette in a drive:

INPUT: :TXSLNK/SYS

NOTE

Appendix F paragraphs F.6 and F.7 have to be completed before this step can be executed.

G.2.2.1 **Input.** The control file can be either a device or a sequential file. The pathname defaults are listed below.

Control File Defaults

Field	Default
DEV	Default Disc Name
FILE	None
EXT	CTL

Refer to the *Terminal Executive Development System (TXDS) Programmer's Guide* for a complete explanation of the pathname format.

The control file must exist prior to link edit. If the format is illegal or if the file does not exist, error messages are printed and the link editor terminates. The control file is rewound upon opening. Refer to the *990 Link Editor Manual,* part number 949617-9701 for error messages. The following set of commands is used for linking a FORTRAN program as a single task, except when the FORTRAN program is to run standalone. Standalone FORTRAN is described in paragraph G.3.

The following set of commands is used for linking a FORTRAN program as a single task, except when the FORTRAN program is to run standalone. Standalone FORTRAN is described in paragraph G.3.

```
NOSYMT
PHASE 0,FORT
INCLUDE DEV:FILE/EXT
FIND DSC:TXLOBJ/LIB
END
```

The name parameter must be completed with the name of the file or device that contains the FORTRAN object output. This fully qualified pathname is the same as that specified for object output during program compilation. If more than one object file is to be linked, the INCLUDE name command must be repeated for each object file. In this case, the main program file must precede all other files in the control input. The last command before the END instructs the link editor to find the runtime routines in the runtime directory. For a more detailed description of the link editor command language syntax refer to the *990 Link Editor Manual,* Part Number 949617-9701.

G.2.2.2 **Output.** There are three output files from TXSLNK: the linked object modules, the load map, and the diskette drive name to which the work files are created. Outputs may be directed to any file or device on the system, except the work files, which must be a diskette. If the device is illegal, a message is printed and the link editor terminates. If an output file does not exist, it is created with the pathname given. The pathname must be specified in the following order: object pathname, load map and work file drive name.

Object Output. The linked object may be written to any file or device capable of supporting object format data. Pathname defaults are listed below:

Object File Defaults

Field	Default
DEV	Default Disc Name
FILE	None
EXT	OBJ

Load Map Output. The load map may be written to any file or device on the system. Pathname defaults are listed below:

Load Map Defaults

Field	Default
DEV	Default Disc Name
FILE	None
EXT	LST

If the pathname is null the system printer (as defined at system generation) is used.

Work File. This field is an optional field used to specify an alternate diskette drive on which the work files are created. If this field is null (normal case) the work files are created on the diskette from which the link editor was loaded. The work files are deleted upon program termination.

G.2.2.3 Options. Only one option is applicable to TXSLNK. The option is used to override the default symbol table size. The default is 8K bytes. The option syntax is as follows:

Mnnnnn

where:

n is a decimal digit.

G.2.3 PROGRAM LOAD AND EXECUTION. Programs may be loaded and executed either through OCP commands or by responding to TXDS Control Program prompts. Refer to the TX990 Programmer's Guide for information concerning OCP commands. Refer to the TXDS Programmer's Guide for information concerning the TXDS Control Program.

G.2.3.1 OCP Loading and Execution. Perform the following steps to load and execute the program through OCP commands:

1. Activate OCP by entering an exclamation point (!).

2. Enter the Load Program (LP) command and specify the pathname of the linked object program. Default priority if none is specified is 3.

   LP, linked object pathanme [,<priority>[P]]

   where:

   P indicates the privileged mode.

3. Assign the LUNOs needed by the FORTRAN program by using the Assign LUNO (AL) command:

   AL,<luno>,<pathname>

   FORTRAN I/O unit numbers are normally decimal integers. For example, the statement:

   READ (10, 100)

   uses I/O unit number ten. These unit numbers correspond to logical unit numbers (LUNOs). The OCP AL command requires the LUNO to be a hexadecimal number. Therefore, to read a file with the above READ statement, the user must assign LUNO A to the file. For example:

   AL,A,DSC:INPUT/FTN

4. Execute the program and terminate OCP using the following commands:

   EX, 10 [,<station number>] .TE.

   The number 10 is the task ID assigned to the FORTRAN program during the load program (LP) command processing. The optional station number is the number of the VDT to be used on any ACCEPT/DISPLAY statements in the program. If station number is not specified, the default value is station 1.

G.2.3.2 Loading and Execution Under TXDS. Perform the following steps to load and execute the program using the TXDS Control Program:

1. Activate TXDS by entering an exclamation point(!). TXDS responds with:

   PROGRAM:

2. Assign any LUNOs needed by the FORTRAN program by executing the Assign/Release LUNO task. Refer to the TXDS Programmer's Guide for a detailed description of the task. The following steps activate that task:

   a. Enter the following responses to the PROGRAM: prompt to load the task:

      PROGRAM: :TXLUNO/SYS*

      The task is loaded and responds with:

      LUNO?

   b. Respond to the task prompts as follows:

      LUNO? <luno 1>
      PATHNAME? <user pathname 1>
      LUNO? <luno 2>
      PATHNAME? <user pathname 2>
      LUNO? *

      Entering an asterisk (*) in response to the LUNO? prompt terminates the task. FORTRAN I/O unit numbers are decimal integers. For example, the statements:

      READ (10, 100)

      uses I/O unit number ten. These unit numbers correspond to logical unit numbers (LUNOs). Therefore, to read a file with the above READ statement, the user must assign LUNO 10 to the file. For example:

      LUNO? 10
      PATHNAME? DSC:INPUT/FTN

3. Enter the following responses to the Control Program prompts to load and execute the FORTRAN Program:

   PROGRAM: <linked object pathanme>
   INPUT:
   OUTPUT:
   OPTIONS: [STxx]

The letters "xx" represent a two-character decimal number that gives the VDT station number used on ACCEPT/DISPLAY statements. If STxx is not specified, the default is station 1. For example:

OPTIONS: ST05

instructs the FORTRAN program to use station 5 during any ACCEPT/DISPLAY statement execution.

NOTE

The station number associated with a VDT is established at system generation time. For a standard TXDS system, the VDT is station number 1.

G.3 STANDALONE FORTRAN

Creation of FORTRAN programs for running standalone is supported on TXDS; table E-1 provides a list of operating system dependent FORTRAN statements. These can *not* be used in a FORTRAN program fpr the standalone environment.

G.3.1 COMPILE AND LINK EDIT. The standalone FORTRAN source program should be compiled the same as for FORTRAN to be run under TXDS (see paragraph G.2.1). The link edit process is the same as that described in paragraph G.2.2 except that different directories are used in the control file. The following is an example of the control file for linking standalone FORTRAN:

```
NOSYMT
TASK FORT          <FORMAT COMPRESSED
INCLUDE <name>
FIND DSC:SALOBJ/LIB
FIND DSC:TXLOBJ/LIB
END
```

For more information on standalone FORTRAN refer to Appendix E.

G.3.2 LOADING STANDALONE PROGRAMS. The FORTRAN standalone programs may be loaded by diskette/cassette ROM loader, card/cassette ROM loader, or a special user-supplied loader. If the ROM loader is used, the object input must be on cassette, diskette, or punched cards. If the linked object output from the Program Development System Link Editor was not on punched cards or cassettes, the linked object file may be transferred to one of these mediums by using the copy/concatenate function of the Terminal Executive Program Development System.

If the linked standalone program is a file (on diskette), load the SYSUTL program and use the Set System File (SF) command to define the operating system file as the linked standalone program file. The format of the SF command is:

```
SF,<pathname>.
```

To load the standalone program, perform the following steps:

1. Ready the object input device.
2. Press HALT/SIE button on the front panel.
3. Press the RESET button.
4. Press the CLEAR button.
5. If punched cards are being loaded, perform the following steps:
   a. Enter 80_{16} on the data switches.
   b. Press the MA ENTER switch.
   c. Press the MDE switch.

6. The default load address is $A0_{16}$. If a different load address is required, perform the following steps:

   a. Enter 92_{16} on the data switches.

   b. Press the MA ENTER switch.

   c. Enter the desired load address on the data switches.

   d. Press the MDE switch.

7. Press the LOAD switch.

The ROM loader then loads from the input device until a colon card is encountered.

G.3.3 EXECUTING STANDALONE PROGRAMS. If a standalone program is loaded by the ROM loader, the loader automatically transfers execution control to the loaded program. If the user uses his own loader, he must specify the procedure for initiating execution. There are two ways that the program may be entered:

- Word 0 of the program contains the initial workspace and word 1 contains the initial entry point. If the program is entered by setting the WP and PC to these values, the status register is not reset.

- The linked object contains a different entry point address from that in 1 above (a 2 tag). If the program is entered at this entry point, the WP is automatically set to the initial workspace and the status register is reset.

G.4 ERROR LOGGING AND TERMINATION OF TXDS

Errors and warnings detected by the FORTRAN runtime support package are written by an error logging routine upon logical unit 0. This logical unit is normally assigned to the system logging device. All messages are the same as those output by the standard directory. Termination is affected by executing an end-of-job supervisor call. Upon normal termination, all user files are closed with an end-of-file before the program is terminated. Termination due to a fatal error leaves all user files open for the operating system to close after program termination. This includes errors that cause the end vector to be taken.

G.5 LUNO ASSIGNMENT UNDER TXDS

FORTRAN under TXDS maps unit numbers directly to LUNOs. This can cause several problems if care is not exercised in choosing FORTRAN unit numbers. Furthermore, VDTs behave in a more restricted manner under TXDS, and caution must be exercised when assigning I/O to VDTs. The following paragraphs discuss the procedure to be followed for proper FORTRAN program execution under TXDS.

G.5.1 ASSIGNING LUNOs TO VDTs ON TXDS. The first access to a LUNO does an open rewind supervisor call. For a VDT, an open/rewind SVC erases the screen. This means that if separate LUNOs are used to read and write to a CRT, and a prompt is written before a read, the screen is erased and the prompt lost. The FORTRAN unit number required by ACCEPT/DISPLAY statements is not used by TXDS FORTRAN. All ACCEPT/DISPLAY statements will perform their requested functions to the designated station (or to the system log by default).

G.5.2 APPLICATION PROGRAM LUNO ASSIGNMENTS. LUNO assignments for tasks that run under TXDS (or TX) are done outside of the task by use of the ASSIGN LUNO command under OCP or the TX LUNO utility under TXDS. Since the FORTRAN unit number is the same as the LUNO assignment under TXDS, choosing standard FORTRAN unit numbers below 20_{10} may cause conflicts with TXDS system LUNOs. This is true for both single- and multiple-task execution under TXDS. Furthermore, if multiple-task execution is to be accomplished successfully under TXDS, the standard FORTRAN unit numbers must be unique for each execution of the task. For example, if the I/O statements

```
READ (20,200) . . .
```

are to be executed from more than one task which may be active under TXDS at the same time, whichever device is assigned to LUNO 20 will be used by both tasks. Also, if the same task is to be executed, for example, at more than one VDT, the FORTRAN unit numbers in each copy of the task must be unique. To avoid conflicts of this nature, the following guideline can be used:

All I/O statements should be of the form

```
DATA INUM/20/
(READ )  (INUM, . . . .
{WRITE}
(etc. )
```

This will permit changing INUM (by patching) without recompiling should a different FORTRAN unit number be required for any reason.

G.6 TX990/TXDS SYSTEM CONSIDERATIONS

To sysgen a system that is able to execute the FORTRAN compiler, refer to *TX990 Operating System Programmer's Guide, Section IX, System Generation.*

G.7 RECONFIGURING TX990/TXDS FOR FORTRAN PROGRAM EXECUTION

The user may configure the TX990/TXDS system to increase the amount of memory available to user FORTRAN programs. The following features can be added to the base system as a function of the program requirements:

- Base System – Supports task scheduling and SVC execution[1]
- Diskette DSR and file management package to allow performance of file I/O[1]
- 911 VDT DSR and station handler to allow the program to read and write to the VDT in record mode[1]
- 911 VDT utility to allow the program to execute ACCEPT/DISPLAY statements[1]
- TXDS Control Program to allow the FORTRAN program to be dynamically loaded
- Manual restart capability to allow a manual restart rather than requiring a reload of the system to restart the system.

Table G-1 lists the memory size requirements of the modules involved in adding these features.

[1] Note: For the program to execute, the following conditions must be met:

1. The program must be loaded with the operating system.
2. The desired FORTRAN logical units must be assigned at system generation.
3. The initial task state must be set to active when the operating system is loaded.

Table G-1. Module Memory Size Requirements

Module	Size (Bytes)	Sub Total	Cumulative Total
TXDATA	1200*		
TASKDF	104*		
TXROOT	1458		
IOSUPR	974		
TBUFMG	130		
TASKFUN	292		
DTASK	178		
TXEND	12	4348	4348
FPYDSR	1090		
FMPLIB	5540		
FMP tasks	456		
Buffers for 3 Files	672	7758	12106
STA911	270		
DSR911	782	1052	13158
CRTPRO	164		
SVC911	654	818	13976
CNTROL	1418		
COMMON	170		
MEMSVC	36	1624	15600
TXSTRT	208	208	15808

*These modules vary in size with the number of tasks and devices in the system.

APPENDIX H

FORTRAN EXECUTION ON A DX10 3.X SYSTEM

H.1 GENERAL

FORTRAN programs that are compiled by the 990/10 DX10 System FORTRAN compiler can be linked for installation and execution in three environments. These environments are DX10 with synonym assigned LUNOs, DX10 with globally assigned LUNOs, and standalone. The following paragraphs describe the procedures for developing programs for each operating environment.

NOTE

> The following description of FORTRAN execution is a general description designed to help the user operate in TTY mode or VDT mode. There are slight differences in operation between TTY and VDT mode, a major difference being one of prompt messages. TTY prompt messages are displayed one line at a time and the user must satisfy that prompt before the next prompt message is displayed. In VDT mode, the prompt messages are all displayed at one time, one message per line. The 911 VDT, 913 VDT, and 733 ASR have different keyboards, and screen dimensions. Refer to *Model 990 Computer DX10 Operating System Reference Manual, Volume II, Production Operations.*

H.2 PROGRAM EXECUTION

Execution of a FORTRAN source program requires the following steps:

1. Program Compilation
2. Program Link-Edit
3. Task Installation or Program Loading
4. Establishing I/O Unit Correspondence to Files and Devices
5. Program Execution.

The following paragraphs describe the actions required to perform each of these steps and successfully execute the FORTRAN source program.

H.2.1 PROGRAM COMPILATION. FORTRAN source programs must be compiled by the 990/10 DX10 FORTRAN compiler for all of the environments under which the program may run. To execute the FORTRAN compiler, enter the command XFC or XFCF (background or foreground, respectively). The following prompt messages are displayed:

```
SOURCE ACCESS NAME:
OBJECT ACCESS NAME:
LISTING ACCESS NAME:
           OPTIONS:
       PRINT WIDTH:  80
```

The cursor moves to the first field aligned for user response. All prompts must be filled in followed by a carriage RETURN (key) to activate the FORTRAN compiler. Pressing the carriage RETURN following the last prompt entry activates the compiler. When the compiler has finished (XFCF – foreground), the terminal local file is displayed showing errors, warnings, and either a normal or abnormal termination. When executing in the background mode (XFC), display is provided by executing a Show Background Status (SBS) command.

H.2.1.1 **Source Access Name.** The source access name parameter may be completed with any device name (input) or DX10 pathname. Device name or pathname designates to the compiler where it can find the FORTRAN source program to be compiled.

H.2.1.2 **Object Access Name.** The object access name parameter may be completed with any device name (output) or DX10 pathname. Device name or pathname designates to the compiler where it may store the program's compiled object for later input to the Link Editor.

H.2.1.3 **Listing Access Name.** The listing access name parameter may be completed with any device name (output) or DX10 pathname designating to the compiler where it may output any requested listing.

H.2.1.4 **Options.** The options parameter may be completed with any character or group of characters from the list in table H-1. The characters may be selected in any order. These options specify functions that are available and are performed by the compiler as described in the following paragraphs. Selected options remain in effect for all modules presently in the source file.

Conditional Compilation (C). Entering the letter C as an option specifies to the compiler that all program input records having a letter D in column 1 are to be compiled with the rest of the program statements. If the letter C is not entered as an option, these input records are treated as comments.

Debug Trace Compilation (D). Entering the letter D as an option directs the system to produce a trace listing of the program during execution. The listing is written on the terminal local file.

Table H-1. FORTRAN Compiler Selectable Options

Option Character	Selected Option
X	Variable cross reference list
O	List the generated object code
C	Conditional compilation
D	Debug trace compilation
F	Free format
R	Reentrant object
S	Assembly language source

At the completion of program execution, the processor displays the contents of this file on the terminal screen. The file is created each time the program is executed, so that the previous contents of the file are destroyed. The listing shows control flow of the program by printing a message when subprograms are entered and exited, and when labeled statements are executed. Each trace line indicates the execution of a FORTRAN statement and gives the line number of the statement (as assigned in the compiler generated listing). The name of the program unit in which the statement resides is also printed. A trace message appears when any of the following conditions occur:

- When the task enters a subprogram, the system prints a trace line in the form:

  LINE line-number IN name, ENTRY

  Where "line-number" is the line number in the output list of the FUNCTION or SUBROUTINE statement, and "name" is the name of the subprogram that was entered.

- When the task exits a subprogram, the system prints a trace line in the form:

  LINE line-number IN name, RETURN

  Where "line-number" is the line number in the output list of the RETURN statement, and "name" is the name of the subprogram that was exited.

- When a labeled statement is executed, the system prints a trace line in one of the two following forms:

  LINE line-number IN name

  LINE line-number

  In either case, "line-number" is the line number in the output list of the labeled statement. The parameter "name" indicates the name of the subprogram containing the labeled statement. This parameter is omitted if the previous trace line contained a statement from the same subprogram.

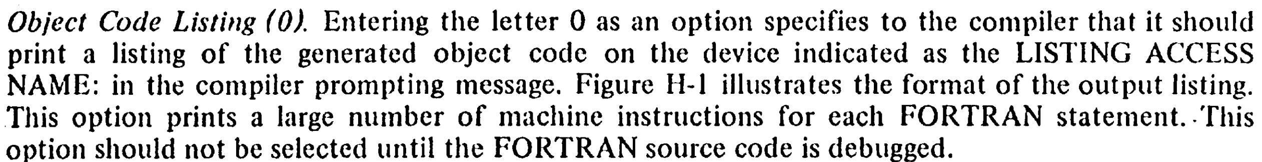
Free Format (F). Entering the letter F as an option specifies to the compiler that the input program is not in the standard format of columns 1-5, column 6 and columns 7 through 80. Instead, the compiler scans the input program according to the following rules:

- If the first nonblank character (possible following a D during a conditional compilation) is an ampersand, the line is a continuation line.

- If the first nonblank character is a digit (0 through 9), the line is a statement.

- If the first character is a C, the line is a comment.

- If the first character is a D, the line is handled by the rules of conditional compilation.



Object Code Listing (0). Entering the letter 0 as an option specifies to the compiler that it should print a listing of the generated object code on the device indicated as the LISTING ACCESS NAME: in the compiler prompting message. Figure H-1 illustrates the format of the output listing. This option prints a large number of machine instructions for each FORTRAN statement. This option should not be selected until the FORTRAN source code is debugged.

Reentrant Object (R). Entering the letter R as an option specifies that the compiler generate object code for subprograms which use a base relative addressing method, and can therefore be used in recursive applications. The R option is not required to have shared FORTRAN subprograms under DX10 (see paragraph H.2.2.1). Paragraph E.3.6 describes recursive FORTRAN applications and the use of the R option.

Variable Cross Reference List (X). Entering the letter X as an option specifies that the compiler should print a listing of the program variables on the device indicated as the LISTING ACCESS NAME: in the compiler prompting message. Figure H-2 illustrates the format of the output listing. Selecting this option requires additional memory space to generate the list. Therefore, programs that overflow memory space when this option is selected, may be able to be successfully executed by deleting X from the OPTIONS: list.

Assembly Source Code (S). Entering the letter S as an option specifies that the compiler should generate 990 assembly language source instead of object code. The source is placed on the file indicated as the OBJECT ACCESS NAME: in the compiler prompting message. The source may then be used as input to the macro assembler to generate an object module.

NOTE

> Care should be taken in modifying the source produced by the compiler since optimization takes place and arbitrary changes to register contents may produce undesirable side effects.

H.2.1.5 Print Width. The print width parameter specifies the number of characters (bytes) in the output listing. Press carriage RETURN (KEY) to select the default value of 80 characters. Should the user desire a print width other than 80 characters, the user may enter (key-in) the desired value and then press carriage RETURN.

H.2.2 PROGRAM LINK EDIT. After the program has been compiled, the resulting object code must be linked with directories of runtime support routines. The link edit must be performed by SDSLNK. If specified, the FORTRAN program may also be installed at link edit time, see paragraph H.2.2.5. To perform the link, a control file must be created using the text editor. The control file created must be assigned a valid pathname. Several example contents of the control file are described later in this appendix.

Once the control file is complete, activate the link editor (SDSLNK) by selecting XLE. The link editor displays the following prompting message:

```
       CONTROL ACCESS NAME:
 LINKED OUTPUT ACCESS NAME:
       LISTING ACCESS NAME:
              PRINT WIDTH: 80
```

```
0033   40    X  = 0.0
                  00A8              #40    EQU    $
                  00A2   1602  A           JNE    $+3
                  00A8   C820  A           MOV    @R#1,@X
                  00AA   0000  A
                  00AC   00FE  D
                  00AE   C820  A           MOV    @R#1+2,@X+2
                  00B0   0000  A
                  00B2   0100  D
0034   60    Y  = Y + 1.0
                  00B4              #60    EQU    $
                  00B4   0420  A           BLWP   @F$RITP
                  00B6   005E  R
                  00B8   0DA0  A           LR     @R#3
                  00B4   0000  A
                  00BC   0C60  A           AR     @Y
                  00BE   010A  D
                  00C0   0DE0  A           STR    @Y
                  00C2   010A  D
0035         IF (I)  70,70,80
                  00C4   0C0E  A           XIT
                  00C6   C0A0  A           MOV    @1,2
                  00C8   00FC  D
                  00CA   C082  A           MOV    2.2
                  00CC   15FF  A           JGT    #80
                  00CE   13FF  A           JEQ    #70
                  00D0   11FF  A           JLT    #70
0036         CONTINUE
                  00D2              #2     EQU    $
```

* CODE TYPE INDICATES HOW THE OBJECT CODE COLUMN IS REPRESENTED.

A = ABSOLUTE CODE
D = DATA SEGMENT (DSEG) RELATIVE
R = PROGRAM SEGMENT (PSEG) RELATIVE

(A)133470

Figure H-1. Object Option Format Sample

```
CROSS REFERENCE

A       /0003/   0044     0044     0045     0045     0046     0046
AA      /0004/  /0005/   *0048*    0048
B       /0003/   0044     0044     0045     0045     0046     0046
C       /0003/   0045     0046
EXT     /0001/
I       *0015*  *0016*   *0017*    0019     0021     0025    *0027*    0031
         0035    0048     0048     0048     0048     0048
J        0013    0014     0020    *0039*    0039     0048     0048     0048
K       *0009*   0009    *0010*    0010    *0011*    0011    *0012*    0012
        *0013*   0013    *0014*    0014     0048     0048
NEXT    *0032*
SAM      0047    0047     0047
SQRT     0047    0047
X       *0018*   0022    *0023*   *0029*   *0030*   *0033*   *0037*   *0038*
         0038
XX      /0002/  *0044*   *0047*
Y       *0034*   0034     0037
YY      /0002/  *0046*
ZZ      /0002/  *0045*
```

OUTPUT LINE NUMBERS THAT CONTAIN THE VARIABLE. OTHER SYMBOLS INDICATE:

/...../ = VARIABLE DECLARATION STATEMENT

..... = POSSIBLE MODIFICATION OF VARIABLE

ƀ.....ƀ = OTHER VARIABLE REFERENCE

VARIABLE NAME (UP TO 6 CHARACTERS)

(A)133471

Figure H-2. Variable Cross Reference Listing

These parameters must be completed to activate the link editor. The control access name is the name assigned to the control file created at the start of the program link edit procedure. The linked output access name receives the output of the link edit procedure. The listing access name is the device or file that receives the load map listing of the link edit. The link editor creates the linked output file and/or listing file if they do not exist. The print width parameter specifies the number of characters (bytes) in the output listing. The default value is 80. Pressing the carriage RETURN key when the cursor is in the print width field activates the link editor. When the link edit is complete, SDSLNK displays the terminal local file saying that the linking is complete and giving the warnings and errors.

H.2.2.1 Linking for Execution Under DX10 With Synonym Assignment. Many of the runtime support routines in the FORTRAN runtime directories are coded reentrantly. Therefore, the compiler generated object code can be linked to produce either a single nonreentrant task or a nonreentrant task with a reentrant procedure. In the latter case, several FORTRAN tasks can use a common reentrant procedure and drastically reduce the total memory requirements in a multi-program environment. The next two paragraphs describe the contents of the link editor control file for these two cases.

Nonreentrant Tasks. To link the compiler-generated object code as a single nonreentrant task, the following link editor control commands can be used:

```
NOSYMT
LIBRARY.FORTRN.OSLOBJ
LIBRARY.FORTRN.STLOBJ
PHASE 0, FORT
INCLUDE name
END
```

The standard runtime directories have the pathnames .FORTRN.OSLOBJ and .FORTRN.STLOBJ. The library cards *must* be in this order.

The name parameter must be completed with the pathname of the FORTRAN object output. This parameter is the same as that specified for the object file during the program compilation phase. If more than one object file is to be linked, the INCLUDE name command must be repeated for each object file. In this case, the INCLUDE for the main program file must precede all other INCLUDE commands in the control input.

The runtime support routines required for executing the FORTRAN program are automatically included in the linked object.

Nonreentrant Task With Reentrant Procedure. Any FORTRAN subprogram may be linked as part of a shared procedure under the DX10 operating system. In addition, the routines which form the FORTRAN run-time library may also be shared in this way. This feature is made possible by the separation of read/write data from executable code via the DSEG assembler directive. The only limitation present is the requirement that all tasks using the shared procedure have entry modules (i.e., main program) of equal length. This problem can be solved by using the following main program for all tasks using the shared procedure:

```
CALL MAIN
END
```

This fixed length dummy main program calls subroutine MAIN which is the routine which would normally be the main program. Any main program can be converted to subroutine MAIN by inserting a SUBROUTINE MAIN statement before the rest of the source statements. After making the above changes, the user should link his tasks in the following way:

```
NOSYMT
LIBRARY (any special library)
  .
  .
  .
LIBRARY .FORTRN.OSLOBJ
LIBRARY .FORTRN.STLOBJ
PROCEDURE FTNPRC
DUMMY (include this directive in all but one task)
INCLUDE (shared FORTRAN subprogram)
  .
  .
  .
INCLUDE (shared FORTRAN runtime routine)
  .
  .
  .
SEARCH
TASK FORT
INCLUDE (dummy main module)
ALLOCATE
INCLUDE (subroutine MAIN)
```

```
INCLUDE (unshared FORTRAN subprogram)
      .
      .
      .
INCLUDE (any other required routines)
END
```

When installing the procedure and the tasks, the procedure must be installed from the linked output file from the link which did not contain the DUMMY directive. Refer to paragraph H.2.3.2 for information on installing a reentrant procedure and a nonreentrant task.

Linking for Shared Data Block. FORTRAN programs which use a shared data block may be linked reentrantly. The form of the control file for this follows:

```
NOSYMT
LIBRARY      .FORTRN.OSLOBJ
LIBRARY      .FORTRN.STLOBJ
PROCEDURE    DATBLK
INCLUDE         name
TASK FTNPRG
INCLUDE         name 1
END
```

The standard directories have the pathnames .FORTRN.OSLOBJ and .FORTRN.STLOBJ. The name parameter is the pathname of the data block to be shared. The name 1 parameter is the pathname of the FORTRAN object output of one of the programs sharing the data block. This parameter is the same as that specified for the OBJECT ACCESS NAME during the program compilation phase.

Linking for Global Common. FORTRAN programs that share a common area may be linked reentrantly. The form of the control file is:

```
NOSYMT
LIBRARY      .FORTRN.OSLOBJ
LIBRARY      .FORTRN.STLOBJ
PROCEDURE       PROC1
INCLUDE         name 1
PROCEDURE       PROC2
INCLUDE         pathname 1
INCLUDE         name 2
TASK         TSK1
INCLUDE         name 3
INCLUDE         pathname 2
END
```

The standard directories have the pathnames .FORTRN.OSLOBJ and .FORTRN.STLOBJ. The name 1 parameter is the pathname of the FORTRAN program containing the common block (i.e. a BLOCK DATA subprogram that defines the common to be stored). The pathname 1 parameter is the pathname of a reentrant FORTRAN runtime routine. There must be an INCLUDE statement for each reentrant routine. The name 2 parameter is the FORTRAN object output pathname that is to be shared. The name 3 parameter is the FORTRAN object output pathname that is not to be shared. This code may call the reentrant portion of the program. The pathname 2 parameter is the pathname of nonreentrant FORTRAN runtime routines.

Figure H-3 shows the BLOCK DATA subprogram and the two programs (task 1 and task 2) that will share the COMMON. Figures H-4 and H-5 show the two link control files required to link the two programs in such a manner that they communicate through a pseudo "global common" defined by the block data subprogram. The COMMON is placed in PROCEDURE 1 which is shared by both tasks. Task 1 is executed first and establishes a loop that looks for a value computed in task 2. Task 2 then executes and when complete issues that value for task 1 and STOPs. Task 1 satisfies IF statement and STOPs. *Model 990 Computer DX10 Operating System Reference Manual, Application Programmer's Guide,* part number 946250-9703, provides additional information describing the shared procedure.

H.2.2.2 Linking For Execution Under DX10 For Global LUNOs. Programs that are linked for installation and execution under DX10, but that use global LUNOs, must be linked with the same two standard DX10 directories plus an additional directory. Below is an example link control file:

```
NOSYMT
LIBRARY     .FORTRN.DXLOBJ    SPECIAL DIRECTORY FOR GLOBAL LUNOs
LIBRARY     .FORTRN.OSLOBJ    STANDARD DIRECTORY
LIBRARY     .FORTRN.STLOBJ    STANDARD DIRECTORY
PHASE 0,    FORT
INCLUDE     name
END
```

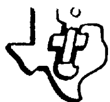
The special runtime directory for DX10 programs for global LUNO assignment has the pathname .FORTRN.DXLOBJ and the standard runtime directories have the pathnames .FORTRN.OSLOBJ and .FORTRN.STLOBJ. The library cards *must* be in this order.

The name parameter must be completed with the pathname of the FORTRAN object output. This parameter is the same as that specified for the OBJECT ACCESS NAME during the program compilation phase. If more than one object file is to be linked, the INCLUDE name command must be repeated for each object file. In this case, the INCLUDE for the main program file must precede all other INCLUDE commands in the control input.

The above control file is used to link a nonreentrant task. The other control files used to link nonreentrant task with a reentrant procedure, to link a shared data block, and to link global common are described in paragraph H.2.2.1. The formats in this paragraph may be used for global LUNOs linkage provided the additional LIBRARY card is added.

H.2.2.3 Linking for Execution Under TX990. Programs that are linked for installation and execution under TX990 must be linked with the two specified standard DX10 directories plus an additional directory. The following is an example link control file:

```
NOSYMT
LIBRARY     .FORTRN.TXLOBJ    SPECIAL DIRECTORY FOR TX990
LIBRARY     .FORTRN.OSLOBJ    STANDARD DIRECTORY
LIBRARY     .FORTRN.STLOBJ    STANDARD DIRECTORY
PHASE 0,    FORT
INCLUDE     <name>
END
```



```
      BLOCK DATA
      COMMON /BLOCK/ A,B,C,D,E,F
      END
```

*BLOCK DATA Subprogram

```
C  PROGRAM NAMED MAIN1
      COMMON /BLOCK/ A, B, C, D, E, F
      IMPLICIT INTEGER*2 (A-Z)
      A = 1
      B = 20
      C = 50
      D = 25
      E = 25
      F = 15
50    X =  C  + D
      WRITE (17, 100) X
100   FORMAT (1X, 'X =', I10)
150   CONTINUE
      IF (A .EQ. 2) GO TO 200
      CALL WAIT (1,1,ISTAT)
      GO TO 150
200   WRITE (17,250) B
250   FORMAT (1X,' B =',I7)
      STOP
      END
```

*Output from MAIN1

```
      X =        75
       B =    200
```

*Program A

```
C  PROGRAM NAMED MAIN2
      COMMON /BLOCK/ A, B, C, D, E, F
      IMPLICIT INTEGER*2 (A-Z)
      Y = 0
50    Y = E + F + Y
      WRITE (18, 100) Y
100   FORMAT (1X, 'Y =', I10)
      CALL WAIT (1,1,ISTAT)
      IF (Y .LT. 200) GO TO 50
      B = Y
      A = 2
      STOP
      END
```

*Output from MAIN2

```
      Y =        40
      Y =        80
      Y =       120
      Y =       160
      Y =       200
```

*Program B

Figure H-3. BLOCK DATA Subprogram and the Two Programs that Share COMMON

```
FORMAT IMAGE
NOSYMT
LIBRARY  .FORTRN.OSLOBJ
LIBRARY  .FORTRN.STLOBJ
PROCEDURE  PROC1
INCLUDE .FOROBJ.F1
TASK TASK1
INCLUDE .FOROBJ.F2
END.
```

```
LINK MAP

CONTROL FILE = .SMITH.SOR.L1

LINKED OUTPUT FILE = .FORPROG

LIST FILE = .LISTF1

NUMBER OF OUTPUT RECORDS = 64

OUTPUT FORMAT = IMAGE


PROCEDURE 1, PROC1    ORIGIN = 0000  LENGTH = 0018   (PROCEDURE ID = 1)

MODULE     NO   ORIGIN   LENGTH    TYPE       DATE       TIME      CREATOR

$BLOCK     1    0000     0000      INCLUDE    07/25/77   11:51:06  FTN990


COMMON     NO   ORIGIN   LENGTH

BLOCK      1    0000     0018

PHASE 0, TASK1    ORIGIN = 0020  LENGTH = 42A8   (TASK ID = 1)

MODULE     NO   ORIGIN   LENGTH    TYPE       DATE       TIME      CREATOR

$MAIN      2    0020     00AA      INCLUDE    07/25/77   11:52:06  FTN990
$DATA      2    3D32     0034
F$RFZ      3    00CA     0354      LIBRARY    04/26/77   18:27:32  SDSMAC
F$RINP     4    041E     0074      LIBRARY    04/26/77   18:25:10  SDSMAC
WAIT       5    0492     0114      LIBRARY    05/21/76   11:46:38  FTN990
$DATA      5    3D66     004A
F$RPAU     6    05A6     00B0      LIBRARY    12/16/76   16:42:02  SDSMAC
F$XPRE     7    0656     004A      LIBRARY    12/03/76   17:42:52  SDSMAC
F$ERRC     8    06A0     013C      LIBRARY    12/16/76   16:13:13  SDSMAC
F$RFTS     9    07DC     006A      LIBRARY    05/21/76   17:49:04  SDSMAC
```

Figure H-4. MAIN1 Link Map

```
FORMAT IMAGE
LIBRARY  .FORTRN.OSLOBJ
LIBRARY  .FORTRN.STLOBJ
PROCEDURE  PROC1
DUMMY
INCLUDE .FOROBJ.F1
TASK TASK2
INCLUDE .FOROBJ.F3
END

LINK MAP

CONTROL FILE = .SMITH.SOR.L2

LINKED OUTPUT FILE = .FORPROG

LIST FILE = .LISTF2

NUMBER OF OUTPUT RECORDS = 63

OUTPUT FORMAT = IMAGE

PROCEDURE 1, PROC1    ORIGIN = 0000  LENGTH = 0018, DUMMY   (PROCEDURE ID = 1)


MODULE    NO   ORIGIN   LENGTH    TYPE       DATE      TIME      CREATOR

$BLOCK    1     0000     0000    INCLUDE   07/25/77  11:51:06   FTN990


COMMON    NO   ORIGIN   LENGTH

BLOCK     1     0000     0018

PHASE 0, TASK2    ORIGIN = 0020  LENGTH = 4270   (TASK ID = 2)


MODULE    NO   ORIGIN   LENGTH    TYPE       DATE      TIME      CREATOR

$MAIN     2     0020     0072    INCLUDE   08/02/77  17:12:06   FTN990
$DATA     2     3CFA     0034
F$RFZ     3     0092     0354    LIBRARY   04/26/77  18:27:32   SDSMAC
F$RINP    4     03E6     0074    LIBRARY   04/26/77  18:25:10   SDSMAC
WAIT      5     045A     0114    LIBRARY   05/21/76  11:46:38   FTN990
$DATA     5     3D2E     004A
F$RPAU    6     056E     00B0    LIBRARY   12/16/76  16:42:02   SDSMAC
F$XPRE    7     061E     004A    LIBRARY   12/03/76  17:42:52   SDSMAC
F$ERRC    8     0668     013C    LIBRARY   12/16/76  16:13:13   SDSMAC
F$RFTS    9     07A4     006A    LIBRARY   05/21/76  17:49:04   SDSMAC
F$FINP    10    080E     0B52    LIBRARY   07/19/77  13:35:41   SDSMAC
F$RBUF    11    1360     008C    LIBRARY   02/18/77  07:52:42   SDSMAC
```

Figure H-5. MAIN2 Link Map

The special runtime directory for TX990 has the pathname .FORTRN.TXLOBJ. The standard runtime directories have the pathnames .FORTRN.OSLOBJ and .FORTRN.STLOBJ. To ensure proper selection of modules, the library cards must be in the order indicated in the example. The <name> parameter must be completed with the pathname of the FORTRAN object output. This parameter is the same as that specified for the OBJECT ACCESS NAME during the program compilation phase. If more than one object file is to be linked, the INCLUDE <name> command must be repeated for each object file. In this case, the INCLUDE for the main program file must precede all other INCLUDE commands in the control input.

H.2.2.4 Linking For Standalone Execution. The special standalone FORTRAN directory is used to produce FORTRAN programs that can execute as standalone programs. The general form for the link edit control file is:

```
NOSYMT
LIBRARY      .FORTRN.SALOBJ
LIBRARY      .FORTRN.STLOBJ
PHASE 0,     FORT
INCLUDE      name
END
```

The LIBRARY commands *must* be in the order shown. The special standalone directory has the pathname .FORTRN.SALOBJ and the standard directory has the pathname .FORTRN.STLOBJ.

The functions provided in the special standalone directory are program entry and setup, program termination, and message logging. All other functions required by the FORTRAN program will be supplied from the standard directory. This standard directory includes only functions that do not require operating support. If the user has referenced any function that requires operating system support such as I/O functions, the link edit list file shows unresolved references. See table E-1 for operating system dependent functions.

If the user wishes to use the FORTRAN I/O statements, he must provide his own support routines. The interface between the compiler generated code and the I/O support routines and additional information about standalone execution is documented in Appendix E.

H.2.2.5 Link Edit Installation of FORTRAN Program. The FORTRAN program may be installed during link edit by changing the control file. The following is an example:

```
FORMAT IMAGE
NOSYMT
LIBRARY      .FORTRN.OSLOBJ
LIBRARY      .FORTRN.STLOBJ
PHASE 0,     FORT
INCLUDE      .TEST
END
```

The first statement selects image mode and instructs the link editor to install the program. It installs both procedures and tasks. The rest of the control file is the same as for any other link.

The linked output is put on a program file. The task ID assigned appears on the list file. See the *Link Editor Manual* for more information.

H.2.3 PROGRAM INSTALLATION. DX10 application programs must be installed either by the link editor or by use of the DX10 System Command Interpreter install commands before execution is possible. The installation procedure is slightly different depending on whether the program was linked as a single nonreentrant task, a nonreentrant task and a reentrant procedure, or a non-reentrant procedure with dummy output. The following paragraphs apply to both synonym assigned LUNO linkage and global assigned LUNO linkage.

H.2.3.1 Install Single Nonreentrant Task. Enter the IT command causing the following prompt message to be displayed:

```
        INSTALL TASK
           PROGRAM FILE OR LUNO:
                     TASK NAME:
                       TASK ID:   0
      OBJECT PATHNAME OR LUNO:
                      PRIORITY:   4
           DEFAULT TASK FLAGS?:   YES
          ATTACHED PROCEDURES?:   NO
```

If the answer to DEFAULT TASK FLAGS? is NO, the following is displayed:

```
                   PRIVILEGED?:   NO
                  SYSTEM TASK?:   NO
              MEMORY RESIDENT?:   NO
                REPLICATABLE?:    YES
            DELETE PROTECTED?:    NO
           EXECUTE PROTECTED?:    NO
           OVERFLOW CHECKING?:    NO
    WRITABLE CONTROL STORAGE?:    NO
```

If the answer to ATTACHED PROCEDURES is YES, the following is displayed:

```
       ATTACH TASK PROCEDURES
            1st PROCEDURE ID:   0
  P1 FROM TASKS PROGRAM FILE?   YES
            2nd PROCEDURE ID:   0
  P2 FROM TASKS PROGRAM FILE?   YES
```

The program file or LUNO parameter designates the program file in which to install the task. The task name is the name given to the task by the user. The task ID is the desired task ID number that is assigned to the task. If the input is zero a task ID is supplied by the system. The object pathname or LUNO is the pathname assigned to the linked output file during link edit or the LUNO to which this linked object has previously been assigned. The priority may be 1, 2, 3, or 4. The reply to the prompt DEFAULT TASK FLAGS? is YES, and the reply to the prompt ATTACHED PROCEDURES? is NO.

H.2.3.2 Install Nonreentrant Task and Reentrant Procedure. Enter the IP command causing the following prompt to be displayed:

```
               INSTALL PROCEDURE
          PROGRAM FILE OR LUNO:
                PROCEDURE NAME:
                  PROCEDURE ID:
    OBJECT PATHNAME OR LUNO:
             MEMORY RESIDENT?:   NO
              DELETE PROTECT?:   NO
             EXECUTE PROTECT?:   NO
               WRITE PROTECT?:   NO
 WRITABLE CONTROL STORAGE?:   NO
```

The program file or LUNO parameter designates the program file in which to install the procedure. The procedure name is the name given to the procedure by the user. The procedure ID is the desired procedure ID number that is assigned to the procedure. The object pathname or LUNO is the pathname assigned to the linked output file during link edit, or the LUNO to which the linked output file has previously been assigned. The default responses to the remaining five prompts are NO.

Enter the IT command and follow the parameter input in paragraph H.2.3.1 except for the last parameter. To the question ATTACHED PROCEDURES respond YES. This response produces the additional following prompt messages:

```
                1st PROCEDURE ID:
P1 FROM TASKS PROGRAM FILE?:  YES
                2nd PROCEDURE ID:  0
P2 FROM TASKS PROGRAM FILE?:  YES
```

In response to these prompts enter the appropriate procedure ID and either a YES, to indicate that the procedure is in the same program file, or a NO to indicate that it is not.

H.2.3.3 Install Nonreentrant Task and Dummy Reentrant Procedure. Enter the IT command and follow the instructions as described in paragraphs H.2.3.1 and H.2.3.2.

H.2.3.4 Loading TX990 Standalone Programs. The FORTRAN TX990 programs may be on cassette, cards, or diskette and then loaded on the TX990 system. For information on loading refer to Appendix G, paragraph concerning PROGRAM LOAD AND EXECUTION.

H.2.3.5 Loading Standalone Programs. The FORTRAN standalone programs may be loaded by ROM cassette or card loader, or alternatively by a special user supplied loader. If the ROM loader is used, the object input must be on cassette or punched cards. If the linked object output from the link editor was not on punched cards or cassettes, the linked object file may be transferred to one of these mediums.

To load the standalone program, perform the following steps:

1. Place the object input device in the READY state.

2. Press HALT/SIE button on the front panel.

3. Press the RESET button.

4. Press the CLEAR button.

5. If punched cards are being loaded perform the following steps:

   a. Press the 8 switch.

   b. Press the MA ENTER switch.

   c. Press the MDE switch.

6. The default load address is $A0_{16}$. If a different load address is required perform the following steps:

   a. ENTER 92_{16} on the data switches.

   b. Press the MA ENTER switch.

   c. Enter the desired load address on the data switches.

   d. Press the MDE switch.

7. Press the LOAD switch.

The ROM loader then loads from the input device until a colon card is encountered.

H.2.4 I/O UNIT CORRESPONDENCE. After the task has been installed under DX10 each FORTRAN unit number used for input or output in the program must be associated with an actual I/O device.

H.2.4.1 DX10 With Synonym Assignment. In this environment, Logical Unit Numbers are assigned through synonyms. This command must be repeated for each LUNO used.

AS. The Assign Synonym Command (AS) may also be used. The form of the command is:

```
AS:
    SYNONYM: UNITX
    VALUE: LP01,ME,ETC.
```

H.2.4.2 DX10 With Global LUNO Assignment. In this environment, Logical Unit Numbers are globally or locally assigned. The form of the commands are:

```
AGL:
    LUNO:
    ACCESS NAME:
    PROGRAM FILE?:  NO
AL:
    LUNO:
    ACCESS NAME:
    PROGRAM FILE?:  NO
```

The user responds with the unit number used in the FORTRAN I/O statement and the appropriate DX10 device or pathname. This must be repeated for each LUNO used.

WARNING

> The special library for standalone programs does not support the FORTRAN I/O statements. Thus, there is no I/O unit correspondence for standalone programs. User support routines must be supplied if Operating System dependent statements are used in a standalone application.

H.2.4.3 **TX990 LUNO Assignment.** Assign LUNOs as described in Appendix G, paragraph concerning PROGRAM LOAD AND EXECUTION.

H.2.5 **PROGRAM EXECUTION.** Once the program I/O unit correspondence is established the program is ready for execution. The following paragraphs describe the steps necessary to execute the task.

H.2.5.1 **Program Execution Under DX10 Using Synonyms.** The command XFT (background) or XFTF (foreground) executes an installed FORTRAN program in this environment. The following prompt messages are displayed:

```
PROGRAM FILE LUNO:
          TASK ID:
```

The user must enter the LUNO (0 if done by the link editor) to which the program file is assigned and the task ID, which comes from the link map, of the FORTRAN installed task. The user may need to assign a global LUNO (AGL) to the program file before execution can begin.

H.2.5.2 **Program Execution Under DX10 With Global LUNOs.** The command XT executes a FORTRAN program in this environment. The following prompt messages are displayed:

```
PROGRAM FILE OR LUNO:
      TASK NAME OR ID:
              PARM 1: 0
              PARM 2: 0
          STATION ID?: ME
```

The user should respond with the program file in which the task was installed and with the task ID. The remaining messages should be skipped (enter RETURN for each), indicating default values.

H.2.5.3 **Program Execution Under TX990.** Program Execution is described in Appendix G, paragraph concerning PROGRAM LOAD AND EXECUTION.

H.2.5.4 **Executing Standalone Programs.** If a standalone program is loaded by the ROM loader, the loader automatically transfers execution control to the loaded program. If the user uses his own loader, he must specify the procedure for initiating execution. There are two ways that the program may be entered:

1. Word 0 of the program contains the initial workspace and word 1 contains the initial entry point. If the program is entered by setting the WP and PC to these values, the status register is not reset.

2. The linked object contains a different entry point address from that in 1 above (a 2 tag). If the program is entered at this entry point, the WP is automatically set to the initial workspace and the status register is reset.

H.2.6 BATCH FORTRAN EXECUTION. The steps described in the preceding paragraphs to execute a FORTRAN program may be combined into a batch stream of commands. The batch stream may be executed to accomplish all or part of the FORTRAN execution process. The following is an example of a batch stream that compiles, link edits, installs, and executes a FORTRAN program.

```
**
**   EXECUTE FORTRAN COMPILER
**        SF = SOURCE FILE
**        OB = OBJECT FILE
**        LF = LIST FILE
**        OP = OPTIONS
**        PW = PRINT WIDTH
XFC  SF = .FTNSRL, OB = .FTNOBJ, LF = .FTNLST, OP = 0, PW = 80
**
**   COPY OBJECT TO FILE TO USE IN LINK EDIT
**        IA = INPUT
**        OA = OUTPUT
**        RE = REPLACE
CC   IA =..FTNOBJ, OA = .FOBOBJ, RE = YES
**
**   EXECUTE LINK EDITOR
**        CF = CONTROL FILE
**        LO = LINKED OUTPUT
**        LF = LINKED LIST
**        PW = PRINT WIDTH
XLE  CF = .FTNCTL, LO = LNKOUT, LF = LNKLST, PW = 80
**
**   INSTALL TASK
**        PF = PROGRAM FILE
**        TN = TASK NAME
**        TID = TASK ID
**        OB = OBJECT
IT PF = S$PROGA,
   TN = FTNPRO,
   TID = 088
   OB = LNKOUT
**
**
**   EXECUTE THE FORTRAN PROGRAM
**        PL = PROGRAM FILE LUNO
**        ID = TASK ID
**
XFT PL = 0, ID = 088
**   DELETE TASK
**        P = PROGRAM FILE
**        TK = TASK ID
DT   P = 0, TK = 088
```

Refer to *The Model 990 Computer DX10 Operating System Reference Manual, Volume II, Production Operation* for more information describing how to build a batch stream.

H.2.7 ERROR LOGGING AND TERMINATION UNDER DX10. Errors and warnings detected by the FORTRAN runtime support package are written by an error logging routine on the terminal local file. Upon successful termination all user files are closed with an end-of-file before the program is terminated. The following message is displayed:

```
STOP 0
NORMAL PROGRAM COMPLETION
```

Termination due to a fatal error leaves all user files open for the operating system to close after program termination. This includes errors which cause the end action vector to be taken.

H.2.8 REPLACING STANDARD MODULES WITH USER-WRITTEN MODULES. Certain characteristics of FORTRAN programs such as the maximum number of unit numbers that can be used can be altered by the user by linking in modules he has written instead of the standard directory modules. Appendix E describes those modules a user may want to replace. The general form for the link edit control file for replacing standard modules is as follows:

```
NOSYMT
LIBRARY  user directory pathname
    .
    .
    .
other    directories
    .
    .
    .
PHASE 0, FORT
INCLUDE  name
END
```

The LIBRARY commands must be in the order shown. The user directory pathname parameter is the pathname of the directory that contains the user written modules. Other LIBRARY commands must be supplied for the standard directories and any special directory if appropriate.

It is preferable to allow the link editor to include user modules with a library search rather than to include them explicitly with an INCLUDE command. In certain situations using the INCLUDE command could cause the user supplied module as well as the standard module to be included in the link.

APPENDIX I

FORTRAN COMPILER AND RUNTIME ORGANIZATION

The following charts show the basic organizational structure of the FORTRAN package.

Figure I-1 shows the organization of the FORTRAN compiler according to function.

Figure I-2 shows the organization of the FORTRAN compiler by specific module. It gives the important functions each module performs.

Figure I-3 shows the organization of the FORTRAN runtime package. It depicts how the various functions are linked together.

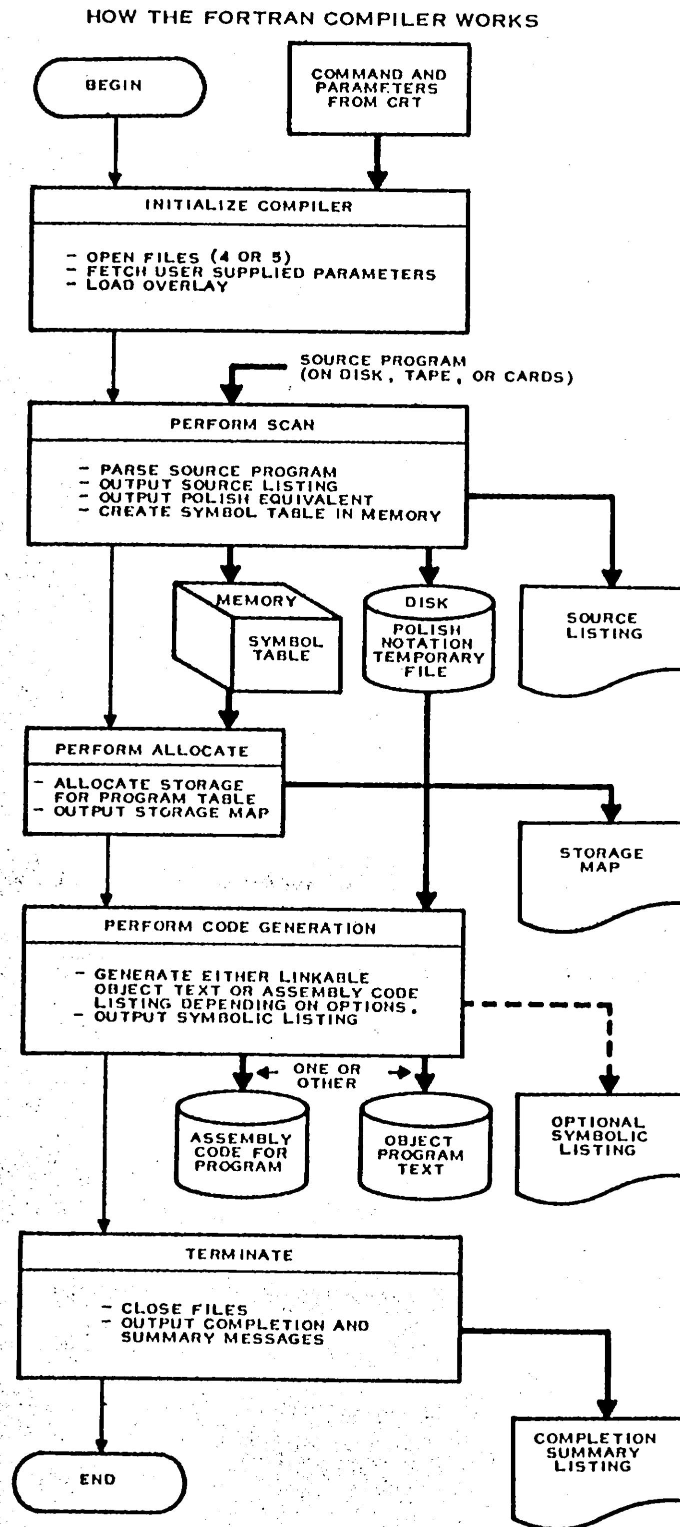


Figure I-1. How the FORTRAN Compiler Works

ROOT

F$ROOT

CONTROLS LOADING OVERLAYS, CALLING FPOP TO EXECUTE EACH OVERLAY, AND PERFORMS MOST I/O AND OTHER OS INTERFACING

FPOP

POPS INTERPRETER EXECUTES ONE OF THE THREE OVERLAYS PER CALL

OVERLAY1

SCAN

PARSES SOURCE PROGRAM, GENERATE SOURCE LISTING, POLISH EQUIVALENT OF SOURCE, AND SYMBOL TABLE

OVERLAY2

ALLOC

ALLOCATES STORAGE FOR PROGRAM VARIABLES, OUTPUTS STORAGE MAP AND ERROR MESSAGES DEALING WITH STORAGE PROBLEMS

OVERLAY3A

OGEN

GENERATES LINKABLE OBJECT PROGRAM TEXT FROM POLISH EQUIV. AND STORAGE ALLOCS. OUTPUTS SYMBOLIC LISTING

OVERLAY3B

AGEN

SAME AS OGEN EXCEPT ASSEMBLY LANGUAGE CODE IS GENERATED INSTEAD OF LINKABLE OBJECT TEXT

Figure I-2. Internal Organization of TI-990 DXS/3.0 FORTRAN IV Compiler

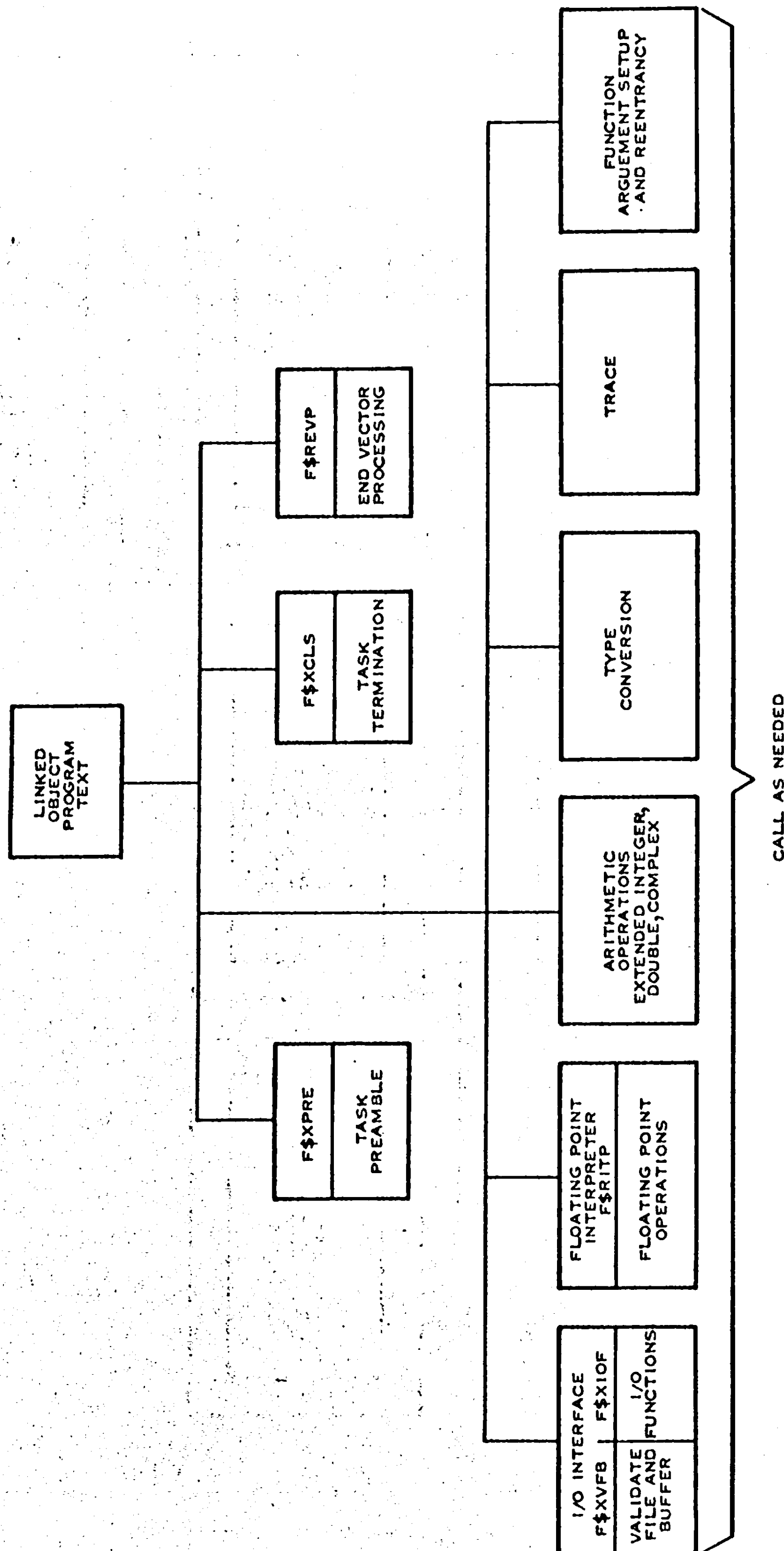


Figure I-3. TI990 DX10/3.0 FORTRAN Runtime Package

APPENDIX J

USAGE OF FIXED POINT NUMBERS

J.1 OVERVIEW

The advantage of fixed point numbers is that integer arithmetic is used rather that slower floating point arithmetic. The only penalty in using fixed point numbers is in the alignment of the operand binary points. The alignment is done through shift instructions so the penalty is not severe.Instruction sequences for fixed point are generated inline (except for division) in the same manner as integers so there is very little overhead involved in fixed point calculations. In fact, after any scaling adjustments have been performed, fixed point operations are the same as integer operations.

J.2 PROBLEM OF PRECISION

The primary difficulty in using fixed point numbers is that they have a widely varying scale factor but a fixed precision (a fixed number of digits). Integers have a precision of 16 binary digits. Fixed point numbers also have a precision of 16 binary digits. This corresponds to 4-1/2 decimal digits. Floating point numbers have fixed precision, but with 24 binary digits. This corresponds to 7-1/2 decimal digits. Besides having fewer significant digits, integer and fixed point numbers cover the number line in a different fashion than floating point numbers. For example, a variable declared:

```
INTEGER*2 IX
```

would cover the range:

```
-2**15 <= IX <= 2**15-1
```

On the number line, this is:

```
<-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+->
-32768                      0                       32767
```

Note every integer value is covered resulting in 65536 distinct values. Fixed point numbers have exactly the same number of values, but the range on the number line can be expanded or compressed. For example, a variable declared:

```
FIXED (-3) DOG
```

would cover the range:

```
-2**15/2**-3 <=DOG <= (2**15-1)/2**-3
```

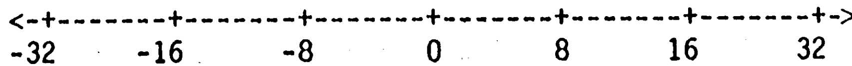
or:

```
-262144 <= DOG <= 262136
```

On the number line, this is:

```
 <-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+->
-262144                      0                        262136
```

and in expanded view:



Note that only every eighth integer is covered. The number of distinct values is the same as that for integers, but by scaling, holes have been introduced. Scaling with a positive value has the opposite effect. For example, a variable declared:

```
FIXED (3) CAT
```

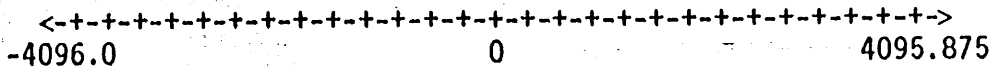
would cover the range:

```
-2**15/2**3  <= CAT  <= (2**15-1)/2**3
```

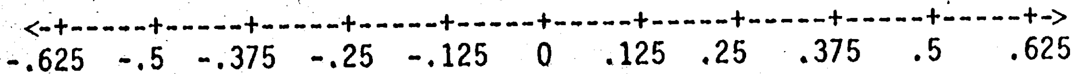
or:

```
-4096.0 <= CAT <= 4095.875
```

On the number line, this is:



and in expanded view:



Note that every integer in the range and seven equally spaced fractions are covered. However, the number of distinct values is unchanged; there are still only 65536 distinct values. This is the reason fixed point numbers present precision problems. Fixed point numbers only expand or compress the range of representable numbers; they do not change the number of representable values, i.e. increase precision over integer numbers.

To obtain more representable values, i.e. more precision, floating point is necessary. For example, a variable declared:

```
REAL*4 PIG
```

would cover the range:

```
-7.237 E 75 <= PIG <= -5.397 E -79
```

and

```
0
```

and

```
5.397 E -79 <= PIG <= 7.237 E 75
```

On the number line, this is represented as:

```
<-+================+---------+--------+=================+->
   -7.237E75       -5.397E-79   0   5.397E-79          7.23E75
```

and in expanded view:

```
<-+-------------+-------------+-------------+-------------+->
.5+2*16-6    .4999999404       5       .5000000596   .5+2*16**-6
```

Note that the range is larger than that of fixed point numbers and that the number of values is much larger. There are about 4.3 billion (2**32) distinct values resulting in much better precision. Also note there is a set of values about zero that are not representable, namely any X such that:

```
ABS(X) < 5.397 E -79,  X not equal zero.
```

J.3 FIXED POINT NUMBER OPERATING CHARACTERISTICS

The user of fixed point numbers must be aware of their operating characteristics since loss of precision is far easier than with other data types such as INTEGER or REAL. An example should demonstrate the dangers involved.

```
      FIXED (-7) A
      FIXED (4) B
      FIXED (0) C
      A = 45Q-7
      B = 27Q4
      C = A + B
      WRITE (6,8000) A,B,C
 8000 FORMAT (1X,3F20.8)
      END
```

Execution of this program results in:

```
A=5760.00000000          B=1.68750000          C=1665.000000
```

At first glance, the above value of C appears to be incorrect. The expected value for C is 5761.687500, which is the value produced under floating point arithmetic. Actually, what has happened is the most significant bit was lost. Such a loss results from alignment of fixed point numbers before arithmetic operations. The FORTRAN compiler generates a command to shift the contents of A eleven bits to the left and to align the binary point of A and B before generating the add A and B command. This shift command loses the most significant bit of A. The following illustrates the process:

```
Original value of A:  0000 0000 0010 1101
 Shifted value of A:  0110 1000 0000 0000
Original value of B:  0000 0000 0001 1011
 Result of addition:  0110 1000 0001 1011
```

By converting the result from binary to decimal, the correct answer is indeed 1665.6875. To avoid such a loss of precision, table J-1 gives the smallest and largest magnitude numbers representable with a given scale factor.

Table J-1. Fixed Point Scale Factor Vs. Decimal Magnitude

Scale	Smallest Magnitude Number	Largest Decimal Number
Q 31	.4656612873D-09	.152583234012127D-04
Q 30	.9313225746D-09	.305166468024254D-04
Q 29	.1862645149D-08	.610332936048508D-04
Q 28	.3725290298D-08	.122066587209702D-03
Q 27	.7450580597D-08	.244133174419403D-03
Q 26	.1490116119D-07	.488266348838806D-03
Q 25	.2980232239D-07	.976532697677612D-03
Q 24	.5960464478D-07	.195306539535522D-02
Q 23	.1192092896D-06	.390613079071045D-02
Q 22	.2384185791D-06	.781226158142090D-02
Q 21	.4768371582D-06	.156245231628418D-01
Q 20	.9536743164D-06	.312490463256836D-01
Q 19	.1907348633D-05	.624980926513672D-01
Q 18	.3814697266D-05	.124996185302734D 00
Q 17	.7629394531D-05	.249992370605469D 00
Q 16	.1525878906D-04	.499984741210937D 00
Q 15	.3051757812D-04	.999969482421875D 00
Q 14	.6103515625D-04	.199993896484375D 01
Q 13	.1220703125D-03	.399987792968750D 01
Q 12	.2441406250D-03	.799975585937500D 01
Q 11	.4882812500D-03	.159995117187500D 02
Q 10	.9765625000D-03	.319990234375000D 02
Q 9	.1953125000D-02	.639980468750000D 02
Q 8	.3906250000D-02	.127996093750000D 03
Q 7	.7812500000D-02	.255992187500000D 03
Q 6	.1562500000D-01	.511984375000000D 03
Q 5	.3125000000D-01	.102396875000000D 04
Q 4	.6250000000D-01	.204793750000000D 04
Q 3	.1250000000D 00	.409587500000000D 04
Q 2	.2500000000D 00	.819175000000000D 04
Q 1	.5000000000D 00	.163835000000000D 05
Q 0	.1000000000D 01	.327670000000000D 05
Q -1	.2000000000D 01	.655340000000000D 05
Q -2	.4000000000D 01	.131068000000000D 06
Q -3	.8000000000D 01	.262136000000000D 06
Q -4	.1600000000D 02	.524272000000000D 06
Q -5	.3200000000D 02	.104854400000000D 07
Q -6	.6400000000D 02	.209708800000000D 07
Q -7	.1280000000D 03	.419417600000000D 07
Q -8	.2560000000D 03	.838835200000000D 07
Q -9	.5120000000D 03	.167767040000000D 08
Q-10	.1024000000D 04	.335534080000000D 08
Q-11	.2048000000D 04	.671068160000000D 08
Q-12	.4096000000D 04	.134213632000000D 09
Q-13	.8192000000D 04	.268427264000000D 09
Q-14	.1638400000D 05	.536854528000000D 09
Q-15	.3276800000D 05	.107370905600000D 10
Q-16	.6553600000D 05	.214741811200000D 10
Q-17	.1310720000D 06	.429483622400000D 10
Q-18	.2621440000D 06	.858967244800000D 10
Q-19	.5242880000D 06	.171793448960000D 11
Q-20	.1048576000D 07	.343586897920000D 11
Q-21	.2097152000D 07	.687173795840000D 11
Q-22	.4194304000D 07	.137434759168000D 12
Q-23	.8388608000D 07	.274869518336000D 12
Q-24	.1677721600D 08	.549739036672000D 12
Q-25	.3355443200D 08	.109947807334400D 13
Q-26	.6710886400D 08	.219895614668800D 13
Q-27	.1342177280D 09	.439791229337600D 13
Q-28	.2684354560D 09	.879582458675200D 13
Q-29	.5368709120D 09	.175916491735040D 14
Q-30	.1073741824D 10	.351832983470080D 14
Q-31	.2147483648D 10	.703665966940160D 14

J.3.1 **ASSIGNMENT STATEMENT.** Given the assignment statement:

```
X = Y
```

where:

X   has a scale factor of sx, and

Y   has a scale factor of sy.

Then if sx = sy, no shifting precedes the MOV command which stores the integer portion of Y in X.

Or, if sx < sy, a SRA of Y sy-sx bit places precedes the MOV. This shift loses sy-sx of the least significant bits.

Or, if sx > sy, a SLA of Y sx-sy bit places precedes the MOV. This shift loses sx-sy bits from the most significant bits. If these bits were leading zeros, no information is lost.

A point to remember in each of these cases is that the resulting value retains the scale factor of sx. An example of a common source of precision loss is:

```
FIXED (0) X
X = 54Q3
```

Since the scale of the right-hand side (3) is greater than the scale of the left-hand side (0), the right-hand side is right shifted three bits. Unfortunately, this results in loss of precision since 54 (110110 in binary) right shifted three bits is 6 (110 in binary).

To avoid problems with assignment statements, be sure to scale the receiving item with a scale factor equal or larger than the scale factor of the source item.

J.3.2 **ADDITION AND SUBTRACTION.** Addition and subtraction are treated identically for fixed point operations so all references below to addition apply equally well to subtraction. Given the expression:

```
X + Y
```

where:

X   is FIXED (sx), and

Y   is FIXED (sy),

then, if sx = sy, no operations precede the normal addition.

Else, the operand whose scale factor is smaller is left shifted, end off (SLA) |sx-sy| bit places. This aligns the X and Y binary points, but loses sx-sy leading bits from the shifted operand. The scale factor of the result is the MAX(sx,sy).

J.3.3 MULTIPLICATION. Multiplication of two fixed point numbers requires no shifting before the normal integer multiplication. The only source of precision loss is a product overflowing 15 binary digits. The scale factor of a product is the sum of the scale factors of the operands.

J.3.4 DIVISION. Division has the same restrictions as multiplication. The only difference is that the resulting scale factor is the difference of the operand scale factors.

J.4 SCALING TO AVOID LOSS OF PRECISION

The fixed point number iQs represents the number i/2**s. This is analogous to the floating point number jEt representing the number j*10**t. The difference is that the fixed point number is divided by the scale factor while the floating number is multiplied by the scale factor. This is the most important point to remember in scaling fixed point numbers.

There are several guidelines to follow in scaling fixed point numbers to avoid loss of precision. First, try to keep the scale factors of the right-hand and left-hand sides of assignments nearly equal. For example, as the results of a calculation grow larger, scale temporaries up to larger values; if they grow smaller, scale temporaries down. Second, avoid using the shift functions in fixed point expressions. Third, analyze and arrange expressions so intermediate values have nearly the same scale factors.

ALPHABETICAL INDEX

INTRODUCTION

HOW TO USE THE INDEX

The index, table of contents, list of illustrations, and list of tables are used in conjunction to obtain the location of the desired subject. Once the subject or topic has been located in the index, use the appropriate paragraph number, figure number, or table number to obtain the corresponding page number from the table of contents, list of illustrations, or list of tables. The table of contents does not contain four-level paragraph entries. Therefore, for four-level paragraph numbers such as 2.3.1.2, use the three-level number and the corresponding page number. In this case, the three-level number is 2.3.1.

INDEX ENTRIES

The following index lists key words and concepts from the subject material of the manual together with the area(s) in the manual that supply major coverage of the listed concept. The numbers along the right side of the listing reference the following manual areas:

- Sections - References to Sections of the manual appear as "Section x" with the symbol x representing any numeric quantity.
- Appendixes - References to Appendixes of the manual appear as "Appendix y" with the symbol y representing any capital letter.
- Paragraphs - References to paragraphs of the manual appear as a series of alphanumeric or numeric characters punctuated with decimal points. Only the first character of the string may be a letter; all subsequent characters are numbers. The first character refers to the section or appendix of the manual in which the paragraph is found.
- Tables - References to tables in the manual are represented by the capital letter T followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the table). The second character is followed by a dash (-) and a number:

  Tx-yy

- Figures - References to figures in the manual are represented by the capital letter F followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the figure). The second character is followed by a dash (-) and a number:

  Fx-yy

- Other entries in the Index - References to other entries in the index are preceded by the word "See"followed by the referenced entry.

The index is divided into sections for the letters of the alphabet. Acronyms and mnemonics (words made up entirely of capital letters) are listed first within each section. Words that begin with a capital letter follow the acronyms and mnemonics.

A Field Specification 5.2.3.2
ABS, Intrinsic Function Format T7-2
Absolute Address 7.4.4
AB1 Error Code 7.3.10, 7.4.3
Accept Statements 5.7, 5.7.1
Accept/Display E.4.4
Access 7.3.12, 7.3.15
Access Name:
  Listing H.2.1.3
  Object H.2.1.2
  Source H.2.1.1
ADATE CALL Subroutine 7.4.10
ADDFRC Double Precision Fraction
  Addition C.5
Addition J.3.2
  F$XAD Double Precision C.5
  F$XAR Single Precision C.5
  Positive Sign T3-1
  Single Precision Floating Point C.4
Address, Absolute 7.4.4
AIMAG, Intrinsic Function Format T7-2
AINT, Intrinsic Function Format T7-2
AIRDW, Subroutine 7.3.9
AISQW, Subroutine 7.3.9
Allocation 7.3.14
  Common 9.2
  Equivalence 9.2
  Program 9.2
ALOG, Basic External Function Format T7-1
ALOG10, Basic External Function
  Format T7-1
Alphanumeric Data Specifications 5.2.3
ALSQW, Subroutine 7.3.9
AMAX0, Intrinsic Function Format T7-2
AMAX1, Intrinsic Function Format T7-2
AMIN0, Intrinsic Function Format T7-2
AMIN1, Intrinsic Function Format T7-2
AMOD, Intrinsic Function Format T7-2
Analog:
  Data Handling 7.3.9
  Input in Random 7.3.9
  Input in Sequence 7.3.9
  Output 7.3.9
ANSI Extensions 1.1
AOW Subroutine 7.3.9
Arithmetic:
  Data, Conversion of 5.2.1
  Definitions 1.2.5
  Expressions 3.2
    Evaluation of 3.2.3
    Formation of 3.2.2
  Floating Point . . . . . Appendix C
  IF Statement 4.3.1
  Operations 3.2.1
  Statement Function 6.3
Array:
  Elements, Storage of 2.6.2
  Example T2-1
  Variables 1.2.5
Arrays and Subscripts 2.6.1
ASCII Date, Obtain 7.4.10
ASCII Time, Obtain 7.4.11
Assembling B.6.3
Assembly Language:
  Program FB-3, FB-7
  Source Code H.2.1.4
  Subroutine FB-6
  Subroutine Test FB-2
Assign, Statement 4.2, 4.2.3
Assigned GO TO Statements 4.2, 4.2.3
Assigning LUNOs to VDTs on TXDS G.5
Assignment:
  DX10 with Global LUNO H.2.4.2
  Statement 3.4.2, J.3.1, T3-2
ATAN, Basic External Function Format T7-1
ATAN2, Basic External Function
  Format T7-1
ATIME, Subroutine 7.4.11
Auto Abort Flag 7.4.3

Backspace, Statements 5.1, 5.5.2
Basic External Function Format T7-1
Basic External Functions, FORTRAN
  Library T7-1
Basic Logical Element 3.3.1
Batch FORTRAN Execution H.2.6
Bid Task 7.4.5
  BIDTSK, Subroutine 7.4.5
  Delayed 7.4.6
BIDTSK, Subroutine 7.4.5
Blank Common 2.7.1
Block:
  Data, Statement 6.6, 6.6.2
  Register 7.4.3
  SVC 7.4.3
Branch Point 4.4.2
BUF 7.3.13
Buffer Size, I/O E.3.2
BUFIN Subroutine 7.2.1
BUFOUT Subroutine 7.2.2, E.4.3

C G.2.1.3, H.2.1.4
  Option TH-1, G.2.1.3
CABS, Basic External Function Format T7-1
CALL:
  ADATE 7.4.10
  AISQW 7.3.9
  AOW 7.3.9
  ATIME 7.4.11
  BIDTSK 7.4.5
  BUFIN 7.2.1
  CFILW 7.3.10
  CLOSEW 7.3.12
  DATIME 7.4.9
  DFILW 7.3.11
  DIW 7.3.4
  DLYBID 7.4.6
  DOLW 7.3.5
  DOMW 7.3.6
  IMGBLD 7.5.3.1
  MDATE 7.4.12
  MODAPW 7.3.15
  OCRU 7.4.8
  OPENW 7.3.12
  RANSET 7.4.2
  RDPROM 7.5.3.3

RDRW 7.3.13
START 7.3.1
STATEMENT 4.5.1
SUB B.2
SVC 7.4.3
SVCFUT 7.3.14
TIME 7.3.8
TRNON 7.3.2
WAIT 7.3.3
WRTRW 7.3.13
Callable:
MACROS, User B.5.1
Subroutines, FORTRAN 7.4
Calling Sequence, READ/WRITE E.4.1
Carriage Control 5.2.3.1
CCOS, Basic External Function Format T7-1
CEXP, Basic External Function Format T7-1
CFILE 7.3.10
CFILW Subroutine 7.3.10
CFILW Parameters 7.3.10
Character Set TA-1
FORTRAN 1.5
Source Program ....Appendix A
Character String Manipulation ....Appendix D
Characteristics:
Operating E.2
Logical Record 5.8
Physical Record 5.8
CLOG, Basic External Function Format T7-1
CLOSEW:
CALL 7.3.12
Parameters 7.3.12
Closing Parenthesis 5.2.6
CLOSMT 7.6.2
CLS32I 7.7.2
CMPLX, Intrinsic Function Format T7-2
Code:
AB1 Error 7.3.10
Object 1.1, 9.4
Source 1.1
Combinations of Formats 5.2.8
Comma, Separating 5.2.6
Comments 1.2.3
Common:
Allocation 9.2
And Equivalence Statements 2.7
Blank 2.7.1
Entries, Redundant 2.7.1
Equivalence Interaction 2.7.3
Statements 2.7, 2.7.1
Variables 1.2.5
Compilation:
Program H.2.1
TXDS Program G.2.1
Compile and Link Edit, TXDS G.3.1
Compiler:
Output, FORTRAN ....Section IX
Program Error Diagnostic Messages T9-2
Selectable Options, FORTRAN TH-1
Statement Error Diagnostic Messages T9-1
TXDS F-1
Complex T3-2
Numbers 2.4.4
Operands 3.3.1
Qualities 5.2.4
Statement 2.3.4
Type Statement 2.3
Computed GO TO Statement 4.2, 4.2.2
Conditional Compilation G.2.1.3, H.2.1.4
CONJ6, Intrinsic Function Format T7-2
Consecutive Slashes 5.2.6
Console Display Input/Output 5.7
Constant(s) 2.2, 2.2.3
Hollerith 2.5
Unique:
Extended Complex 1.2.5
Extended Double Precision 1.2.5
Extended Integer 1.2.5
Extended Real 1.2.5
Continuation:
Indicator T1-1
of a Program, Delay 7.3.3
Continue Statement(s) 1.3, 4.4, 4.2.2
Control:
Carriage 5.2.3.1
File Defaults, TXDS G.2.2.1
Transfer of 4.5
Conversion:
Module:
Floating Point to Integer C.3
Integer to Floating Point C.2
of Arithmetic Data 5.2.1
to Higher Rank 3.2.3
Type T3-2
Copy Statement 1.3
Correspondence, I/O Unit H.2.4
COS, Basic External Function Format T7-1
COSH, Basic External Function Format T7-1
Creating the MACROs 8.5
Cross-Reference Listing, Variable FH-2
CRU:
Functions, OS Dependent, I/O Except TE-1
Input 7.4.7
Output 7.4.8
CSIN, Basic External Function Format T7-1
CSQRT, Basic External Function Format T7-1

D G.2.1.3, H.2.1.4
Field Specification 5.2.1, 5.2.1.5
Option G.92.1.3, TH-1
DABS, Intrinsic Function Format T7-2
DATA:
Flow, IMGBLD F7-1
Handling, Analog 7.3.9
Initialization, Specification Subprograms for 6.6
Specifications ....Section II
Alphanumeric 5.2.3
Block Statement 6.6, 6.6.2
Literal 5.2.3
Numerical 5.2.1
Statement 6.6, 6.6.1
Types 2.4

DATAN, Basic External Function
  Format T7-1
DATAN2, Basic External Function
  Format T7-1
Date and Time, Obtain 7.4.12
Date, Obtain Military 7.4.12
DATIME:
  Call 7.4.9
  Subroutine 7.4.9
DBLE, Intrinsic Function Format T7-2
DCOS, Basic External Function Format T7-1
Debug Trace Compilation G.2.1.3, H.2.1.4
Decode Statement 5.4.2
Defaults:
  TXDS:
    Control Field G.2.2.1
    Listing File G.2.1.2
    Object File 2.2.2, G.2.1.2
    Source File G.2.1.1
Define File Statement 5.6.1
Definitions:
  of Terms 2.2
  Subprogram 6.2.1
Delay Continuation of a Program 7.3.3
Delayed Bid Task 7.4.6
Dependent: Functions OS TE-1
  I/O Except CRU Functions OS TE-1
  Initialization Functions OS TE-1
  ISA:
    Extensions Functions OS TE-1
    File Contention Functions OS TE-1
  OS Error Logging Functions TE-1
  Termination Functions OS TE-1
Description, FORTRAN
  Runtime ....Appendix E
DEXP, Basic External Function Format T7-1
DFILE 7.3.11
DFILW:
  Call 7.3.11
  Parameters 7.3.11
  Subroutine 7.3.11
Diagnostic Messages:
  Compiler Program Error T9-2
  Compiler Statement Error T9-1
  Runtime Package T9-3
Diagnostics, Error 9.3
Digital Input 7.3.4
Digital Output:
  Latched 7.3.5
  Momentary 7.3.6
DIM, Intrinsic Functional Format T7-2
Dimension Statement 2.6.3
Dimensioned Variables 2.6
Dimensions Dummy 6.4
Direct Access:
  File Definition E.5.2
  Input/Output 5.6
Directory:
  DXLOBJ E.2.2
  Stand-alone E.2.4
  TXLOBJ E.2.3
Display, Statements 5.7, 5.7.2
Division J.3.4, T3-1
  F$XDD Double Precision C.9
  F$XDR Single Precision C.8
  Module:
    Double Precision C.9
    Single Precision C.8
DIW:
  Parameters 7.3.4
  Subroutine 7.3.4
DLOG, Basic External Function Format T7-1
DLOG10, Basic External Function
  Format T7-1
DLYBID:
  CALL 7.4.6
  Subroutine 7.4.6
DMAX0, Intrinsic Function Format T7-2
DMIN1, Intrinsic Function Format T7-2
DMOD, Basic External Function
  Format T7-1
DO:
  Example 4.4.1
  Extended Range 4.4.1
  Inner 4.4.1
  LOOP 4.4.1
  Outer Statement 4.4, 4.4.1
  Transfer Into 4.4.1
  Transfer Out 4.4.1
DOLW:
  Call 7.3.5
  Parameters 7.3.5
  Subroutine 7.3.5
DOMW Subroutine 7.3.6
Double Precision T3-2
  Addition:
    F$XAD C.5
    Module, Subtraction, and Fraction
      Addition C.5
  Division:
    F$XDD C.9
    Module C.9
  F$XCED Extended Integer to C.2
  F$SCID Integer to C.2
  Floating Point Multiplication
    Module C.7
  Fraction Addition, ADDFRC C.5
  Multiplication, F$XMD C.7
  Numbers, Examples 2.4.3.1
  Statement 2.3.3
  Subtraction, F$XSD C.5
  To:
    Extended Integer, F$SCDE C.3
    Integer, F$SCDI C.3
  Type Statement 2.3
DSIGN, Intrinsic Function Format T7-2
DSIN, Basic External Function Format T7-1
DSQRT, Basic External Function
  Format T7-1
Dummy:
  Dimensions 6.4
  Identifiers 6.2.2
  Reentrant Procedure, Install
    Nonreentrant Task and H.2.3.3

DXLOBJ Directory E.2.2
DX10:
Error Logging and Termination
Under H.2.7
with:
Global LUNO Assignment H.2.4.2
Synonym Assignment H.2.4.1
3.X:
Standard Directories E.2.1
System, FORTRAN Execution
on a ....Appendix H

E: Field Specification 5.2.1, 5.2.1.3
Edit Program Link B.4
Editor, TXDS Link F.2
Encode Statement 5.4.1
End:
File Statements 5.1, 5.5.3
Line 1.2.2
Statement 1.2.2, 4.6, 4.6.3
Vector Processing.....................E.3.4
EPROM Example F7-2, F7-3
EQ 3.3.1
Equivalence Statements, Common and 2.7
Equal to 3.3.1
Equivalence 2.7.2
Allocation...............................9.2
Interaction Common 2.7.3
Statement 2.7, 2.7.2
Equivalenced Variable Names 1.2.5
Error:
Code, AB1 ....7.3.10
Diagnostic 9.3
Program 9.3.2
Runtime 9.3.3
Statement 9.3.1
Diagnostic Messages:
Compiler Program T9-2
Compiler Statement T9-1
Logging:
and Termination of TXDS.............G.4
and Termination under DX10 ....H.2.7
Function OS Dependent..............TE-1
Message and E.3.5
Messages 9.2, 9.3
Runtime T9-4
Table, IERR.............................T7-3
Evaluation of:
Arithmetic Expressions 3.2.3
Logical Expressions 3.3.4
Examples:
Array...................................T2-1
DO 4.4.1
Double Precision Numbers...........2.4.3.1
EPROM F7-2, F7-3
Fixed Point Integer 2.4.6.3
PROM...................................F7-4
Real and Relocate F7-5
Real Numbers 2.4.2.1
Executing Stand-alone
Programs...................G.3.3, H.2.5.3

Execution:
for Global LUNOs, Linking or........H.2.2.2
Program.......................H.2, H.2.5
TXDS Program Load and G.2.3
EXP, Basic External Function Format T7-1
Exponentiation T3-1
Expression(s) 3.4.2
Arithmetic...............................3.2
Evaluation of Arithmetic 3.2.3
Formation of Arithmetic 3.2.2
FORTRAN 3.1
Logical...................................3.3
Summary of Rules for 3.4
Extended:
Complex Constants, Unique 1.2.5
Double Precision Constants,
Unique 1.2.5
Constants, Unique 1.2.5
F$XCDE Double Precision to C.3
F$XCRE Single Precision to C.3
to Double Precision, F$XCED C.2
to Single Precision, F$XCER...........C.2
Range DO Loop 4.4.1
Real Constants, Unique 1.2.5
Extensions:
ANSI 1.1
Functions OS Dependent, ISA TE-1
ISA.......................................7.3
External:
Representation.....................2.4.6.1
Statement........................2.3, 6.7

F G.2.1.3, H.2.1.4
Field Specification 5.2.1, 5.2.1.2
Option G.2.1.3, TH-1
F$XAD Double Precision Addition C.5
F$XAR Single Precision Addition C.4
F$XCDE Double Precision to Extended
Integer C.3
F$XCDI Double Precision to Integer C.3
F$XCED Extended Integer to Double
Precision.................................C.2
F$XCER Extended Integer to Single
Precision.................................C.2
F$CID Integer to Double Precision..........C.2
F$XCIR Integer to Single Precision C.2
F$XCRE Single Precision to Extended
Integer C.3
F$XCRI Single Precision to Integer C.3
F$XDD Double Precision Division C.9
F$XDR Single Precision Division C.8
F$XMD Double Precision
Multiplication C.7
F$XMR Single Precision
Multiplication C.6
F$XSD Double Precision Subtraction C.5
F$XSR Single Precision Subtraction.........C.4
Factors, Field Scale 5.2.1.7
Field(s) T1-1
Hollerith 2.5
Identification T1-1

Field(s) (Continued)
in a Source Record 1.2.1
Scale Factors 5.2.1.7
Specification(s)
A 5.2.3.2
D 5.2.1, 5.2.1.5
E 5.2.1, 5.2.1.3
F 5.2.1, 5.2.1.2
G 5.2.1, 5.2.1.4
H 5.2.3.3
I 5.2.1, 5.2.1.1
Logical Data 5.2.2
Repeated Group and 5.2.5
Repetition of 5.2.5.1
T 5.2.3.5
X 5.2.3.4
Z 5.2.1, 5.2.1.6
Statement T1-1
FILANDISC 7.3.14
File:
Characteristics:
Read Sequential 5.3.1
Write Sequential 5.3.1
Contention Functions TE-1
Defaults:
TXDS Control G.2.2.1
TXDS Listing G.2.1.2
TXDS Object G.2.1.2, G.2.2.2
Definition, Direct Access E.4.2
Input/Output Statements, Mass
Storage 5.1, 5.5
Statements, End 5.1
TXDS Word G.2.2.2
Filename 7.3.12
Fixed T3-2
Fixed Point:
Integer Examples 2.4.6.3
Integers 2.4.6
Numbers .... Appendix J
Precision J.2
Scale Factor Decimal Magnitude TJ-1
Fixed:
Statement 2.3.6
Type Statement 2.3
Flag, Auto Abort 7.4.3
Float, Intrinsic Function Format T7-2
Floating Point:
Addition, Single Precision C.4
Arithmetic .... Appendix C
Multiplication Module:
Double Precision C.7
Single Precision C.6
to Integer Conversion Module C.3
Format:
Free Field 5.2.9
Sample, Object Option FH-1
Statement 5.1, 5.2
Formation of:
Arithmetic Expressions 3.2.2
Logical Expressions 3.3.3

Formats:
Combinations of 5.2.8
Stored as Data 5.2.7
Formatted:
Direct Access Read/Write E.4.1.3
Read Statement 5.3.4
Read/Write E.4.1.1
Record 5.3.2
Write Statement 5.3.5
FORTRAN:
Callable Subroutines 7.4
Character Set 1.5
Compiler:
and Runtime Organization .... Appendix I
Operation FI-1
Output .... Section IX
Selectable Option TH-1
Directories, Introduction to E.1
Execution:
Batch H.2.6
on DX10 3.X System .... Appendix H
Expressions 3.1
Installation on TXDS .... Appendix F
Library:
Basic External Functions T7-1
Functions 7.1
Intrinsic Functions T7-2
Program FB-1, FB-5
Link Edit Installation of H.2.2.4
Routine, User 7.5.3
Runtime Description .... Appendix E
Runtime Package FI-3
Standalone G.3
Statements 1.2.2
Subroutine FB-4, FB-8
Subscripts Permissible 2.6.1
Unit Numbers 5.9
Verification, TXDS F.3
FORTRAN-PROM, Limitations 7.5.1
FORTRAN-PROM Program Subroutines 7.5
FORTRAN IV Compiler, Internal
Organization FI-2
FPMDAT MACRO B.5.2.1
FPMEQU MACRO B.5.2.4
FPMGEN MACRO B.5.2.5
Fraction Addition:
ADDFRC Double Precision C.5
Double Precision Addition Module,
Subtraction and C.5
Free Field Format 5.2.9
Free Format G.2.1.3, H.2.1.4
FTNCAL FB-3
FTNCAL MACRO B.6.1.3
FTNCAL MACRO B.5.1.1
FTNINT MACRO B.5.1.2, B.6.1.1, FB-3
FTNRET MACRO B.5.1.3, B.6.2.3, FB-6
FTNSUB MACRO B.5.1.4, B.6.1.2
Function:
Arithmetic Statement 1.2.5
IVERFY D.5

KOMSTR D.3
Length D.7
Mathematical 2.8
MFLD D.6
OS Dependent TE-1
Error Logging TE-1
I/O Except CRU TE-1
INDEX D.4
Initialization TE-1
ISA Extensions TE-1
ISA File Contention TE-1
Termination TE-1
Reference 2.8
Statement 6.3
Subprogram 6.4
SUBSTR D.2

G: Field Specification 5.2.1, 5.2.1.4
GE 3.3.1
Generate Supervisor Call 7.4.3
Global:
Common Linking for H.2.2.1
LUNO:
Assignment, DX10 with H.2.4.2
Linking for Execution for H.2.2.2
Program Execution with H.2.5.2
GO TO Statement 4.2, 4.2.3
Assigned 4.2
Computed 4.2, 4.2.2
Unconditional 4.2, 4.2.1
Greater Than 3.3.1
Greater Than or Equal to 3.3.1
GT 3.3.1

H: Field Specification 5.2.3.3
Hexadecimal Constants 2.4.1
Hollerith:
Constant 2.5, 3.4.2
Field 2.5

I: Field Specification 5.2.1, 5.2.1.1
I/O:
Buffer Size E.3.2
List Element, Handlers E.4.1.5
Statements, Mass Storage File E.4.5
Unit Correspondence H.2.4
Units Allowed E.3.1
IBCLR, Basic External Function Format T7-1
IBSET, Basic External Function Format T7-1
IBTEST, Basic External Function Format ...T7-1
ICRU, Subroutine 7.4.7
Identification Field T1-1
Identifiers 2.2, 2.2.1
Dummy 6.2.2
IDIM, Intrinsic Function Format ...T7-2
IDINT, Intrinsic Function Format ..T7-2
IOER, Basic External Function Format T7-1
IERR, Error Table T7-1

IF Statement 4.3
Arithmetic 4.3.1
Logical 4.3.2
IFIX, Intrinsic Function Format ...T7-2
IMGBLD:
Data Flow F7-1
Module 7.5.2.1
Implicit Statement 2.3.7
Increment 4.4.1
INDEX, Function D.4
Index 4.4.1
Indicator, Continuation T1-1
Information, Runtime Error Traceback 9.5
Initial Value 4.4.1
Initialization Functions OS Dependent TE-1
Inner DO 4.4.1
Input:
CRU 7.4.7
Digital 7.3.4
in Random, Analog 7.3.9
in Sequence, Analog 7.3.9
Lists, Read and Write 5.3.3
TXDS G.2.1.1, G.2.2.1
Input/Output:
Console Display 5.7
Direct Access 5.6
Statements, Mass Storage File 5.1, 5.5
Inputting the PROM Program from Mass Storage 7.5.3.1
Install Nonreentrant Task and:
Dummy Reentrant Procedure H.2.3.3
Reentrant Procedure H.2.3.2
Install Single Nonreentrant Task ..H.2.3.1
Installation, Program H.2.3
INT, Intrinsic Function Format T7-2
Integer:
Constants, Unique 1.2.5
Conversion Module, Floating Point to C.3
Examples, Fixed Point 2.4.6.3
Fixed Point 2.4.6
F$XCDI Double Precision to C.3
F$XCRI Single Precision to C.3
Statement 2.3.1
to Double Precision, F$XCID C.2
to Floating Point Conversion Module C.2
to Single Precision, F$XCIR C.2
Type Statement 2.3
Integer*2 ....T3-2
Integer*4 ....T3-2
Interaction Common, Equivalence ...2.7.3
Interface E.4
Internal:
Organization FORTRAN IV Compiler ..FI-2
Representation 2.4.6.2
Transmission 5.4
Intrinsic Function:
Format T7-2
FORTRAN Library T7-2
Introduction to FORTRAN Directories E.1

IOR, Basic External Function Format T7-1
ISA:
Extensions 7.3
Functions OS Dependent TE-1
File Contention Functions OS
Dependent TE-1
ISHFT, Basic External Function
Format..............................T-7-1
ISIGN, Intrinsic Function Format T7-2
IUNIT, Basic External Function Format T7-1
IVERFY, Function D.5

KOMSTR, Function D.3

LABS, Intrinsic Function Format T7-2
LAND, Basic External Function Format T7-1
Latched Digital Output 7.3.5
LDIM, Intrinsic Function Format..........T7-2
LE......................................3.3.1
LENGTH, Function D.7
Less Than..............................3.3.1
Less Than or Equal To 3.3.1
Level One..............................5.2.6
Level Zero 5.2.6
LFIX, Intrinsic Function Format T7-2
LFLOAT, Intrinsic Function Format T7-2
Library:
Basic External Functions,
FORTRAN..........................T7-1
FORTRAN ....Section VII
Functions, FORTRAN 7-1
Intrinsic Functions, FORTRAN T7-2
Subroutines 7.2
Limit, Range 4.4.1
Limitations FORTRAN-PROM 7.5.1
Line, End 1.2.2
Link:
Control File for Linking a Task and
a Procedure FH-3
Edit:
Installation of FORTRAN
Program H.2.2.4
Program B.4, H.2.2
TXDS, Compile and G.3.1
TXDS Program G.2.2
Editor, TXDS...........................F.2
Linking a Task and a Procedure,
Link Control File for FH-3
Linking for:
Execution for Global LUNOs.........H.2.2.2
Execution with Synonym
Assignment H.2.2.1
Global Common...................H.2.2.1
Shared Data Block H.2.2.1
Stand-alone Execution...............H.2.2.3
Linking with User-Supplied Modules E.3.7
LINT, Intrinsic Function Format T7-2
List Element Handlers, I/O E.4.1.5
Listing:
Access Name........................H.2.1.3
Elements, Output 9.2
File Defaults, TXDS G.2.1.2
Output, TXDS G.2.1.2

Lists, Read and Write Input
and Output 5.3.3
Literal Data Specifications 5.2.3
LMAX0, Intrinsic Function Format T7-2
LMAX1, Intrinsic Function Format T7-2
LMIN0, Intrinsic Function Format T7-2
LMIN1, Intrinsic Function Format T7-2
LMOD, Intrinsic Function Format T7-2
Load:
Map Output, TXDS G.2.2.2
Stand-alone Program Procedure G.3.2
Loading Stand-alone Programs ...G.3.2, H.2.3.4
LOC, Subroutine 7.4.4
Logging:
and Termination of TXDS, Error G.4
Functions OS Dependent, Error TE-1
Logical:
AND 3.3.2
Constants, Unique 1.2.5
Data Field Specification 5.2.2
Dot 3.3.2
Element, Basic........................3.3.1
Expressions 3.3
Evaluation of........................3.3.4
Formation of 3.3.3
IF Statement 4.3.2
NOR 3.3.2
Numbers 2.4.5
Operations 3.3.2
OR 3.3.2
Quantities 3.3
Record Characteristics....................5.8
Statement 2.3.5
Type Statement..........................2.3
Loop, DO 4.4.1
LSIGN, Intrinsic Function Format T7-2
LT....................................3.3.1
LUN 7.3.14
LUNO 7.3.12

M G.2.1.3
MACRO:
FTNCAL B.5.1.1
FTNRET B.6.2.3
RCALL B.6.2.2
RSUB B.6.2.1
MACROs:
Creating the B.5
Support B.5.2
Using the.............................B.6
Map Output, TXDS Load G.2.2.2
Mass Storage:
File Input/Output Statements 5.1
5.5, E.4.5
Inputting the PROM Program
from 7.5.3.1
Mathematical Function 2.8
MAX0, Intrinsic Function Format T7-2
MAX1, Intrinsic Function Format T7-2
MDATE, Subroutine 7.4.12
Memory:
Management, Reentrant Subprogram E.3.6
Option G.2.1.3

Representation 2.4.2.2
Message and Error Logging E.3.5
Messages, Error 9.2, 9.3
MFLD, Function D.6
Military Date, Obtain 7.4.12
MIN0, Intrinsic Function Format T7-2
MIN1, Intrinsic Function Format T7-2
MOD, Intrinsic Function Format T7-2
MODAPW Subroutine 7.3.15
Module:
IMGBLD 7.5.2.1
PRGROM 7.5.2.2
RDPROM 7.5.2.3
Subtraction, and Fraction Addition,
Double Precision Addition C.5
Modules:
Linking with User-Supplied E.3.7
Subroutine Package 7.5.2
Momentary Digital Output 7.3.6
Multiplication J.3.3, T3-1
F$XMD Double Precision C.7
F$XMR Single Precision C.6
Module:
Double Precision Floating Point C.7
Single Precision Floating Point C.6
Multi-key Index File Handler 7.8

Names in Explicit Type Statements 1.2.5
NE 3.3.1
Negative Sign T3-1
NERRST, Basic External Function
Format T7-1
Nested Groups 3.2.3
Nonreentrant:
Routines B.2
Subroutines B.6.1
Task H.2.2.1
and Dummy Reentrant Procedure,
Install H.2.3.3
and Reentrant Procedure,
Install H.2.3.2
Install Single H.2.3.1
With Reentrant Procedure H.2.2.1
Not Equal to 3.3.1
Number, Statement T1-1
Numbers:
Complex 2.4.4
Examples, Double Precision 2.4.3.1
Fixed Point ....Appendix J
Logical 2.4.5
Real 2.4.2
Numerical Data Specifications 5.2.1
NUMREC 7.3.14

O G.2.1.3, H.2.1.4
Option G.2.1.3, TH-1
OS Dependent:
Error Logging Functions TE-1
Functions TE-1
I/O Except CRU Functions TE-1
Initialization Functions TE-1
ISA Extension Functions TE-1
ISA File Contention Functions TE-1
Termination Functions TE-1

Object:
Access Name H.2.1.2
Code 1.1, 9.4
File Defaults, TXDS G.2.1.2, 6.2.2.2
Output, TXDS G.2.1.2, G.2.2.2
ASCII Date 7.4.10
ASCII Time 7.4.11
Date and Time 7.4.9
Military Date 7.4.12
Time 7.3.8
OCRU, Call 7.4.8
Subroutine 7.4.8
OPCDE 7.3.14
OPENMT 7.6.1
OPENW:
Call 7.3.12
Parameters 7.3.12
OPENW/CLOSEW Subroutine 7.3.12
Operands, Complex 3.3.1
Operating Characteristics E.2
Operations 3.3
Arithmetic 3.2.1
Logical 3.3.2
Relational 3.3.1
Operator Precedence 3.4.1, T3-1
OPN32I 7.7.1
Options H.2.1.4
C G.2.1.3, TH-1
D G.2.1.3, TH-1
F G.2.1.3, TH-1
O G.2.1.3, TH-1
R G.2.1.3, TH-1
S TH-1
TXDS G.2.1.3, G.2.2.3
X G.2.1.3, TH-1
Ordering of FORTRAN Source
Statements T1-2
Outer DO 4.4.1
Output:
Analog 7.3.9
CRU 7.4.8
Lists, Read and Write ....5.3.3.
Listing Elements 9.2
TXDS G.2.1.2, G.2.2.2
Listing G.2.1.2
Load Map G.2.2.2
Object G.2.1.2, G.2.2.2

PADSTR MACRO B.5.2.3
Parameters:
CFLW 7.3.10
CLOSEW 7.3.12
DFILW 7.3.11
DIW 7.3.4
DOLW 7.3.5
OPENW 7.3.12
RDRW 7.3.13
START 7.3.1
TRNON 7.3.2
WRTRW 7.3.13
Parenthesis, Closing 5.2.6
Pathname, TXDS Source File G.2.1.1
Pause Statement 4.6, 4.6.1
TXDS 4.6.1.3

2.X Releases 4.6.1.1
3.X Releases 4.6.1.2
Physical Record Characteristics 5.8
Precedence of Operators T3-1, 3.4.1
Precision Fixed Point Numbers J.2
Preparation, Program 1.2
Preset Random Number Generator 7.4.2
PRGROM Module 7.5.2.2
Print Width H.2.1.5
Procedure:
Link Control File for Linking a
Task and a FH-3
Load Stand-alone Program G.3.2
Processing:
End Vector E.3.4
Termination E.3.3
Program:
Allocation 9.2
Assembly Language FB-3, FB-7
Compilation H.2.1
TXDS G.2.1
Descritions, Sample..................7.5.3.4
Error Diagnostics 9.3.2
Execution H.2, H.2.5
Execution using Synonyms H.2.5.1
Execution with Global LUNOs......H.2.5.2
FORTRAN FB-1, FB-5
from Mass Storage, Inputting the
PROM 7.5.3.1
Generation Steps, TXDS G.2
Installation H.2.3
Link Edit B.4, H.2.2
TXDS G.2.2
Load and Execution, TXDS G.2.3
Preparation 1.2
Restrictions 1.2.5
Source 9.2
Start a 7.3.1
Programmer Modules, PROM 7.5.2
Programming Technique...........Section VIII
PROM:
Example F7-4
Program from Mass Storage,
Inputting the 7.5.3.1
Programmer Modules 7.5.2
Reading the 7.5.3.3
Writing the 7.5.3.2
Pseudo-Random Number 7.4.1

Quantities, Logical........................3.3

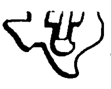
R G.2.1.3, H.2.1.4
Option G.2.1.3, TH-1
Random Number Generator, Preset........7.4.2
RANF Subroutine.......................7.4.1
Range Limit 4.4.1
Rank, Conversions to Higher 3.2.3
RANSET, Subroutine 7.4.2
RCALL MACRO B.5.1.6
RCALL MACRO B.6.2.2
RDIBIT 7.7.5
RDMTR................................7.6.4
RDMTS 7.6.5

RDPROM:
CALL 7.5.3.3
Module 7.5.2.3
RDRW:
CALL 7.3.13
Parameters..........................7.3.13
RDRW/WRTRW Subroutine............7.3.13
RDSTS 7.6.3
Read and Relocate, Example F7-5
Read and Write:
Input and Output Lists.................5.3.3
Record Characteristics 5.3.2
Read Sequential File Characteristics........5.3.1
Read Statements 5.1, 5.3, 5.6.2
Formatted 5.3.4
Type 2.3
Unformatted 5.3.6
Read/Write:
Calling Sequence E.4.1
Formatted E.4.1.1
Direct Access E.4.1.3
Unformatted E.4.1.2
Direct Access E.4.1.4
Reading the PROM 7.5.3.3
READ32 ....7.7.4
Real T3-2
Intrinsic Function Format T7-2
Numbers 2.4.2
Examples,..........................2.4.2.1
Statement 2.3.2
RECLNGTH 7.3.10, 7.3.14
RECNUM......................7.3.10, 7.3.13
Record Characteristics, Read and
Write 5.3.2
Record:
Fields in a Source 1.2.1
Formatted 5.3.2
Separation Indicator...................5.2.6
Recursion, Subprogram 6.2.1
Redundant Common Entries..............2.7.1
Reentrant:
Object....................G.2.1.3, H.2.1.4
Procedure:
Install Nonreentrant Task
and H.2.3.2
Nonreentrant Task with H.2.2.1
Routines B.3
Statement 6.8
SUB B.3
Subprogram Memory Management E.3.6
Subroutines B.6.2
Reference, Function 2.8
Register Block 7.4.3
Relational Operations 3.3.1
Repeated Group and Field
Specifications 5.2.5
Repetition of:
Field Specifications 5.2.5.1
Groups 5.2.5.2
Replacing Standard Modules E.3
With User-Written Modules H.2.8
Representation:
External 2.4.6.1

Internal 2.4.6.2
Memory 2.4.3.2
Required Subroutines 9.2
Restrictions, Program 1.2.5
RETURN Statement.....................4.5.2
REWIND Statement 5.1, 5.5.1
Routines, Reentrant B.3
RSUB MACRO B.5.1.5, B.6.2.1, FB-1
Rules for Expression, Summary of 3.4
Runtime:
Description, FORTRAN ....Appendix E
Error:
Diagnostics 9.3.3
Messages T9-4
Traceback Information 9.5
Organization, FORTRAN Compiler
and ....Appendix I
Package:
Diagnostic Messages T9-3
FORTRAN FI-3

S Option.........................H.2.1.4, TH-1
Sample Program Descriptions 7.5.3.4
Scalar Variables 1.2.5
Scale Factors, Field 5.2.1.7
Scaling Loss of Precision....................J.4
Separation:
Comma 5.2.6
Indicator, Record 5.2.6
Set, Character TA-1
Shared Data Block, Linking for H.2.2.1
Sign:
Addition T3-1
Intrinsic Function Format T7-2
Negative T3-1
Positive T3-1
SIN, Basic External Function Format.......T7-1
Single Precision:
Addition, F$XAR C.4
Division, F$XDR C.8
Division Module C.8
F$XCER Extended Integer to C.2
F$XCIR Integer to C,2
Floating Point:
Addition C.4
Multiplication Module C.6
Multiplication, F$XMR C.6
Subtraction, F$XSR C.4
to Extended Integer, F$XCRE............C.3
to Integer, F$XCRI.....................C.3
SINH, Basic External Function Format T7-1
Slash 5.2.6
Slashes, Consecutive......................5.2.6
SNGL, Intrinsic Function Format..........T7-2
Source:
Access Name.......................H.2.1.1
Code 1.1
File:
Defaults, TXDS G.2.1.1
Pathname, TXDS..................G.2.1.1
Program 9.2
Character Set.................Appendix A
Record, Fields in a 1.2.1
Statement Ordering 1.2.4
of FORTRAN T1-2

Specification:
A Field 5.2.3.2
D Field 5.2.1, 5.2.1.5
Data ....Section II
E Field 5.2.1, 5.2.1.3
F Field 5.2.1, 5.2.1.2
G Field 5.2.1, 5.2.1.4
H Field 5.2.3.3
I Field 5.2.1
Numerical Data.........................5.2.1
Subprograms for Data Initialization 6.6
T Field...............................5.2.3.5
X Field 5.2.3.4
Z Field 5.2.1, 5.2.1.6
Specified Time, Start a Program at a 7.3.2
SQRT, Basic External Function Format.....T7-1
Stand-alone:
Directory E.2.4
Execution, Linking for H.2.2.3
FORTRAN G.3
Program Procedure, Load G.3.2
Programs:
Executing G.3.3, H.2.5.3
Loading H.2.3.4, G.3.2
Standard:
Directories, DX10 3.X..................E.2.1
Modules, Replacing......................E.3
Start:
Program...............................7.3.1
At a Specified Time 7.3.2
Parameters............................7.3.1
Subroutine 7.3.1
Statement(s)..............................1.2.2
Accept.........................5.7, 5.7.1
Arithmetic if 4.3.1
Assign 4.2, 4.2.3
Assigned GO TO 4.2, 4.2.3
Assignment...............3.4.2, J.3.1, T3-2
Backspace.......................5.1, 5.5.2
Block Data 6.6, 6.6.2
CALL 4.5.1
Common 2.7, 2.7.1
Complex 2.3, 2.3.4
Computed GO TO.................4.2, 4.2.2
Continue 1.3, 4.4, 4.4.2
Copy 1.3
Data 6.6, 6.6.1
Decode 5.4.2
Define File 5.6.1
Dimension 2.6.3
Display 5.7, 5.7.2
DO 4.4, 4.4.1
Double Precision 2.3, 2.3.3
Encode 5.4.1
END.....................1.2.2, 4.6, 4.6.3
End File 5.1, 5.5.3
Equivalence 2.7, 2.7.2
Error Diagnostics 9.3.1
External 2.3, 6.7
Field T1-1
Fixed 2.3, 2.3.6
Format 5.1, 5.2
Formatted:
READ 5.3.4
WRITE 5.3.5



FORTRAN 1.2.2
Function Definitions 6.3
GO TO 4.2
IF 4.3
Implicit 2.3.7
Integer 2.3, 2.3.1
Logical 2.3.5
IF 2.3, 4.3.2
Mass Storage File I/O 5.1, 5.5, E.4.5
Number 1.2.5, T1-1
Ordering, Source 1.2.4
Pause 4.6, 4.6.1
READ 2.3, 5.1, 5.3, 5.6.2
Real 2.3.2
Reentrant 6.8
RETURN 4.5.2
REWIND 5.1, 5.5.1
STOP 4.6, 4.6.2
TXDS:
PAUSE 4.6.1.3
STOP 4.6.2.3
Unformatted:
READ 5.3.6
WRITE 5.3.6
Unconditional GO GO 4.2, 4.2.1
WRITE 5.1, 5.3, 5.6.3
2.X Releases:
PAUSE 4.6.1.1
STOP 4.6.2.1
3.X Releases:
PAUSE 4.6.1.2
STOP 4.6.2.2
STOP Statement 4.6, 4.6.2
TXDS 4.6.2.3
2.X Releases 4.6.2.1
3.X Releases 4.6.2.2
Storage, Inputting the PROM Program from Mass 7.5.3.1
Storage of Array Elements 2.6.2
Stored as Data, Formats 5.2.7
SUB, Reentrant B.3
Subprogram(s) 6.2
Called, Unique 1.2.5
Definitions 6.2.1
Function 6.4
Recursion 6.2.1
Subroutine 4.5.1, 6.5
Subroutine:
ADATE 7.4.10
AIRDW 7.3.9
AISQW 7.3.9
ALSQW 7.3.9
AOW 7.3.9
Assembly Language FB-6
Test FB-2
ATIME 7.4.11
BIDTSK 7.4.5
BUFIN 7.2.1
BUFOUT 7.2.2
CFILW 7.3.10
DATIME 7.4.9
DFILW 7.3.11
DIW 7.3.4
DLYBID 7.4.6
DOLW 7.3.5
DOMW 7.3.6
DNCASE D.8
FORTRAN FB-4, FB-8
ICRU 7.4.7
LOC 7.4.4
MDATE 7.4.12
MODAPW 7.3.15
OCRU 7.4.8
OPENW/CLOSEW 7.3.12
Package Modules 7.5.2
RANF 7.4.1
RANSET 7.4.2
RDRW/WRTRW 7.3.13
START 7.3.1
Subprogram 4.5.1, 6.5
SVC 7.4.3
SVCFUT 7.3.14
Time 7.3.8
TRANS D.10
TRNON 7.3.2
UPCASE D.9
Wait 7.3.3
Subroutines:
BUFIN/BUFOUT E.4.3
FORTRAN-PROM Program 7.5
Library 7.2
Nonreentrant B.6.1
Reentrant B.6.2
Required 9.2
Subscripts, Arrays, and 2.6.1
Subscripts Permissible, FORTRAN 2.6.1
SUBSTR, Function D.2
Subtraction T3-1, J.3.2
F$XSD Double Precision C.5
F$XSR Single Precision C.4
Summary of Rules for Expressions 3.4
Supervisor Call, Generate 7.4.3
Support MACROS B.5.2
SVC:
Block 7.4.3
CALL 7.4.3
Subroutine 7.4.3
SVCFUT Subroutine 7.3.14
Synonym Assignment:
DX10 with H.2.4.1
Linking for Execution with H.2.2.1

T: Field Specification 5.2.3.5
Table, IERR Error T7-3
TANH, Basic External Function Format T7-1
Task and a Procedure, Link Control File for Linking FH-3
Task, BID 7.4.5
Technique, Programming .... Section VIII
Termination:
Functions OS Dependent TE-1
of TXDS, Error Logging and G.4
Processing E.3.3
under DX10, Error Logging and H.2.7
Terms, Definitions of 2.2
Test Value 4.4.1

Time:
- Obtain 7.3.8
  - ASCII 7.4.11
- Subroutine 7.3.8

Traceback Information, Runtime Error 9.5
TRANS, Subroutine D.10
Transfer:
- Into DO LOOP 4.4.1
- of Control 4.5
- Out DO LOOP 4.4.1

Transmission, Internal 5.4
TRNON:
- Parameters 7.3.2
- Subroutine 7.3.2

TXDS 1.1
- Assigning LUNOs to VDTs on G.5
- Compile and Link Edit G.3.1
- Compiler F.1
- Control File Defaults G.2.2.1
- Error Logging and Termination of G.4
- FORTRAN:
  - Installation on ....Appendix F
  - Verification F.3
- Input G.2.1.1, G.2.2.1
- Link Editor F.2
- Listing:
  - File Defaults G.2.1.2
  - Output G.2.1.2
- Load Map Output G.2.2.2
- Object:
  - File Defaults G.2.1.2, G.2.2.2
  - Output G.2.1.2, G.2.2.2
- Options G.2.1.3, G.2.2.3
- Output G.2.1.2, G.2.2.2
- PAUSE Statement 4.6.1.3
- Program:
  - Compilation G.2.1
  - Generation Steps G.2
  - Link Edit G.2.2
  - Load and Execution G.2.3
- Source File:
  - Defaults G.2.1.1
  - Pathname G.2.1.1
- STOP Statement 4.6.2.3
- Work File G.2.2.2

TXLOBJ Directory E.2.3
Type 7.3.14
- Conversion T3-2
- Statement 2.3
  - Complex 2.3
  - Double Precision 2.3
  - Fixed 2.3
  - Integer 2.3
  - Logical 2.3
  - Read 2.3

Types, Data 2.4

Unconditional GO TO Statement 4.2, 4.2.1
Unformatted:
- Direct Access READ/WRITE E.4.1.4
- Read Statements 5.3.6
- READ/WRITE E.4.1.2
- WRITE Statements 5.3.6

Unique:
- Extended:
  - Complex Constants 1.2.5
  - Double Precision Constants 1.2.5
  - Integer Constants 1.2.5
  - Real Constants 1.2.5
- Integer Constants 1.2.5
- Logical Constants 1.2.5
- MACRO B.5.2.2
- Subprograms Called 1.2.5

Unit Numbers, FORTRAN 5.9
UPCASE, Subroutine D.9
User:
- Callable MACROs B.5.1
- FORTRAN Routine 7.5.3
- Supplied Modules, Linking with E.3.7
- Written Modules, Replacing Standard Modules with H.2.8

Using:
- MACROS B.6
- Synonyms, Program Execution H.2.5.1

Valid 7.3.14
- FORTRAN I/O Operatons 5.10

Validate DX FORTRAN I/O Operations T5-1
Validate TX FORTRAN I/O Operations T5-2
Value:
- Initial 4.4.1
- Test 4.4.1

Variable(s) 2.2, 2.2.2, 3.4.2
- Array 1.2.5
- Common 1.2.5
- Cross-Reference List G.2.1.3, H.2.14, FH-2
- Dimensioned 2.6
- Names, Equivalenced 1.2.5
- Scalar 1.2.5

VDTs on TXDS, Assigning LUNOs to G.5
Verification, TXDS FORTRAN F.3

Wait:
- Call 7.3.3
- Subroutine 7.3.3

Width, Print H.2.1.5
Work File, TXDS G.2.2.2
WRDCNT 7.3.13
Write Sequential File Characteristics 5.3.1
WRITE Statement(s) 5.1, 5.3, 5.6.3
- Carriage Control Editing 5.3.5.1
- Formatted 5.3.5
- Unformatted 5.3.6

Writing the PROM 7.5.3.2
WRMSK 7.7.3
WRMTR 7.6.6
WRMTS 7.6.7
WRTRW:
- Call 7.3.13
- Parameters 7.3.13

X G.2.1.3, H.2.1.4
- Field Specification 5.2.3.4
- Option G.2.1.3, TH-1

Z: Field Specification 5.2.1, 5.2.1.6

1ABS, Intrinsic Function Format T7-2
2.X Releases:
PAUSE Statement 4.6.1.1
STOP Statement 4.6.2.1
3.X Releases:
PAUSE Statement 4.6.1.2
STOP Statement 4.6.2.2

32 Input/Transition Detection Module 7.7
5MT/6MT Serial Interface Module 7.6
990 FORTRAN 1.1