User's Manual

FORTRAN IV

093-000053-08

Ordering No. 093-000053
© Data General Corporation, 1971, 1972, 1973, 1974, 1975
All Rights Reserved.
Printed in the United States of America
Rev. 08, February 1975

NOTICE

Data General Corporation (DGC) has prepared this manual for use by DGC personnel, licensees and customers. The information contained herein is the property of DGC and shall neither be reproduced in whole or in part without DGC prior written approval.

DGC reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented, including but not limited to typographical, arithmetic, or listing errors.

Original Release	March 1971
First Revision	March 1971
Second Revision	October 1971
Third Revision	February 1972
Fourth Revision	July 1972
Fifth Revision	April 1973
Sixth Revision	December 1973
Seventh Revision	July 1974
Eighth Revision	February 1975

This revision to the FORTRAN IV User's Manual, 093-000053-08, supersedes 093-000053-07. This revision is a minor revision to the manual.

INTRODUCTION

Format of the FORTRAN IV Manual

The FORTRAN IV User's Manual has been segmented into three parts. Part I consists of a description of the DGC FORTRAN IV language. Part II describes the FORTRAN IV interface to the DGC operating systems. Chapters 1, 2, and 3 of Part II apply to users having either SOS, RTOS, or RDOS. Chapters 4 and 6 apply to users of RTOS or RDOS, and Chapters 5 and 7 apply only to users of RDOS. The third section of the manual consists of appendices which may be used with both Part I and Part II of the manual.

Features of FORTRAN IV

Data General's FORTRAN IV for the DGC family of computers is an implementation of the ANSI FORTRAN Standard X3.9-1966 plus extensions. Certain restrictions in the interest of compiler efficiency and the run-time stack placement of variables will be noted in this manual, where pertinent, and summarized in Appendix C.

All improved features of FORTRAN IV, including complex arithmetic, logical expressions, labeled COMMON, DATA initialization, and run-time FORMATs, are included in Data General's FORTRAN IV. Thus, the DGC computer user has a widely known language fully adapted for any major computer application.

Extensions include certain important features of FORTRAN that are not yet standardized. Some of these are double precision complex arithmetic, mixed mode arithmetic, generalized subscript expressions, abnormal returns from subprograms via a dummy variable, Hollerith constants surrounded by quotation marks or apostrophes, array declarations in which the lower subscript bound need not be one, full word bit-by-bit logical operations, a string FORMAT descriptor, reentrant subroutines, and end-of-file and error returns from I/O statements.

Extensions to I/O offer the user a choice of standard formatting or simplified input/output using either conversational I/O from the teletypewriter or programmed, unformatted I/O. Both ASCII and binary I/O are implemented. All I/O can be device independent with devices assigned to channel numbers at run time.

The code generated provides optimized register and storage allocation and reentrant machine language code, which can be interfaced with assembly language code. Since the generated code is in the form of assembler source code, the user can include segments of his own machine language code directly in a FORTRAN-generated program. An option permits the user to intersperse assembly language instructions with FORTRAN statements in the FORTRAN source code as well. Another option allows the user to selectively inhibit the compilation of any source statement.

Additional Reference Material

Insofar as feasible, the FORTRAN IV manual constitutes a complete guide to the FORTRAN user's needs under any DGC operating system. For more detailed descriptions and further uses of the operating systems and related utility programs, refer to one or more of the following manuals:

Manual	Number
FORTRAN IV Run Time Library User's Manual	093-000068
Extended Relocatable Assembler	093-000040
Introduction to RDOS	093-000083
RDOS User's Manual	093-000075
RDOS User Device Driver Implementation	017-000002
BATCH User's Manual	093-000087
Introduction to RTOS	093-000093
RTOS User's Manual	093-000056
RTOS User Device Driver Implementation	017-000006
Stand-alone Operating System User's Manual	093-000062
Relocatable Loaders User's Manual	093-000080
Octal Editor User's Manual	093-000084
Library File Editor User's Manual	093-000074
Text Editor User's Manual	093-000018
DOS - Compatible SOS System Manual	093-000094
The Symbolic Debugger	093-000044
RTIOS User's Manual	093-000095
Discrete Fourier Transform	093-000104
Commercial Subroutine Package	093-000106
Dataplot User's Manual	093-000060

PART I

Chapter 1 - FORTRAN PROGRAMS
Chapter 2 - ARITHMETIC AND STRING DATA
Chapter 3 - EXPRESSIONS
Chapter 4 - ASSIGNMENT STATEMENTS
Chapter 5 - CONTROL STATEMENTS
Chapter 6 - INPUT/ OUTPUT STATEMENTS
Chapter 7 - SPECIFICATION STATEMENTS
Chapter 8 - DATA INITIALIZATION
Chapter 9 - FUNCTIONS AND SUBROUTINES

TABLE OF CONTENTS

CHAPTER 1 - FORTRAN PROGRAMS

Program Units 1-1
Lines of Program Text 1-1
Comment Line (C) 1-1
Optionally Compiled Line (X) 1-1
Assembly Source Code Line (A) 1-2
Label 1-2
Comment Following 1-2
FORTRAN Statements 1-2
Continuation Lines 1-2
Partial Ordering of Statements 1-2
FORTRAN Source Program 1-3
FORTRAN Character Set 1-3

CHAPTER 2 - ARITHMETIC AND STRING DATA

Constants, Variables and Parameters 2-1
Integer Data 2-2
Real Data 2-3
Double Precision Data 2-3
Complex Data 2-4
Double Precision Complex Data 2-4
Logical Data 2-5
String (Hollerith) Constants 2-5
Arrays and Subscripts 2-6

CHAPTER 3 - EXPRESSIONS

Definition of an Expression 3-1
Arithmetic Expressions 3-1
Sample Arithmetic Expressions 3-3
Relational and Logical Expressions 3-3
Relational Expressions 3-3
Logical Expressions 3-4
Evaluation of Logical and Relational
Expressions 3-4

CHAPTER 4 - ASSIGNMENT STATEMENTS

Definition 4-1
Assignment to a Variable 4-1
Assignment Examples 4-2

CHAPTER 5 - CONTROL STATEMENTS

Definition 5-1
Unconditional GOTO Statement 5-1
Computed GOTO Statement 5-1
Assigned GOTO Statement 5-2
ASSIGN Statement 5-2

CHAPTER 5 - CONTROL STATEMENTS (Continued)

Arithmetic IF Statement 5-3
Logical IF Statement 5-3
CALL Statement 5-3
RETURN Statement 5-4
CONTINUE Statement 5-4
PAUSE Statement 5-5
STOP Statement 5-5
DO Statement 5-5

CHAPTER 6 - INPUT/OUTPUT STATEMENTS

FORTRAN Input/Output 6-1
Programmed I/O Using READ and WRITE 6-1
I/O Lists of READ and WRITE Statements. 6-2
Unformatted I/O 6-3
Formatted I/O 6-6
FORMAT Statement. 6-6
Specification of Format Information 6-6
Separators of Descriptor Field 6-7
Basic Numeric Field Descriptors. 6-7
Numeric Conversion on Input 6-7
Output Conversion of Integers 6-8
Output Conversion of Real and Double Precision Data 6-8
Radix 8 I/O Using the O Specifier 6-10
Non-numeric field Descriptors 6-10
I/O of Logical Data 6-10
Positioning Descriptors 6-11
String Data 6-11
String Literals 6-12
Alphabetic Data 6-12
Multiple Record Forms 6-13
Vertical Carriage Control 6-15
Scale Factor 6-17
Run Time Format Specifications 6-18
Binary I/O 6-19
Teletype I/O 6-20
ACCEPT and TYPE Statements 6-20
Sample Program 6-21
Rules of Teletype I/O 6-21
CONTROL I/O 6-23
Channel Access 6-23
End-of-File or Transfer of Control 6-23
REWIND Statement 6-23
ENDFILE Statement 6-24
Random Access Files (FSEEK) 6-24
Rereading and rewriting Records (CHSAV, CHRST) 6-24

CHAPTER 7 - SPECIFICATION STATEMENTS

Definition 7-1
DIMENSION Statement. 7-1
Data-type Statements 7-3
COMPILER DOUBLE PRECISION Statement 7-3
COMMON Statement 7-4
EQUIVALENCE Statement 7-5
EXTERNAL Statement 7-6
COMPILER NOSTACK 7-7

CHAPTER 8 - DATA INITIALIZATION

DATA Statement 8-1
BLOCK DATA Subprogram 8-2

CHAPTER 9 - FUNCTIONS AND SUBROUTINES

Functions 9-1
  Statement Functions 9-1
  Function Subprograms 9-2
  Arguments of Function Subprograms 9-4
  FORTRAN Library Functions 9-4
Subroutines 9-8
Abnormal Returns 9-9
DGC Fortran IV Library 9-10
Bit/ Word Manipulation 9-10
  Clear a Bit (ICLR, BCLR) 9-10
  Set a Bit (ISET, BSET) 9-10
  Test a Bit (ITEST, BTEST) 9-11
  Shift a Word (ISHFT) 9-12

CHAPTER 1

FORTRAN PROGRAMS

PROGRAM UNITS

A FORTRAN program is made up of one or more program units. Program units are separately compiled entities that may be any of the following:

1) Main program
2) Subroutine subprogram
3) Function subprogram
4) Block Data subprogram
5) Task subprogram*

A FORTRAN program can have only one main program as a program unit.

FORTRAN program units are implemented as reentrant programs. All variables and arrays not declared as being in COMMON storage (see COMMON statement, Chapter 7) are placed on a run-time stack. Repeated entry prior to taking a RETURN can be made to any program that does not alter COMMON storage.

LINES OF PROGRAM TEXT

The source text of a program unit consists of those ASCII characters that make up the FORTRAN character set. The text is divided into lines and terminates at an END line. An END may appear anywhere on the line, starting at or after character position 7 and must be terminated by a carriage return.

Comment Line (C)

A line of text that has a C in character position 1 is a comment line. The comment may be written anywhere in the line following the C.

Optionally Compiled Line (X)

A line of text that has an X in character position 1 will be optionally compiled. At compile time, the compiler tests whether or not lines having an X in that position should be compiled as part of the source program. If yes, all lines having an X in column 1 will be compiled; if no, the lines containing X in column 1 will be ignored. See Appendix D, "Operating Procedures", for a description of how to use the X option.

*A task subprogram is used under the Real Time Disk Operating System or the Real Time Operating System. Task subprograms can be activated by the run time routines FTASK or ITASK, discussed in PART II of this manual.

LINES OF PROGRAM TEXT (Continued)

Assembly Source Code Line (A)

Lines of assembly source code may be included in a FORTRAN source program. A line of assembly source code must have an A in character position 1. The compiler will delete the A when the line is encountered and pass the line intact to the assembler.

Label

If a line does not have a C, X, or A in character position 1, character positions 1 through 5 are reserved for a label. If the line contains an X in column 1, character positions 2 through 5 are reserved for a label.

A label can be any unsigned integer of 1 to 5 digits and can be placed anywhere in character positions 1 through 5.

Leading zeroes are significant in labels; 12 and 0012 will be treated as two different labels.

Comment Following Semicolon

The syntactical scan of a line may be terminated by a semicolon. A semicolon in column 7 or any character position thereafter reserves the remainder of the line for an optional comment. (A semicolon appearing within a Hollerith constant is not recognized for this purpose.)

FORTRAN STATEMENTS

The basic semantic unit of a FORTRAN program unit is called a statement. A line of text may contain a FORTRAN statement or part of a FORTRAN statement.

A statement must start at character position 7 or beyond. The programmer can press the TAB key once to tabulate to position 8 or can press the space bar 6 times. The programmer can also tabulate to position 8 after a label.

Continuation Lines

When a FORTRAN statement requires more than one line of text, continuation lines must be indicated by putting a character other than 0 or blank in character position 6 of the continuation line. Initial FORTRAN statment lines must have 0 or blank in character position 6. Continuation lines must never be labelled.

Partial Ordering of Statements

For compiler efficiency, Data General's FORTRAN IV requires a partial ordering of statements. The order of statements is:

1) COMPILER DOUBLE PRECISION or COMPILER NOSTACK statement.

2) OVERLAY or CHANTASK statement.

3) PARAMETER statements.

4) FUNCTION, SUBROUTINE, and TASK statements.

5) Declaration statements. These begin with the keywords: COMMON, COMPLEX, DIMENSION, DOUBLE PRECISION, EQUIVALENCE, EXTERNAL, INTEGER, LOGICAL, or REAL.

*6) Statement functions.

*7) Executable statements.

*FORMAT statements and DATA initialization statements are permissible in either category.

FORTRAN SOURCE PROGRAM

An example of a DGC FORTRAN IV source program is:

```
      REAL FUNCTION MAG (A, I, J)
      DIMENSION A (20,20)
C     ASSUME A (1,1) LARGEST
      MAG=A(1,1)
C     SCAN ARRAY A TO CHECK ASSUMPTION
      DO 5 K = 1, I
      DO 6 L = 1, J
C     IF A(K, L) IS LARGER, SUBSTITUTE IT
C     FOR A(1,1)
      IF (MAG. LT. A(K, L))MAG=A(K, L)
6     CONTINUE
5     CONTINUE
      RETURN
      END
```

FORTRAN CHARACTER SET

The ASCII characters that make up the FORTRAN character set are:

Letters: A through Z
Digits: Ø through 9
Special symbols listed below:

Symbol	Name of Symbol
Δ	Blank
=	Equals
+	Plus
-	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
$	Currency Symbol
:	Colon
'	Apostophe
"	Quotation Mark
!	Exclamation Point

Blanks are significant delimiters in Data General's FORTRAN IV, except as noted in Chapter 2 in regard to variable names and in Chapter 5 in regard to the GO TO statement.

All ASCII characters are allowed in Hollerith constants with the exception of those characters that have special meaning for the operating system in control or for the Extended Assembler. The characters that are not permitted are:

FORTRAN CHARACTER SET (Continued)

ASCII Code	Character
012	Line Feed
014	Form Feed
015	Carriage Return
034	Shift L (\)
074	<

CHAPTER 2

ARITHMETIC AND STRING DATA

CONSTANTS, VARIABLES AND PARAMETERS

A constant is a known value that does not alter during program execution.

A variable is represented by a symbolic name and is a quantity that may be altered during execution.

The symbolic name of the variable must consist of from 1 to 31 alphanumeric characters, beginning with a letter. Subprogram names must be distinguishable within the first five characters for compatibility with the DGC Assembler and Loader.

Imbedded blanks appearing within variable names are not significant. For example, the following identifiers are equivalent.

```
XSQUARE

X  SQUARE

X S Q U A R E
```

DGC FORTRAN IV has a list of reserved words consisting of statement names (DO, PAUSE, etc.), library function names (SQRT, AIMAG, etc.) and operator names (.AND., .LE., etc.).* The reserved words cannot be used as variable names. In addition, imbedded blanks in a variable name that cause a portion of the name to be recognized as a reserved word are not permitted. For example:

DOZEN	legal variable name
DO ZEN	illegal variable name

A parameter is represented by a symbolic name and can be used anywhere a constant of the same type can be used. (Statement numbers, octal strings in PAUSE and STOP statements, and numbers in FORMAT statements are not constants.)

A parameter has the data type of the associated constant. Parameters are given their value in a PARAMETER statement. The format of the PARAMETER statement is:

PARAMETER $v_1=c_1, v_2=c_2, \ldots, v_n=c_n$

*This exception to ANSI FORTRAN Standard X3.9-1966 was made in the interest of FORTRAN compiler efficiency. A list of the reserved words is given in Appendix B.

CONSTANTS, VARIABLES, AND PARAMETERS (Continued)

where: each v is a variable name.

each c is a numerical or logical constant

Example of PARAMETER statement:

```
PARAMETER PI=3.141592653, Q1=.1731523D-7
```

Examples of use of parameter K:

```
PARAMETER K = 8
COMMON / COMLABEL / C1(K), C2(K)
DATA C1/K*K/
DO 1 I=1, K
C1(I) = I*K
CALL SUBROUT (C1, AB, K)
```

Constants and variables have data types associated with them. Mathematical data may be of types INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or DOUBLE PRECISION COMPLEX.

INTEGER type data is represented internally in fixed-point notation. All other mathematical data types are represented internally in floating-point notation.

Constants and variables may be associated with a LOGICAL data type. In addition, string constants are permitted in the source code. String constants cannot be associated with parameters.

Integer Data

An integer constant is a signed or unsigned whole number written without a decimal point.

An integer variable is usually implicitly typed, i.e., if the first character of the symbolic name is I, J, K, L, M, or N, the symbolic name represents an integer variable unless otherwise specified. Examples of integer constants and variables are:

Constants	Variables
-125	ITEM
0	JOBNO
+4525	LUCKY
377K	MASKBYTE

As shown, integer constants can be specified in octal format by writing the number followed by the letter K. Some additional examples are:

Integer Data (Continued)

Octal Constant	Decimal Value
10K	8
777K	511
-1K	-1

An integer datum is stored in one word (16 bits). The range of integer values is -32,767 to 32,767 inclusive.

Real Data

A real constant is signed or unsigned and consists of one of the following:

1) One or more decimal digits written with a decimal point.

2) One or more decimal digits written with or without a decimal point, followed by a decimal exponent written as the letter E followed by a signed or unsigned integer constant. When the decimal point is omitted, it is always assumed to be immediately to the right of the rightmost digit. The exponent value may be explicitly 0; the exponent field may not be blank.

A real variable is usually implicitly typed. If the first character of the symbolic name is not I, J, K, L, M, or N the symbolic name represents a real variable unless otherwise specified.

Constants	Constant Value	Variables
0.0	0.0	ALPHA
.000056789	.000056789	B25
+15.E-04	+.0015	EXIT
-005E2	-500	C

A real datum is stored in two 16-bit words.

Double Precision Data

A double precision constant is signed or unsigned and consists of the following:

A sequence of decimal digits written with or without a decimal point, followed by a decimal exponent written as the letter D followed by a signed or unsigned integer constant. When the decimal point of a double precision constant is omitted, it is always assumed to be immediately to the right of the rightmost digit. The exponent value may be explicitly 0; the exponent field may not be blank.

A double precision variable must be explicitly specified as such in a DOUBLE PRECISION type statement. *

*If the first statement of the FORTRAN program is: "COMPILER DOUBLE PRECISION" each real variable or constant will be forced to type DOUBLE PRECISION. See page 7-3.

Double Precision Data (Continued)

Constants	Constant Value
-21987654321D0	-21987654321
5.0D-3	.005
.203D0	.203

Variable Type Statement

```
DOUBLE PRECISION D, E, F2
```

A double precision datum is stored in 4 words.

Complex Data

A complex constant is an ordered pair of signed or unsigned real constants, separated by a comma and enclosed in parentheses.

A complex single-precision variable must be explicitly specified as such in a COMPLEX type statement.

Constants	Constant Value
(3.2, 1.86)	3.2+1.86i
(2.1, 0.0)	2.1+0.0i
(5.0E3, -2.12)	5000. -2.12i

Variable Type Statement

```
COMPLEX C1, C2
```

A complex single-precision datum is stored in 4 words.

Double Precision Complex Data

A double precision complex constant is an ordered pair of signed or unsigned double precision constants separated by a comma and enclosed in parentheses.

A double precision complex variable must be explicitly specified as such in a DOUBLE PRECISION COMPLEX type statement. *

Constant

```
(-3456.0012D-5,.0034567D+3)
```

Variable Type Statement

```
DOUBLE PRECISION COMPLEX DC1, DC2
```

*If the first statement of the FORTRAN program is: "COMPILER DOUBLE PRECISION" each complex variable or constant will be forced to type DOUBLE PRECISION COMPLEX. See page 7-3.

Double Precision Complex Data (Continued)

A double precision complex datum is stored in 8 words.

Logical Data

A logical constant is a truth value written as:

.TRUE. or .FALSE.

A logical variable must be explicitly specified as such in a LOGICAL type statement. For example:

```
LOGICAL BOOL1, BOOL2
```

A logical datum is stored in one word. The value .TRUE. is stored as octal 177777 and .FALSE. as 000000. When testing for logical, any non-zero word will be treated as the value .TRUE. . (Octal 177777 is also the integer value -1.)

String (Hollerith) Constants

String constants are strings of characters of the FORTRAN character set, including blanks. A string may be enclosed in quotation marks, it may be enclosed in apostrophes, or it may be represented by a Hollerith constant. A Hollerith constant is an integer constant followed by the letter H followed by the string. The integer constant indicates the number of characters in the string:

```
"END"  }
'END'  }        equivalent strings
3HEND  }
```

Quotation marks may appear in strings that are surrounded by apostrophes; apostrophes may appear in strings that are surrounded by quotation marks. Both apostophes and quotation marks may be used as characters within Hollerith constants.

String constants may appear in:

1) The I/O list of a TYPE or ACCEPT statement. The constant is written out precisely as it appears in the statement.

2) A FORMAT statement. On output, the string is written to the output device. On input, the string is overwritten by an equal number of characters from the input record.

3) A DATA initialization statement. The data type of the corresponding variable (or variables) in the variable list is ignored.

4) The argument list of a CALL statement or the argument list of a function reference.

String constants may also appear in relational and logical expressions as described in Chapter 3. In these cases, however, only the first two characters of the string constant are significant and are treated as a one-word integer operand rather than a string.

CONSTANTS, VARIABLES AND PARAMETERS (Continued)

String (Hollerith) Constants (Continued)

Within string constants, octal codes for ASCII control characters, delimited by angle brackets, may appear. The codes will be passed to the DGC assembler for interpretation. For example, a carriage return can be passed in a string as follows:

```
"DATA FOLLOWS: <15> "
```

Note that when using formatted I/O (WRITE/FORMAT statements), carriage control information should not be passed in string constants but should follow ANSI FORTRAN standard conventions.

String constants are stored one character per byte (two characters per word). Normally, if the character count of a string is even, a word of all zeroes is generated to indicate the end of a string. This does not occur, however, in the case of DATA initialization using a string constant.

ARRAYS AND SUBSCRIPTS

An array is an ordered set of data of one or more dimensions. Up to 128 dimensions are permitted; a single symbolic name identifies the array. Each element of the array is identified by a qualifier of the array name, called a subscript.

An array is specified by appearance of its symbolic name in a DIMENSION, COMMON, or data type statement with parenthesized dimensioning information. Some examples are:

Statement	Meaning
DIMENSION A(10, 10)	A is a two-dimensional real array of 100 elements.
COMMON B(2, 5, 5)	B is a 3-dimensional real array of 50 elements stored in common.
INTEGER C(5, 2, 2, 2)	C is a 4-dimensional integer array of 40 elements.

Variables may be used in the specification of array subscripts in DIMENSION and data type statements. This is called adjustable dimensioning.

Variables can only be specified if the array name and the variable subscript bounds are dummy arguments of a subprogram in which they specify the array. Then, actual arguments giving values to variable subscript bounds can be passed when the subprogram is called. Example:

```
SUBROUTINE  SUB25 (MAT, I, J)
INTEGER I, J
REAL MAT (I,  0:J)
```

Values for I and J would be passed in a call to SUB25.

Note that array MAT is subscripted as

MAT(I, 0:J)

ARRAYS AND SUBSCRIPTS (Continued)

This specifies that the lower bound of the second subscript is 0 and the upper bound is J. If only the upper bound of an array dimension is specified, the lower bound is assumed to be 1, e.g.:

```
DIMENSION B(5)
```

assumes that the first array element is B(1) and the last is B(5). Both upper and lower bounds are specified in the following example:

```
DIMENSION F(0:11), G(-2:4, -3:0)
```

The array F is a one-dimensional array of 12 elements, the first of which is F(0). The second array is a two-dimensional array of 7x4 or 28 elements; the first element is G(-2, -3).

The subscript of an array element is written as a parenthesized list of subscript expressions. Each subscript expression can have one of the following forms:

1) In the non-executable statements EQUIVALENCE and DATA, each element of the subscript list must be an integer constant (or a parameter representing an integer constant.)

2) In expressions within executable statements, each element of the subscript list must be an expression whose value is of type integer.

Examples:

```
A(I, J)
    could be an element of array A(10,10)
B(1,1,1)
    is the first element of array B(2,5,5)
C(ITEM-1)
    could be an element of array C(0:6)
D(I+IFIX(SQRT(R -B/3.0)), J/K)
    could be an element of D(3,8)
```

The subscript of an array element cannot assume a value during execution that is less than the lower bound for that dimension of the array or larger than the upper bound for that dimension of the array, except if a single subscript is given for a multidimensional array. (A single subscript can be used to index a multidimensional array. If A is dimensioned (0:4, 0:4), the 25 elements of A can be referenced as A(1) through A(25). This is the same as the single subscript reference allowed in DATA and EQUIVALENCE statements.)

ARRAYS AND SUBSCRIPTS (Continued)

Values are assigned to array elements so that the first subscript expression varies most rapidly, then the second subscript expression, etc.

For example, elements of array C(15) are stored:

```
C(1), C(2),. . . , C(15)
```

Elements of array A(10,10) are stored:

```
A(1,1), A(2,1),. . . , A(9,1),
A(10,1), A(1, 2),. . . , A(9, 10), A(10, 10)
```

Elements of array B(2,3,4) are stored:

```
B(1,1,1), B(2,1,1), B(1,2,1), B(2,2,1),
B(1,3,1), B(2,3,1), B(1,1,2),. . . ,
B(1,3,4), B(2,3,4)
```

CHAPTER 3

EXPRESSIONS

DEFINITION OF AN EXPRESSION

An expression is a combination of data elements (variables, array elements, functions, and constants) with operators. The FORTRAN operators are arithmetic, relational and logical.

ARITHMETIC EXPRESSIONS

An arithmetic expression is formed with arithmetic operators and arithmetic data elements. The operators are:

Operator	Operation
+	Addition (or unary plus)
-	Subtraction (or unary minus)
*	Multiplication
/	Division
**	Exponentiation

An arithmetic datum has one of five possible data types:

Type	Rank of Data
double precision complex	highest
complex	↓
double precision	
real	
integer	lowest

The following rules apply to evaluating arithmetic expressions:

1) When either plus (+) or minus (-) is used as a unary operator, the data type of the result is the same as that of the operand.

2) When two operands of the same data type are used in an expression containing one of the operators

   + - * /

   the data type of the result is the same as that of the operand.

3) Mixed data type operands are permitted. In expressions formed with the operators

   + - * /

   the lower ranking data type operand is converted to a temporary of the higher ranking type, and the result of evaluation will have the higher ranking data type. When a COMPLEX or DOUBLE PRECISION COMPLEX operand is combined with an operand that is not complex, the temporary has an imaginary part equal to zero.

   When an expression consists of a DOUBLE PRECISION operand and a COMPLEX operand, the DOUBLE PRECISION operand is converted to a single precision COMPLEX temporary and the result of evaluation is COMPLEX.

3) (Continued)

Arguments of library functions are not converted to temporaries of the appropriate type.

4) Mixed data types are permitted in expressions consisting of base and exponent operands (** operator), except that it is illegal to raise an INTEGER base to a COMPLEX or DOUBLE PRECISION COMPLEX exponent.

When base and exponent operands are of the same type, the result is also that data type.

When the base and exponent are of differing data types, the resultant value will be of the higher data type, except in the case of raising a COMPLEX base to a DOUBLE PRECISION exponent. The result of this operation is DOUBLE PRECISION COMPLEX.

5) The following rules govern the order in which operations are evaluated within an arithmetic expression.

a) The arithmetic operators have the following precedence:

Operator	Precedence
**	highest (evaluated first)
/ *	↓
+ -	lowest (evaluated last)

b) When two operators are of equal precedence, operations are evaluated from left to right in the expression.

c) Parentheses are used to alter the order of operator precedence. A parenthesized expression is evaluated as an entity before further evaluation proceeds. When parenthesized expressions are nested, the innermost is evaluated first.

The rules for evaluating mixed data types are shown in tabular form in the tables following. The resulting data types are given in the double lined area.

OPERATORS + - * /		OPERAND A				
		Integer	Real	Double Precision	Complex	Double Complex
OPERAND B	Integer	Integer	Real	Double Precision	Complex	Double Complex
	Real	Real	Real	Double Precision	Complex	Double Complex
	Double Precision	Double Precision	Double Precision	Double Precision	Complex	Double Complex
	Complex	Complex	Complex	Complex	Complex	Double Complex
	Double Complex	Double Complex	Double Complex	Double Complex	Double Complex	Double Complex

ARITHMETIC EXPRESSIONS (Continued)

OPERATOR **		EXPONENT OPERAND Integer	Real	Double Precision	Complex	Double Complex
BASE OPERAND	Integer	Integer	Real	Double Precision	ILLEGAL	ILLEGAL
	Real	Real	Real	Double Precision	Complex	Double Complex
	Double Precision	Double Precision	Double Precision	Double Precision	Complex	Double Complex
	Complex	Complex	Complex	Double Complex	Complex	Double Complex
	Double Complex	Double Complex	Double Complex	Double Complex	Double Complex	Double Complex

SAMPLE ARITHMETIC EXPRESSIONS

Some examples of legal arithmetic expressions are shown below with the data type of each data element of the expression.

```
INTEGER I, J
REAL A, B
DOUBLE PRECISION D
COMPLEX C
DOUBLE PRECISION COMPLEX DC, CD

C*DC**C  }
B/(I+J)  }   legal arithmetic expressions
D-B**A   }
D*I**J   }
```

RELATIONAL AND LOGICAL EXPRESSIONS

Relational Expressions

A relational expression consists of 2 arithmetic expressions separated by a relational operator. The relational operators are:

Operator	Representing
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

RELATIONAL AND LOGICAL EXPRESSIONS (Continued)

Logical Expressions

A logical expression is formed with logical operators and logical or integer elements. Logical elements are those that have been given the data type LOGICAL. The logical operators are:

Operator	Representing
.OR.	Logical disjunction. Result is 1 if either operand has a 1 in that bit position.
.AND.	Logical conjunction. Result is 1 if and only if both operands are 1 in that bit position.
.NOT.	Logical negation. Result is the bit complement of the operand.

Evaluation of Logical and Relational Expressions

Logical and relational operations may be combined within expressions.

Logical and relational expressions can be treated as full-word operations, evaluated bit-by-bit, as in masking, or can be considered as evaluating to truth values:

.TRUE. or .FALSE.

where: .TRUE. is 177777_8 (-1)

.FALSE. is 000000_8 (0)

These octal values for .TRUE. and .FALSE. are generated for literals and as the result of evaluations to a truth value. When testing for a truth value, any non-zero word is considered .TRUE. and a word of all zeroes is considered .FALSE.

Hollerith (string) constants may appear wherever integers are permitted in logical and relational expressions. Only the first two characters of the string constant are significant, and represent the ASCII value of the characters. The first two characters of the string constant can be ANDed, ORed, or compared with integer and logical values.

The general rules of precedence in evaluating relational and logical expressions are the same as for arithmetic expressions: parenthesized expressions are first evaluated as entities and evaluation proceeds from left to right when two operators are of equal precedence.

Arithmetic expressions are evaluated first, in accordance with the rules of arithmetic operator precedence, then relational operations, and then logical operations. The precedence of all operators in a FORTRAN expression is:

Operator	Precedence
**	highest
*/	↓
+ -	↓
.GE.,.GT.,.EQ.,.NE.,.LT.,.LE.	↓
.NOT.	↓
.AND.	↓
.OR.	lowest

The tables following show examples of evaluation of logical and relational expressions, both for truth value results and on a full-word, bit-by-bit basis.

Evaluation of Logical and Relational Expressions (Continued)

TRUTH OPERATIONS

(.FALSE.=000000_8 .TRUE.=177777_8)

Logical Truth Table

Y Operand	Z Operand	.NOT.Y	Y.AND.Z	Y.OR.Z
.FALSE.	.FALSE.	.TRUE.	.FALSE.	.FALSE.
.FALSE.	.TRUE.	.TRUE.	.FALSE.	.TRUE.
.TRUE.	.FALSE.	.FALSE.	.FALSE.	.TRUE.
.TRUE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.

Logical and Relational Truth Evaluations

Assume that: A = .TRUE. W = 2 X = 4 Y = 6

Expression	Value	Interpretation
W.LE.X	.TRUE.	
W.LT.X.AND.W.LT.Y	.TRUE.	true and true
W.NE.X.AND..NOT.A	.FALSE.	true and false
.NOT.A.OR.W.EQ.X	.FALSE	false or false

FULL WORD OPERATIONS

Bit-by-Bit Logical Operations

Operands	.NOT.Y	.NOT.Z	Y.AND.Z	Y.OR.Z
Y = 101010 Z = 110100	010101	001011	100000	111110

Logical and Relational Full Word Evaluations

Assume that: J = 377_8 K = 47117_8 L = 200_8

Expression	Value	Interpretation
J .AND. K	117	Mask K with J.
.NOT. J .AND. K	47000	Mask K with the complement of J.
5*(-(K .EQ. L))	0	Since K does not equal L, the parenthesized expression evaluates to .FALSE. (0).
5*(- (K .EQ."NO"))	5	47117 is ASCII for NO. The result of the parenthesized expression is .TRUE. (-1).

CHAPTER 4

ASSIGNMENT STATEMENTS

DEFINITION

The format of an assignment statement is:

variable = expression

where: variable is a subscripted or non-subscripted variable name.

expression is any legal FORTRAN expression.

The expression on the righthand side of the equals sign is evaluated according to the rules given in Chapter 3, and the resulting value is assigned to the variable on the lefthand side of the assignment statement.

ASSIGNMENT TO A VARIABLE

The rules for assignment of the value of an expression to a variable are:

1) The only illegal assignment statements are:

   INTEGER = (DP) COMPLEX
   REAL = (DP) COMPLEX
   DOUBLE PRECISION = (DP) COMPLEX

2) If the expression and variable are of the same data type, the expression is simply evaluated and assigned.

3) INTEGER = REAL
   INTEGER = DOUBLE PRECISION

   Fix the floating point number and assign.

4) REAL = INTEGER

   Float the fixed point number and assign.

5) REAL = DOUBLE PRECISION

   Truncate the mantissa by 32 bits and assign.

6) DOUBLE PRECISION = INTEGER

   Float the fixed point number to a full 56 bits of mantissa and assign. (The expression is not floated to REAL and then extended by 32 bits of zeroes.)

ASSIGNMENT TO A VARIABLE (Continued)

7) DOUBLE PRECISION = REAL

   Extend the single precision mantissa by 32 bits of zeroes and assign.

   When assigning any single-precision expression to a double-precision variable, error message No. 2 will be given. This is a warning that the double-precision variable will only be precise to 6 or 7 decimal digits.

8) COMPLEX = DOUBLE PRECISION COMPLEX

   Truncate each mantissa by 32 bits and assign.

9) DOUBLE PRECISION COMPLEX = COMPLEX

   Extend each single-precision mantissa by 32 bits of zeroes. (Error message 47 will be given)

10) COMPLEX = INTEGER

    Float the fixed point number, set the imaginary part to zero, and assign.

11) COMPLEX = REAL

    Set the imaginary part to zero and assign.

12) COMPLEX = DOUBLE PRECISION

    Truncate the mantissa by 32 bits, set the imaginary part to zero, and assign.

13) DOUBLE PRECISION COMPLEX = INTEGER

    Float the fixed point number to a full 56 bits of zeroes, set the imaginary part to zero, and assign.

14) DOUBLE PRECISION COMPLEX = REAL

    Extend the single-precision mantissa by 32 bits of zeroes, set the imaginary part to zero, and assign.

15) DP COMPLEX = DOUBLE PRECISION

    Set imaginary part to zero and assign.

16) Assignment to a LOGICAL variable follows the rules of assignment to integers.

17) String constants may be assigned to INTEGER variables. The ASCII value of the first two characters of the constant are assigned.

ASSIGNMENT EXAMPLES

Some examples of assignment statements are:

ASSIGNMENT EXAMPLES (Continued)

```
S = 5.*(3**A+SQRT(A*I)/I)

B = C(I)+SIN(C(I))

LOGIC(4) = X.GT.5.OR.Y.LT.Z

K(I) = 34567.D+4/I

J = 10.0*SIN(X)

L = Z.LE.2.5.OR.I.NE."NO"
```

The following chart is a synopsis of the rules for the assignment of an expression to a variable.

Type of Variable	Type of Expression: INTEGER	REAL	DOUBLE PRECISION	COMPLEX	DOUBLE PRECISION COMPLEX
INTEGER	evaluate and assign	fix and assign	fix and assign	ILLEGAL	ILLEGAL
REAL	float and assign	evaluate and assign	truncate mantissa and assign	ILLEGAL	ILLEGAL
DOUBLE PRECISION	float to 56 bits and assign	extend mantissa and assign	evaluate and assign	ILLEGAL	ILLEGAL
COMPLEX	float, set imaginary part and assign	set imaginary part to zero and assign	truncate mantissa, set imaginary part to zero, and assign	evaluate and assign	truncate and assign
DOUBLE PRECISION COMPLEX	float, set imaginary part to zero, and assign	extend mantissa, set imaginary part, and assign	set imaginary part to zero, and assign	extend mantissa by 32 bits of zeroes	evaluate and assign

CHAPTER 5

CONTROL STATEMENTS

DEFINITION

Statements in a FORTRAN program are normally executed sequentially. Control statements allow the programmer to change the flow of program logic.

UNCONDITIONAL GO TO STATEMENT

Format:

```
GO TO n
GO TO v
```

where: n is a statement number.

v is a non-subscripted integer variable.

Format 1 of the statement causes control to transfer to the statement numbered n.

Format 2 of the statement causes transfer to the address which is the current value of integer variable v. This value must have been preset by an ASSIGN statement.

```
   GO  TO 25
        .
        .
        .
25 CONTINUE
```

Control is transferred to statement 25.

In all GO TO statements, "GOTO" may be written with blanks between "GO" AND "TO".

COMPUTED GO TO STATEMENT

Format:

```
GO TO(n1, n2,. . . , nm), v
```

where: n_1, n_2, . . . , n_m are statement numbers.

v is a non-subscripted integer variable name. (Note that the comma separating the right parenthesis from v is not required.)

COMPUTED GO TO STATEMENT (Continued)

Control is transferred to $\underline{n}_i$, where i is the value of $\underline{v}$. If $\underline{v} > \underline{m}$ or $\underline{v} < 1$, the GO TO statement is not executed and a fatal run-time error will result.

```
GO TO (10, 100, 40, 25,9), K
```

K must evaluate to 1, 2, 3, 4, or 5

ASSIGNED GO TO STATEMENT

Format:

```
GO TO v, (n1, n2, . . . , nm)
```

where: $\underline{n}_1, \underline{n}_2, \ldots, \underline{n}_m$ are statement numbers.

$\underline{v}$ is a non-subscripted integer variable name appearing in a previously executed ASSIGN statement. (Note that the comma separating the $\underline{v}$ from the left parenthesis is not required.)

The statement causes control to transfer to the statement whose number was last assigned to $\underline{v}$ by an ASSIGN statement. (This statement is accepted by DGC FORTRAN IV but is treated as an unconditional GO TO, i.e., the list $(\underline{n}_1 \ldots \underline{n}_m)$ is superfluous.)

ASSIGN STATEMENT

Format:

```
ASSIGN n TO v
```

where: $\underline{n}$ is a statement number.

$\underline{v}$ is a non-subscripted integer variable name that appears in an assigned or unconditional GO TO statement.

The statement causes a subsequent assigned GO TO statement to transfer control to the statement numbered $\underline{n}$.

```
ASSIGN 5 TO J
      .
      .
      .
GO TO J
      .
      .
      .
GO TO J, (25,16,5,40)
```

Control is transferred to the statement numbered 5 when the unconditional GO TO or when the assigned GO TO is executed.

ARITHMETIC IF STATEMENT

Format:

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

where: e is an integer, real , or double precision expression.

n_1, n_2, n_3 are statement numbers.

The expression is evaluated. Control transfers to statement n_1 if the value of the expression is less than zero. Control transfers to statement n_2 if the value of the expression is zero, and control transfers to statement n_3 if the value of the expression is greater than zero.

```
IF (A(J,K)-B) 10,4, 30
IF (Q*R) 5, 5, 2
```

LOGICAL IF STATEMENT

Format:

```
IF (le) s
```

where: le is a logical expression.

s is any executable statement (assignment statement, control statement, or I/O statement) except a DO.

The logical expression is evaluated. If the expression is true, statement s is executed. Control then passes to the next statement following the logical IF unless statement s transfers control.

If the expression is false, statement s is bypassed and control passes to the next sequential statement.

```
IF (A.AND.B) F=SIN(R)
IF (I.GT.0) GO TO 25
```

CALL STATEMENT

Format:

```
CALL subr (a1, a2,. . . , an)
CALL subr
```

CALL STATEMENT (Continued)

where: subr is the name of a subroutine or a dummy variable (see EXTERNAL)

$a_1, a_2, \ldots, a_n$ are actual argument names that replace dummy argument names in the subroutine.

The statement references the designated subroutine, which is executed. Control is returned to the statement after the CALL statement when execution of the subroutine is completed unless the subroutine makes an abnormal return.

Arguments of subroutines are described in the section dealing with subprograms.

```
CALL QUAD (9.73,Q/R,5,R-S**2.0,X1,X2)
CALL OPTIONS
```

RETURN STATEMENT

Format:

```
RETURN
RETURN v
```

where: v is a dummy integer variable whose value represents a statement number in the calling program.

The statement marks the logical end of a subprogram. Execution of a return without v is a normal return. Control is returned to the calling program as follows:

1) Return from a subroutine is made to the statement following the CALL statement.

2) Return from a function is made to the statement containing the function reference and a value is substituted for the function in that statement.

An abnormal return (RETURN v format) allows for error returns or multiple-decision branches, and is described further in Chapter 9, "Abnormal Returns." An abnormal return must return to the immediately calling program.

CONTINUE STATEMENT

Format:

```
CONTINUE
```

CONTINUE is a dummy statement that causes continuation of the normal execution sequence. It is most frequently used as the last statement in the range of a DO to provide a transfer address for IF and GO TO statements that are intended to begin another repetition of the DO range.

CONTINUE STATEMENT (Continued)

```
      S = 0
   5  DO 15 I = 1, N
      IF (B(I)-1000.) 10,15,15
  10  S = S+C(I)*B(I)
  15  CONTINUE
```

PAUSE STATEMENT

Format:

```
PAUSE

PAUSE s
```

where: s is a string of ASCII characters which will be typed out following the pause.

The statement causes the program to cease executing. A message, indicating a pause and giving the text string at the pause, will be printed at the console printer. To resume execution at the point of interruption, the programmer presses any console key.

STOP STATEMENT

Format:

```
STOP

STOP s
```

where: s is a string of ASCII characters.

The statement causes unconditional termination of program execution. A message indicating a stop and giving the text string, if present, will be printed at the console printer.

```
STOP LABEL 70
```

DO STATEMENT

Format:

```
DO n i = m1, m2, m3

DO n i = m1, m2
```

DO STATEMENT (Continued)

where: $\underline{n}$ is a statement number.

$\underline{i}$ is a nonsubscripted integer variable name called the control variable.

$\underline{m_1}$, $\underline{m_2}$, $\underline{m_3}$ are integer constants or nonsubscripted integer variable names. They are the initial parameter, final parameter, and incremental parameter respectively of $\underline{i}$. Default value of $\underline{m_3}$ is 1. $\underline{m_3}$ must be greater than or equal to 1.

The DO statement sets up a loop. Statements following the DO statement up to and including the statement labeled $\underline{n}$ can be repetitively executed. This set of statements is called the range of the DO.

The parameters indicate the values that control variable $\underline{i}$ may assume within the range of the DO. $\underline{m_1}$ is the starting value of $\underline{i}$; $\underline{m_2}$ is terminal value (or value beyond which $\underline{i}$ cannot assume values); and $\underline{m_3}$ is the value by which $\underline{i}$ is incremented at each execution of the loop. A simple DO loop is:

```
      DIMENSION A(100)
        .
        .
        .
      SUMSQ = 0.0
      DO 25 I = 1,100
25    SUMSQ = SUMSQ + A(I)**2
```

The DO range, which is assignment statement 25, is used to form the sums of squares of the elements of array A:

$$\sum_{i=1}^{100} A(i)^2$$

DO loops can be nested. The range of a nested DO cannot extend beyond the range of an outer DO loop. Following is an example of the summation of values of an integer, two-dimensional array. Both the nested and outer DO loops terminate at dummy statement CONTINUE.

```
      INTEGER SUM, MATRIX (10, 20)
        .
        .
        .
      SUM = 0
      DO 30 I = 1, 10
      DO 30 J = 1, 20
      SUM = SUM + MATRIX (I, J)
30    CONTINUE
```

DO loops have the following restrictions:

1) Control cannot be transferred into the range of a DO. (Control can be transferred out of the DO range.)

2) The statement terminating the range of the DO cannot be a GO TO of any form, an arithmetic IF, RETURN, STOP, PAUSE, DO, or a logical IF containing any of these statements.

3) The control variable cannot be redefined within the DO range.

If DO loop conditions are satisfied by the control variable reaching its final parameter value, the control variable becomes undefined and the DO loop is exited by executing the next statement following the statement labeled $\underline{n}$.

DO STATEMENT (Continued)

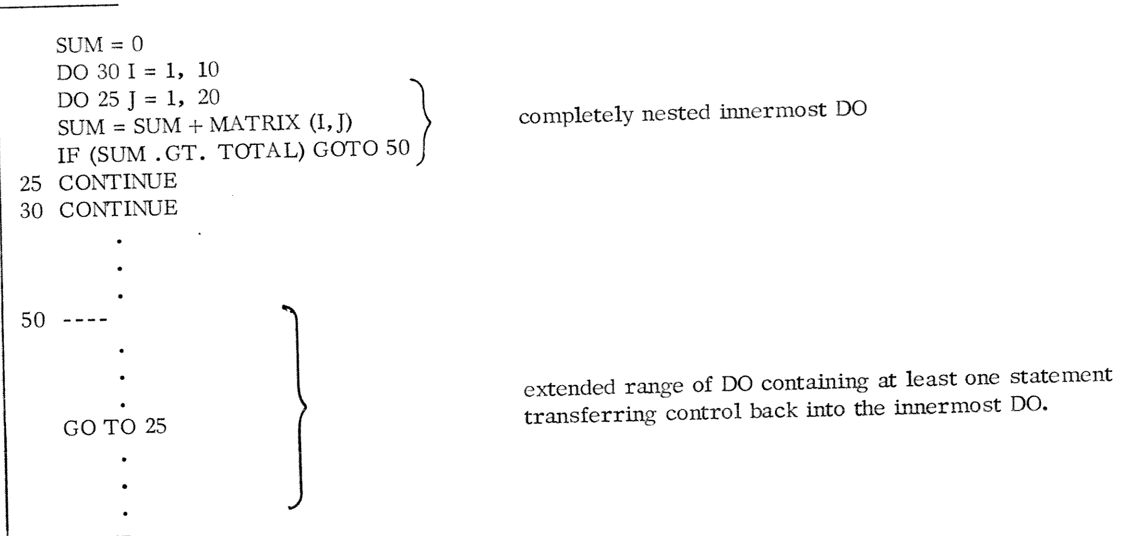
If a DO loop is exited by execution of a GO TO or arithmetic IF statement, the control variable remains defined. Its value is equal to its value at the time of exit from the loop via the GO TO or arithmetic IF.

The range of a DO loop can be extended to include additional statements or program units if:

1) A statement (GO TO or arithmetic IF) exists in an innermost, completely nested loop of the DO loop that transfers control out of the innermost, completely nested loop, and

2) A statement transferring control back into the innermost, completely nested loop exists and might logically be executed as part of the extended range.

The extended range includes all statements that might logically be executed outside the innermost, completely nested loop (including the transfer statement).



CHAPTER 6

INPUT/OUTPUT STATEMENTS

FORTRAN INPUT/OUTPUT

FORTRAN IV provides for five types of input/output:

1) Unformatted values: Externally recognizable numbers in ASCII (teletype standard) code can be read and converted to their internal computer representation and vice versa.

2) Formatted values: On input, external values in ASCII code are interpreted according to a FORMAT specification. On output, internal values are displayed or stored on the external medium according to a FORMAT specification.

3) Binary values: Internal data is transferred to an external device or vice versa with absolutely no change in structure (representation).

4) Run-time values: Conversational input/output is permitted from the teletypewriter, in accordance with programmed ACCEPT and TYPE statements.

5) Control: Positioning of devices and end of input and output are directed by specific statements (REWIND and ENDFILE) or by subroutine CALLs.

PROGRAMMED I/O USING READ AND WRITE

The two basic FORTRAN IV input/output statements are READ and WRITE. These statements are both taken from the computer's viewpoint, i.e., the computer READs into itself from an external device and WRITEs out to an external device.

General forms of the READ and WRITE statements for ASCII mode are:

```
READ | WRITE    (channel) { list}

READ | WRITE    (channel, format) { list}
```

The first form is used for unformatted I/O and the second for formatted I/O.

In the statement formats:

channel is an I/O channel number associated with the file or device. There are 64 channels ($0-63_{10}$). See page 6-23 for a list of devices and files associated with pre-assigned channels and for information on programmer assignment of channels.

format is the number of the associated FORMAT statement or the name of an array containing the format specification.

list is a list of names of variables which are to be given values (READ) or whose values are to be written (WRITE). If list is not given, READ will read and ignore an entire record while WRITE will output any Hollerith information in the FORMAT statement, perform any carriage control specified and write a record.

I/O LISTS OF READ AND WRITE STATEMENTS

The I/O list contains the names of variables, including arrays and array elements, which are to be given values or whose values are to be written. Array elements must be specified with unsubscripted integer variables and/or constants as subscripts.

If a WRITE statement is unformatted, the I/O list may contain Hollerith strings to be written verbatim as they are encountered in the list.

An I/O list may be written as a simple list of variables; in addition, an I/O list can specify effective DO loops with reference to all or a portion of the list of variables. These are called DO-implied lists. The form of the DO-implied list is:

```
(list, i=m1, m2 [,m3])
```

where: $\underline{i}$ is the control variable and must be an unsubscripted integer variable.

$\underline{m}_1$ is the lower bound; $\underline{m}_2$ is the upper bound; and $\underline{m}_3$ is the increment of i. $\underline{m}_1$, $\underline{m}_2$, and $\underline{m}_3$ must be integer constants or unsubscripted integer variables. If $\underline{m}_3$ is not given, the increment is +1.

Some examples of I/O lists are:

```
READ(13,5) G, B(1), C, B(2), D, B(3)

WRITE (12, 101) A, B, I
```

```
DIMENSION A(3,4)
     .
     .
     .
READ (11, 5) A          The READ statement reads in the entire array and is the same as:

READ (11,5) A(1,1),A(2,1),...,A(3,4)
```

For punctuation purposes, any portion of an I/O list can be enclosed in parentheses, except within the loop specification of a DO-implied loop, i.e.,

```
WRITE (12) I, A, A(I,J)        }
WRITE (12) (I),A,A(I,J)        }
WRITE (12) (I,A),(A(I,J))      }   are all equivalent
WRITE (12) ((I,A),(A(I,J)))    }
```

I/O LISTS OF READ AND WRITE STATEMENTS (Continued)

The DO-implied list affects the transfer of its associated list of variables in much the same way that the DO statement affects the range of the DO. Some examples of DO-implied lists are:

```
READ (13, 20)A, B, (C(I), I=1, 3)

     is equivalent to

READ(13, 20)A, B, C(1), C(2), C(3)
```

```
WRITE (10, 20) (A, B, C, D, I=1, 2)

     is equivalent to

WRITE (10, 20)A, B, C, D, A, B, C, D
```

```
READ (11, 20) (C(I, I),  I=1, 4, 1)

     is equivalent to

READ (11, 20) C(1, 1), C(2, 2), C(3, 3), C(4, 4)
```

Note that a DO-implied list must be enclosed in parentheses.

DO-implied lists may be nested to any depth. A comparison of examples of nested DO-implied lists with nested DO loops will indicate how the nested DO-implied lists are interpreted.

```
READ(13, 25)((A(I, J), J=1, 4), I=1, 9, 2)

    The equivalent DO statements would be:

    DO 20 I = 1,  9,  2
    DO 20 J = 1,  4
 20 READ (13, 25) A(I, J)

    The equivalent simple list would be:

READ (13, 25)A(1, 1), A(1, 2), A(1, 3), A(1, 4),
             A(3, 1), A(3, 2), A(3, 3), A(3, 4),
           .
           .
           .
             A(9, 1), A(9, 2), A(9, 3), A(9, 4)
```

UNFORMATTED I/O

The spacing of output, when unformatted, provides an 8-character field for integer and logical data, a 16-character field for real data, a 32-character field for double precision or complex data, and a 64-character field for double precision complex data. A carriage return is inserted when the next value to be output would make the line longer than 72 characters. Some examples of unformatted WRITE statements and their possible output are:

```
I = 7
WRITE(12) I
```

Channel 12, by default, is the line printer. The above causes printing of the line:

```
△△△△△△△7
```

```
R = 7.1
WRITE (12) R, I
```

causes printing of the line:

```
△△△△0.710000E△△1△△△△△△△7
```

```
    DIMENSION A(3,2)
      .
      .
      .
    R = 7.1
    DO 2 I = 1,3
    DO 2 J = 1,2
  2 A(I,J) = I+(J-1)
    WRITE (12) A(3,1), R, A(1,1)
```

causes the printout, where I+(J-1) has been floated before assignment to A(I,J), of the following:

```
△△△△0.300000E△△1△△△△0.710000E△△1△△△△0.100000E△△1
```

The same array A with the output statement as:

```
WRITE (12)"ARRAY A:  <15> ",A
```

causes the printout:

```
ARRAY A:
△△△△0. 100000E△△1△△△△0. 200000E△△1△△△△0. 200000E△△1△△△△0. 300000E△△1
△△△△0. 300000E△△1△△△△0. 400000E△△1
```

If the output statement is:

```
WRITE (12) ((J, K, A(J, K), " <15 >", K=1, 2)J=1, 3)
```

the following printout results:

```
△△△△△△△1△△△△△△△ 1△△△△0. 100000E△△1
   |        1   |      2  |   0. 2          |
   |        2   |      1  |   0. 2          |
   |        2   |      2  |   0. 3          |
   ↓        3   ↓      1  ↓   0. 3          ↓
△△△△△△△3△△△△△△△ 2△△△△0. 400000E△△1
```

Free form or unformatted READ uses an I/O list to determine the order of input exactly as unformatted WRITE uses it for output.

On input, the programmer distinguishes individual data by separating the data with commas or end-of-record indicators (carriage return from the teletype.) Thus to fill array A of six elements the FORTRAN program will read the teletype (channel 11):

```
READ (11) A
```

The programmer can satisfy the READ by typing:

```
1, 2, 3, 4, 5, 6 )

        or by typing:

1, 2, 3 )
4, 5 )
6 )

        or by typing

1, 2, 3.1,  -5E2, 0, .1E-3 )
```

The READ will convert data types from integer to floating point or vice versa if required by the internal data type.

FORMATTED I/O

In DGC FORTRAN IV, the specification of format serves two basic purposes:

1) For input (READ), formatting allows the data to be represented compactly and in a form suitable for ready duplication of large quantities of input from a card or card image medium.

2) For output (WRITE), formatting allows precise control of the layout of the data as it will appear on the printed page.

Formatting specifications allow the programmer to control field width allotted to any datum, the spacing between fields, the assignment of data to particular records or lines, and the notation in which the data will be represented externally.

The specification of format can be given in a FORMAT statement or can be contained in an array that is read into at run-time.

FORMAT Statement

A FORMAT statement has the form:

```
n FORMAT (specification)
```

where: n is a statement number that appears in the READ or WRITE statement that references the format specification.

specification is the list of field descriptors, field separators, Hollerith strings, repetition constants, etc., that together define the formatting of the data being input or output.

The field descriptors of a FORMAT statement are associated with the variables appearing in corresponding order in the list of a READ or WRITE statement.

```
20 FORMAT (F10. 2, E15. 5)
   WRITE (12, 20) A, B
```

In the example, the WRITE statement references FORMAT statement 20. Variable A is associated with the field descriptor F10. 2, and variable B is associated with the field descriptor E15. 5.

Specification of Format Information - General

The specification allows the programmer to describe the format of all forms of numeric data and string data, to include Hollerith constants, to set tabular output, to control vertical spacing of output, etc.

Essentially, the specification consists of one or more field descriptors that must be separated unambiguously.

Separators of Descriptor Fields

The following are used to separate field descriptors in a format specification:

Commas can be used to separate field descriptors within a single unit record. A unit record is generally defined in terms of the device being accessed -- end of lines on teletype or printer, 80 columns on a card, etc.

No separator is needed in DGC FORTRAN IV if two field descriptors can be identified unambiguously.

Slashes separate field descriptors at the termination of a unit record.

Repetitive slashes can be used to indicate empty unit records, for example, lines to be skipped on the teletype or the line printer.

```
21  FORMAT (I4, E15.5)
```

comma between 2 numeric descriptor fields.

```
22  FORMAT (I4 "DATA IS: " 4E15.5)
```

quotes set Hollerith string off unambiguously from preceding and following descriptors.

```
23  FORMAT  (I4, 4E15.5///5F10.2)
```

if output to TTO or LPT, triple space before output of data represented by descriptor following slashes.

Basic Numeric Field Descriptors

The basic field descriptors to handle numeric data have the following formats:

Iw	-	Integer
Fw.d	-	Floating Point
Ew.d	-	Explicit exponent floating point
Dw.d	-	Double precision floating point
Gw.d	-	Generalized floating point

where: w is the field width given in character positions.

d is the number of digits after the decimal point in real and double precision data (except for G output conversion, described later.)

Complex data is represented by two real (F, E, G) descriptors. Double precision complex data is represented by two double precision (D) descriptors.

Numeric Conversion on Input

Blanks are ignored for all purposes other than field width count, unless they are between two digits or between a digit and a decimal point; in those cases they are treated as zeroes.

FORMATTED I/O (Continued)

Numeric Conversion on Input (Continued)

Any decimal point in the datum will override the position given for the decimal point in the FORMAT descriptor.

All real or double precision data (F, E, G, or D conversion) may have the following forms:

1) A string of digits optionally signed, containing an optional decimal point, e.g.,

   -33.456   67321   7890.001

2) A string of digits as above, followed by an exponent of one of the forms:

   Signed integer constant: +44.5+05
   E followed by signed integer constant: 673E+04
   D followed by signed integer constant: 789.1D-01
   E followed by unsigned integer constant: 90.E03
   D followed by unsigned integer constant: -25D02
   (D and E are equivalent forms)

The field width w always represents the exact number of characters in the external datum on input. This includes decimal point, sign if any, and any leading blanks.

```
15 FORMAT (I3,F7.2,E13.3,G9.1,D16.7)

△22△+25.65△-4.22201E-01△△7654321△△△△-67567567-02

 I3     F7.2     E13.3     G9.1     D16.7
 w=3    w=7      w=13      w=9      w=16
```

Output Conversion of Integers

The integer is right justified in the field w, and is signed if negative. If the field width is not wide enough to output the datum, an * is output, followed by as many digits of the number as will fit.

```
10 FORMAT (I3, I4, I3, I6)

△△4△-33*21△△-388

I3  I4  I3  I6
```

Output Conversion of Real and Double Precision Data

For all numeric conversions, the datum is right justified in the output field with leading blanks, if needed. Negative data are signed, and the decimal point will occupy the position determined by the decimal indicator d in E, F, G, and D conversions.

If the field width w is not wide enough to output the datum, an * is output followed by an many digits of the number as will fit.

Output Conversion of Real and Double Precision Data (Continued)

F conversion causes output of a real number, signed if negative. Example:

```
2 FORMAT (F10.2)

△△-2107.99
```

E conversion outputs a real number, signed if negative, as a fraction and an exponent, with the rightmost four character positions reserved for an exponent of the form:

E△ee	E-ee	D△ee	D-ee

```
3 FORMAT (E16.8)

△△△.10001110E△03
```

D conversion outputs a double precision number, signed if negative, with the rightmost four character positions reserved for an exponent of the same form as that for E conversion. (The choice of D or E as the fourth from the rightmost character position depends upon the internal data type.)

```
4 FORMAT (D25.18)

△-.212212211000005000D△07
```

In the G conversion format, d represents the number of significant digits in the external field. Output of G conversion is either in E format or F format depending upon the magnitude of the stored datum. The output is in E format, except when the magnitude of the datum, N is:

$$.1 \leq N < 10^{d}$$

Within that range F conversion is used according to the following formula:

Magnitude of Datum	Conversion
$0.1 \leq N < 1$	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

Output Conversion of Real and Double Precision Data (Continued)

For example:

```
Stored data:            90          9000

Format:                 FORMAT (2G9.3)

External                △90.0△△△△△.900E△04
Representation:         \______/\_______/
                           90      9000
```

Radix 8 Input/Output Using the O Specifier

The I, F, D, E, and G descriptors, when preceded by the letter O, do a radix 8 (octal) conversion of the same form as they normally do a radix 10 conversion. Note, however, that exponents in all cases for the D, E, and G specifiers will be decimal, not octal.

For example:

```
100 FORMAT (2I3, 2OI3, E13.7, OE13.7)
    WRITE (2,100) I1, I2, I1, I2, R1, R1
```

where: I1 = 20_{10}, I2 = 8, R1 = $.125*10^9$, the output will be:

△20△△8△24△10△.1250000E△09△.1000000E△09

Non-numeric Field Descriptors

In addition to numeric field descriptors, the following descriptors are used:

Lw	-	Logical
Aw	-	Alphabetic
Sw	-	String with maximum width w
Tw	-	Tabulate to position w
nX	-	Leave n blank character positions
"string"	-	ASCII character string
'string'	-	ASCII character string
nHstring	-	ASCII character string of n characters
Z	-	Suppress output of the carriage return at the end of a record.

Input and Output of Logical Data (Lw)

On input T or F as the first non-blank character in the field determines the value. T stores a -1 (177777) word (true) and F stores a word of all zeroes (false). On output T or F is right justified in field w.

FORMATTED I/O (Continued)

Input and Output of Logical Data (Lw) (Continued)

```
5 FORMAT (L3)

ΔΔT
```

Positioning Descriptors (nX, Tw)

The nX descriptor can be used on both input and output. On input n characters of the external record will be skipped. On output, n blank spaces will precede the next datum.

The tabular descriptor Tw is used on output to cause tabulation to the character position given by w. If the carriage is currently positioned beyond the value of w, the descriptor is ignored.

```
8 FORMAT (10X, I4, T25, I4)

ΔΔΔΔΔΔΔΔΔΔ-456 ΔΔΔΔΔΔΔΔΔΔ-789
```

String Data (Sw)

In Nova line computers, characters are stored two per word, and when read in, a character string is always terminated by a null byte (8 bits of zero). In Sw format w represents the number of characters to be read or written. A maximum field width of 80 characters is allowed. On input, w characters are read to an associated single variable* (not array) in the I/O list, with as many words used as are needed to store w characters, followed by a terminating null byte. If the record read does not contain w characters only those characters read are stored and terminated by a blank. Use of the null byte/blank may increase the number of words required to store a string as shown in the examples following:

External datum:	ΔΔ@Δ$2	
S2 stores:	ΔΔ	in two words
S3 stores:	ΔΔ@	in two words
S4 stores:	ΔΔ@Δ	in three words
S6 stores:	ΔΔ@Δ$2	in four words
S8 stores:	ΔΔ@Δ$2	in four words

On output, if the length of the string is n characters, characters will be written as follows:

w = n	entire string is written out
w > n	entire string is written out, followed by w-n spaces.
w < n	first w characters are written out.

*A single variable includes an array element. To input or output a string variable to or from an array, specify the initial array element.

FORMATTED I/O (Continued)

String Data (Continued)

Internal string:	△NOW△IS△THE△TIME
S16 produces:	△NOW△IS△THE△TIME△△△△
S20 produces:	△NOW△IS△THE△TIME△△△△△△△△
S11 produces:	△NOW△IS△THE

String Literals

ASCII character literals may be read or written, using one of the string literal forms:

nHstring   "string"   'string'

"THIS△IS△A△STRING."
'THIS△IS△A△STRING.'
17HTHIS△IS△A△STRING.
} equivalent string literal formats.

Use of delimiting quotation marks or apostrophes eliminates the need for counting characters, required in the nH format.

An apostrophe cannot appear within a string delimited by apostrophes. A quotation mark cannot appear within a string delimited by quotation marks.

Alphabetic Data (Aw)

ASCII characters can be read or written using the Aw format descriptor. On both input and output, w represents the field width on the external device.

On input, since the computer stores two characters per word, a limit of two characters can be read to a single variable. The variable should be typed INTEGER (or LOGICAL). The rightmost two characters in the field w will be stored. If w is 1, one character will be stored in the left half of a word and a blank stored in the right half. To store a series of characters in contiguous locations, an integer array variable and repetitive Aw formats can be used.

If the next input for processing is: △△@△$2

A4 stores:	@△
A2 stores:	△△
A3 stores:	△@
A6 stores:	$ 2
2A2 stores:	△△ and @△ in next two variables of the I/O list
3A2 stores:	△△ and @△ and $ 2 in next three variables of the I/O list
3A1 stores:	△△ and △△ and @△ in next three variables of the I/O list

Alphabetic Data (Continued)

On output, the characters are right-justified with leading blanks, if any. If the field width is less than two, the leftmost character will be represented with truncation to the right.

```
6 FORMAT (A6)
  WRITE (12,6) B        where B contains HOUR
△△△△HO                 representation on external device
```

Note that all four characters (HOUR) could have been output using 2A2 as the format specification.

Repetition Constant

One field descriptor or group of field descriptors can be preceded by an integer, called a repetition constant. The field descriptor or group of field descriptors will be repeated the number of times indicated by the integer.

All numeric field descriptors and the Aw and Lw descriptors can be preceded by repetition constants. The remaining non-numeric descriptors cannot have repetition constants.

An example of repetition of individual field descriptors is:

```
9 FORMAT (2I2, 3F11.2)
        which is the same as:
9 FORMAT (I2, I2, F11.2, F11,2, F11.2)
```

If a group of two or more field descriptors are enclosed in parentheses, the entire group can be preceded by a repetition constant. For example, the specification:

```
10 FORMAT (I2, 3(E14.5, L1) )
        is the same as:
10 FORMAT (I2, E14.5, L1, E14.5, L1, E14.5, L1)
```

Individual and group repetition constants can be combined in a given format; for example:

```
11 FORMAT (G13.2, 2(F10.1, 3I4 ) )
        is the same as:
11 FORMAT (G13.2, F10.1, I4, I4, I4, F10.1, I4, I4, I4 )
```

Multiple Record Forms

The statement

```
10  FORMAT (I2, 3F12.1)
```

can be used to transmit more than four items of data. Each record (or output line) would consist of four data. For example:

```
    WRITE (12, 10) I, A, B, C, J, D, E, F

10  FORMAT (I2, 3F12.1)
```

might produce:

```
△4△△△3456798.6△△△4545551.1△△33333366.7
△2△△△△△99999.2△△△△△△△112.3△△△△900785.4
```

The FORMAT specification may have two or more different record formats. They are separated by slashes. For example:

```
    WRITE (12, 10) I, A, B, C, J, D, E, F

10  FORMAT (I2, 3F12.1/I4, 3F12.1)
```

would affect the same data as follows:

```
△4△△△3456798.6△△△4545551.1△△33333366.7
△△△2△△△△△△99999.2△△△△△△△112.3△△△△900785.4
```

If the list of the WRITE statement above has 16 variables, then the first and third lines would be output in the same format and the second and fourth lines would be output in the same format. Record processing thus returns to the delimiting left parenthesis when the format descriptors are exhausted.

If multiple-line format is desired, the second record specification is enclosed in parentheses. Multiple-line format is where the first line is printed in a given format while the remaining lines are printed in another format. This is done without returning to the first left parenthesis when the list has been exhausted. For example:

Multiple Record Forms (Continued)

```
   WRITE (12, 10) I, A, B, C, J, D, E, F

10 FORMAT("RESULTS IN INCHES"/(I5, 3F12. 1))
```

would produce the following:

```
RESULTS IN INCHES
△△△△4△△△3456798. 6△△△4545551. 1△△33333366. 7
△△△△2△△△△△99999. 2△△△△△△△112. 3△△△△900785. 5
```

Additional slashes will cause vertical lines (records) to be skipped. For example:

```
   WRITE (12, 10) I, A, B, C, J, D, E, F
10 FORMAT("RESULTS IN INCHES"/ / (I5, 3F12. 1))

RESULTS IN INCHES

△△△△4△△△3456798. 6△△△4545551. 1△△33333366. 7
△△△△2△△△△△99999. 2△△△△△△△112. 3△△△△900785. 4
```

When parentheses are nested in a FORMAT statement, they are assigned level numbers, with the outermost parentheses assigned level 0. For example:

```
10 FORMAT (3E10. 3, (I2, 2(F12. 4, F10. 3)), D20. 12)
          0         1  2                 21        0
```

If data items remain to be transmitted after the descriptors in a multiple level FORMAT statement have been "used", the format is repeated from the last previous parenthesis that is a level zero or a level 1 left parenthesis. In the FORMAT statement above, the format would be repeated beginning at I2, the first descriptor following a level 1 left parenthesis.

Vertical Carriage Control

The first character of formatted output is a vertical carriage control character. The control characters recognized are:

0	-	double space before printing
1	-	form feed before printing

Carriage control characters are normally placed at the beginning of unit records in the FORMAT specification. One of the string descriptors can be used to insert the carriage control character.

Vertical Carriage Control (Continued)

```
5  FORMAT (1H1, 4E15.5/("0", F11.2, 4E15.5))
             ↑            ↑ ↑
            form          | double space
            feed       single space
```

When the first character of formatted output is part of a datum associated with a numeric field descriptor, it will be interpreted as a carriage control character.

```
15  FORMAT (I2)
```

If the datum associated with I2 is 15, a form feed is given and 5 is printed. If the associated datum is 05, a double space is given before 5 is printed. If the associated datum is △ 5, the normal single carriage return/line feed occurs before 5 is printed.

The Z field descriptor can be used to suppress carriage return on output. The Z descriptor should always be the last descriptor in the FORMAT statement and will suppress the carriage return when writing the record.

```
   I = 3
   J = 4
3  FORMAT (1X, I6)
   WRITE (12, 3) I
   WRITE (12, 3) J
```

will print:

```
△△△△△3
△△△△△4
```

but

```
   I = 3
   J = 4
3  FORMAT (1X, I6, Z)
   WRITE (12, 3) I
   WRITE (12, 3) J
```

will print

Vertical Carriage Control (Continued)

```
△△△△△3△△△△△4
```

The Z descriptor should only be used with WRITE statements.

Scale Factor

All floating point numeric conversions (F, E, D, G) can be preceded by a scale factor of the form:

```
nP
```

where: n is a signed or unsigned integer.

A scale factor precedes the basic field descriptor and any repetition constant. Once a scale factor is given for a field descriptor it remains in effect for all F, E, G, and D conversions in the FORMAT statement, unless changed. For example:

```
10 FORMAT (3PF9.3, 2E15.1)

    is the same as:

10 FORMAT (3PF9.3, 3PE15.1, 3PE15.1)
```

A scale factor of 0P is the same as no scale factor. For example:

```
11 FORMAT (0PG10.2)

    is the same as:

11 FORMAT (G10.2)
```

The effect of a scale factor on a datum varies with the datum, the type of conversion (F, E, D, or G), and the direction (input or output).

On input if the datum has an explicit exponent, the scale factor has no effect. This is true for all conversion formats: E, F, G, or D.

On input if the datum has no explicit exponent, the scale factor conversion formula is:

input datum x 10^{-n} = internal representation

Scale Factor (Continued)

For example:

External data:	-25.44	345.71
Format:	15 FORMAT (2PF10.2, G8.2)	
Data stored: (decimal representation)	-.2544	3.4571

On output using E or D conversion, the real constant portion of the stored value is multiplied by 10^{n} and n is subtracted from the exponent portion of the stored value. This means that the value remains the same although formatted differently, e.g.:

Stored data:	9000	9000
Format:	14 FORMAT (E13.4, 2PE13.4)	
External Representation:	ΔΔΔΔ.9000EΔ04ΔΔ90.0000EΔ02	

When G conversions are in E format, they also follow the formula above.

On output on F conversion, the stored value is multiplied by 10^{n}, actually altering the external value, e.g.:

Stored data:	9000	9000
Format:	16 FORMAT (F10.2, -4PF10.2)	
External Representation:	ΔΔΔ9000.00ΔΔΔΔΔΔΔ.90	

The scale factor has no effect on G conversion within the F range. G conversion thus always transfers the value unchanged whether the F or E format is chosen.

Run-Time Format Specifications

I/O statements can reference an array containing a formatting specification, rather than a FORMAT statement. This allows formatting information to be read in at run-time and changed for different data.

The formatting array contains a format specification, including the zero-level left and right parentheses, but not the word FORMAT. The closing right parenthesis must be followed by an exclamation point (!). The character string that is the format specification can be stored in an array by use of a formatted READ that uses a format containing Aw or Sw descriptors.

FORMATTED I/O (Continued)

Run-Time Format Specification (Continued)

To use run-time formatting, the user must:

1) Determine how large an array will be needed for the largest incoming format specification. If core space is not critical, the user can estimate.

2) Dimension the array in a DIMENSION, COMMON, or type declaration statement.

3) Include an appropriate storage statement or statements. Most commonly, this will be a READ statement and a FORMAT statement that will read the format specification into array storage using A_w_ or S_w_ descriptors.

4) Reference the format array in the READ statement used for input of data.

5) Supply formatting information to be read into the array at run-time.

For example using the A_w_ format:

```
   DIMENSION FT (12)
 2 FORMAT(24A2)
   READ (11,2) (FT(I), I = 1, 12)
   READ (11, FT) J, W, X, Y, Z, (C(I), I = 1, 7)
```

Or using the S_w_ format:

```
   DIMENSION FT (12)
 2 FORMAT (S47)
   READ (11, 2) FT (1)
   READ (11, FT) J, W, X, Y, Z, (C(I), I = 1,7)
```

The information supplied at run-time might be:

```
(I3, 4E15.6/7F10.3)!
```

The number of characters to be stored in the example is 20, including blanks, well below the 48-character maximum allowed in array FT.

BINARY I/O

Binary data can be transferred to and from an external medium using the statements:

BINARY I/O (Continued)

```
WRITE BINARY (channel) list

READ BINARY (channel) list
```

where: channel and list are the same as for ASCII mode

In accordance with the I/O list, data is transferred at two bytes per word, where the number of words transferred depends upon the internal data representation: 1 word for integer, 2 for real, 4 for double precision and complex, and 8 for double precision complex. The high order or left byte is transferred first.

CONSOLE INPUT AND OUTPUT

ACCEPT and TYPE Statements

Unformatted I/O on the console frees the user from the details of FORMAT specifications while providing for legible documents and easy-to-use I/O statements.

The statements ACCEPT and TYPE are used with console input and output respectively. The format of ACCEPT is:

```
ACCEPT list
```

where: list is a list of variables and, optionally, string constants. When the ACCEPT statement is executed, values for the variables of the ACCEPT list are input from the console. Any string constants given in the list of the ACCEPT statement are output at the console and can serve as a guide as to what input value is required.

The format of the TYPE statement is:

```
TYPE list
```

where: list is a list of variables for which values are to be output when the statement is executed and, optionally, string constants to be output.

Note that if channel 10 is to be reassigned via OPEN or FOPEN, either no TYPE or ACCEPT statements should be used or channel 10 must be closed. The following code will produce a fatal run-time error (illegal channel number):

```
TYPE "STARTED"
CALL OPEN (10, "FILE",. . . )
```

Channel 10 is already open to $TTO at this point; insertion of CALL CLOSE (10, IER) will prevent this problem.

Sample Program

A sample program using teletype I/O is shown below:

```
C    BENCHMARK TEST OF DGC FORTRAN
     DIMENSION RARRAY (2000)
     COMMON RARRAY, AUTOC, SD
1    ACCEPT "ARRAY SIZE=", IAS,
    1"INITIAL RANDOM NUMBER=", RN1
     IF(IAS-2000) 2,2,3
3    TYPE "ARRAY SIZE MAX IS 2000"
     GO TO 1
2    CALL RANDOM (RN1, IAS, RARRAY)
     CALL CORRELATE (IAS)
     TYPE "AUTOCORRELATION=",AUTOC,
    1" <15> ", "STANDARD DEVIATION=", SD
     PAUSE
     GO TO 1
     END
```

The teletype operation might appear as follows. Underscoring indicates computer output, (values not underscored are input by the programmer), and) stands for carriage return given by the programmer.

```
ARRAY SIZE = 500 )
INITIAL RANDOM NUMBER = .93826 )
AUTOCORRELATION = .73152E -△1
STANDARD DEVIATION = .201552E△△0
PAUSE
```

Rules of Teletype I/O

The following rules apply to input:

1) More than one value can be called for in an ACCEPT statement. The input values can be separated by commas or a carriage return.

2) Output of Hollerith strings can be mixed with input of data in the ACCEPT statement, providing for example, a guide as to what input value is required.

3) When Hollerith string output is interspersed with data, a carriage return must be given at the teletypewriter to force the next string to be output. For example, the carriage return after 500 is necessary to prompt the typing of "INITIAL RANDOM NUMBER=".

4) ACCEPT will convert integers to real or double precision if the data type of the internal variable requires.

Rules of Teletype I/O (Continued)

On output the TYPE statement provides the following field widths:

8	-	integer
16	-	real
32	-	double precision and complex
64	-	double precision complex

A carriage return is inserted when the next quantity will not fit on the current line. In either the ACCEPT or TYPE statement, a carriage return is output by "<15>". A form feed is output by "<14>". These characters must be the last characters in a Hollerith string since they cause the operating system to terminate output.

TYPE or ACCEPT statements also provide for transfer of whole arrays and array elements with integer variables or constant subscripts.

```
TYPE "RARRAY:  <15> ", RARRAY
```

causes the entire array, RARRAY, to be typed out. More reasonably:

```
 TYPE "THE", IAS, "RANDOM NUMBERS ARE  <15>",
1(RARRAY (I), I = 1, IAS)
```

outputs only that portion of RARRAY that is filled by subroutine RANDOM.

Note in the example that the DO-implied loop must be enclosed in parentheses and that a comma precedes the control variable, I.

DO-implied loops can be nested.

```
      DIMENSION A(3, 5)
      .
      .
      .
      ACCEPT ((A(I,J),  I = 1,3),J = 1,5)
C     VERIFY INPUT
      TYPE "J       I       VALUE     <15> ",
     1((J,I,A(I,J)), I = 1,3), J = 1,5)
```

I/O lists can contain all combinations of variables, arrays, array elements, Hollerith strings, and DO-implied loops, separated by commas.

CONTROL I/O

Channel Access

Files including devices, are associated with a channel number (0-63) before that file or device may be accessed. To open a file, an I/O statement must reference a pre-assigned channel from the list below or the file and channel must be associated by a call to FOPEN or OPEN (see pages 3-10 and 3-11, Part II).

Pre-assigned Channel #	Device Name	Device	
6	$PLT	Incremental Plotter	
8	$TTP	TTY punch	
9	$CDR	Card reader	
10	$TTO	TTY printer	$TTO1 in foreground
11	$TTI	TTY keyboard	$TTI1 in foreground
12	$LPT	Line printer	
13	$PTR	Paper tape reader (ASCII input must be even parity.)	
14	$PTP	Paper tape punch	
15	$TTR	TTY reader (ASCII input must be even parity.)	

Any of the 64 channels 0 - 63 can be referenced in a call to FOPEN with any device or file name as an argument. If the channel has an associated device, this association is temporarily suspended until FCLOS or RESET is called.

End-of-File or Error Transfer of Control

The user can regain control after an end-of-file has been encountered or an I/O error at the driver level (parity, record size) has been detected.

Within a READ or WRITE statement, the return statement number is given by the following syntax:

```
READ   (channel, [format,]ERR = n1) [list]
WRITE  (channel, [format,]ERR = n1) [list]

READ   (channel, [format,]END = n2) [list]
WRITE  (channel, [format,]END = n2) [list]

READ   (channel, [format,]ERR = n1 , END = n2) [list]
WRITE  (channel, [format,]ERR = n1 , END = n2) [list]

READ   (channel, [format,]END = n2 , ERR = n1) [list]
WRITE  (channel, [format,]END = n2 , ERR = n1) [list]
```

where: n_1 is the return statement number for an I/O error.

n_2 is the return statement for an end-of-file.

REWIND Statement

```
REWIND channel
```

REWIND Statement (Continued)

The REWIND statement causes the file associated with channel ($0-63_{10}$) to be positioned at the initial record.

If the REWIND statement is executed in the same program as the OPEN (or FOPEN) call for that particular channel, no special handling is required for the file name associated with the channel. However, if the REWIND is to be executed in some other program (for example, one at a higher level than that containing the OPEN), the file name of the file to be rewound must be stored in blank or labeled COMMON.

ENDFILE Statement

ENDFILE channel

The ENDFILE statement causes the file associated with channel ($0-63_{10}$) to be closed. If an end of file is encountered during execution of a READ statement, execution of the program is terminated unless the end of file was prepared for in the READ statement.

Random Access Files (FSEEK)

Using the Real Time Disk Operating System, random access files are keyed by record number. By default, a random file is initially positioned to the beginning of record 0. As records are read or written, the file is positioned to the beginning of the next unread or unwritten record.

The user, though, can position the random file to a given record for reading or writing by giving a call to FSEEK preceding READ or WRITE. The call to FSEEK has the format:

CALL FSEEK (channel, recordnumber)

where: channel is the channel number of the random file.

recordnumber is the number of the next record to be read or written.

An example of a call to FSEEK is:

```
CALL   FSEEK   (JCHAN, INUM)
```

Rereading and Rewriting Records (CHSAV, CHRST)

Two library routines are provided that enable the user to reread or rewrite records of a disk file. The mechanism employed is to save the status of a FORTRAN channel, issue any number of reads or writes, and then restore the original status of the channel. The records processed between the save and restore can now be read or written again. The following call to CHSAV is used to save the status of a channel:

CALL CHSAV (channel, start-word)

where: channel is an integer constant or variable specifying the number of the channel to be used within the range 0 to 63 (decimal).

start-word is an element of an integer array specifying the start of a three-word block. The three-word block is used to save the channel status for restoration.

The call to CHRST is used to restore channel status, it's format is:

Rereading and Rewriting Records (CHSAV, CHRST) (Continued)

CALL CHRST (channel, start-word)

where: channel is an integer variable or constant with a value between 0 and 63 (decimal) specifying the number of the channel to be used.

start-word is the first element of the three-word block in which the previously saved channel status is stored.

Note, for example, that this provides the user with the ability to read a record that contains formatting information and use this information to reread the same record using a different format.

The status on more than one channel may, of course, be saved, and the status of every read on a given channel may be saved using an appropriate two-dimensional integer array. This gives the user a powerful means of returning to process any record within a given disk file. An array declared as:

I(3, 100)

can be used to save up to 100 blocks of channel status information.

Both routines will cause a non-fatal error message if the channel specified is not open, and CHRST will cause a non-fatal error message if an attempt is made to restore channel information that has not been saved.

CHAPTER 7

SPECIFICATION STATEMENTS

DEFINITION

Specification statements are non-executable statements that provide the FORTRAN IV compiler with information about storage allocation and data types of simple variables and arrays to be used in the program.

DIMENSION STATEMENTS

Format:

```
DIMENSION a1 (i1), a2(i2),. . . , an,(in)
```

where: each a is the name of an array.

each i represents the subscript bounds of the array.

DIMENSION statements give the subscript bounds of arrays for allocation of storage to the arrays. A given array can only be dimensioned once. It can be dimensioned in a DIMENSION, COMMON, or data-type statement. Dummy array argument names may appear in DIMENSION statements (adjustable dimensions).

The general form of a subscript bound is:

```
sb1, sb2,. . . , sbn
```

where: each sb is an integer constant, a dummy integer variable, or a (possibly mixed) pair of these separated by a colon (:).

When a subscript bound consists of a pair of values or variables separated by a colon, the first value or variable gives the lower bound of the dimension of the array and the second value or variable gives the upper bound of the dimension of the array.

When a subscript bound is a single integer, a lower subscript bound of 1 is implied. For example:

```
DIMENSION GEORGE (3, 5, 2, 2)
```

is identical to:

DIMENSION STATEMENTS (Continued)

```
DIMENSION GEORGE (1:3, 1:5, 1:2, 1:2)
```

If the same array structure were desired with the subscripts starting at zero, the following statement would accomplish this:

```
DIMENSION GEORGE (0:2, 0:4, 0:1, 0:1)
```

Subscript bounds may give adjustable dimensions when the dimensions and the array name are contained within a subprogram and are dummy arguments to that subprogram. For example:

```
SUBROUTINE R(A, I, J, K)
DIMENSION A(I, J, K)
```

Array dimensions are not passed to subroutines, the dimensions declared within the subroutine determine the array size and structure usable within the subroutine.

Two methods are available to support variable size arrays within subroutines. The first method is called adjustable dimensioning and merely involves using variables to specify array dimensions. For example, the array dimensions could be passed as arguments to the subroutine as follows:

```
SUBROUTINE  ABC (I, J, K, A)
DIMENSION A (I, J, K)
        .
        .
        .
END
```

The second method available is to dimension the array to be essentially boundless. This is performed by specifying the array size to be one. Caution must be exercised with this method. For example, if the following subroutine is executed, an endless loop will result:

```
SUBROUTINE  PRINT  (A)
DIMENSION  A(1)
WRITE  (10)  A
RETURN
END
```

Since the array A has no bounds, subroutine PRINT will start printing the contents of core starting where array A is allocated. This subroutine should have used an implied DO loop to write the contents of array A, such as the following:

```
WRITE  (10)  ( A ( I ),  I = 1,  10)
```

DATA-TYPE STATEMENTS

Format:

```
INTEGER v1, v2,..., vn

REAL v1, v2,..., vn

DOUBLE PRECISION v1, v2,..., vn

COMPLEX v1, v2,..., vn

DOUBLE PRECISION COMPLEX v1, v2,..., vn

LOGICAL v1, v2,..., vn
```

where: each v is a variable name, an array name, a dimensioned array name, a function name, or a statement function argument name.

A data-type statement is used to specify the type of data that can be assigned to a variable. Variables used for storage of double precision, complex, double precision complex, and logical values must be specified in the appropriate data type statement. The data type of a variable may not be changed within a program unit. INTEGER and REAL type statements may be used to override implicit data typing.

Arrays may be dimensioned in data-type statements and dummy arguments may appear in data-type statements:

```
INTEGER X1, X2
REAL MEAN, MEDIAN
DOUBLE PRECISION DBL, LONG(10)
COMPLEX IMAG
LOGICAL QUES, WHICH (0:9, 0:9)
```

COMPILER DOUBLE PRECISION STATEMENT

As the initial statement of a program, the statement

```
COMPILER DOUBLE PRECISION
```

forces all REAL variables and constants to DOUBLE PRECISION and all COMPLEX to DOUBLE PRECISION COMPLEX. The COMPILER DOUBLE PRECISION statement overrides any succeeding REAL or COMPLEX statements and forces all floating-point constants to four word precision. Single precision library functions having double precision counterparts will be recognized, and calls generated to the appropriate double precision functions. (Library function precision is not overridden for functions passed as arguments nor at any time in the 8K compiler.)

The programmer can reduce his object program size by using all single or all double precision variables and constants, since the single and double precision arithmetic packages are separate. Each requires about 600 words of storage. Use of the COMPILER DOUBLE PRECISION statement thus insures that only the double precision arithmetic package is loaded.

COMMON STATEMENT

Format:

```
COMMON / block₁ / list₁ . . . / blockₙ / listₙ
```

where: each list is a list of names of variables and arrays

each block is the name of a block of common storage that is to contain the list following

A common block is a storage area shared by program units of a FORTRAN program. Storage is allocated to variables and arrays in a common block in the order in which the variables appear in COMMON statements.

There are two types of common storage. If a block name precedes a list of variables, all listed variables following that name are placed in a common storage area having the block name as a label; this is called labeled common. If no block name precedes the list, all variables of the list are placed in an unlabeled common area; this is called blank or unlabeled common. In a COMMON statement, blank common can be indicated by an empty field between two slashes (/ /). If the blank common list appears as the first list in the COMMON statement, the slashes are not needed.

The size of blank common in the various program units does not have to match; blocks of labeled common must match in size in the different program units. The size of a common block can be increased by EQUIVALENCE statements as well as COMMON statements.

A given common block may appear more than once in a COMMON statement or given program unit. Variables continue to be assigned in that order to the given common block. Arrays may be dimensioned in a COMMON statement. Dummy arguments may not appear in a COMMON statement.

Labeled COMMON takes space at load time, whereas unlabeled COMMON (and stack variables and arrays) are allocated at execution time and can thus use space previously occupied by the relocatable loader. To reduce object program space requirements, keep labeled COMMON to a minimum.

```
COMMON A, B, C, D (3,4), E
COMMON / LB / U, V (2,3), VAR
```

The two COMMON statements above are the same as the following COMMON statement:

```
COMMON / LB / U, V(2,3), VAR // AV,B,C,D(3,4), E
```

COMMON STATEMENT (Continued)

```
COMMON / BLK / A, B, C          ←program unit 1
COMMON / BLK / E, F, G          ←program unit 2
```

storage in BLK

A	B	C
E	F	G

```
COMMON A, B, C (10, 10)          ←program unit 1
DOUBLE PRECISION A, B
      .
      .
      .
COMMON P(4), D (100)             ←program unit 2
```

In the above example, the programmer wants to reference array C in program unit 1 by array D in program unit 2. To do so, he must leave four dummy locations in common (P(1) to P(4)) representing the two double precision variables A and B:

storage of blank common

A P(1)	A P(2)	B P(3)	B P(4)	C(1,1) D(1)	C(2,1) D(2)	...	C(10,10) D(100)

EQUIVALENCE STATEMENT

Format:

EQUIVALENCE ($list_1$), ($list_2$), ..., ($list_n$)

where: each list is a list of names of variables, arrays , and array elements having constant subscripts

An array name with no subscript is assumed to be the first element of the array.

All variables named within a given list of an EQUIVALENCE statement share the same storage area.

Dummy argument names of arrays cannot appear in EQUIVALENCE statements.

Since Data General's FORTRAN IV places non-COMMON variables on a stack separate from all other variables, no EQUIVALENCE is allowed to non-COMMON variables.

Equivalencing storage should not be used to equate entities mathematically. For example, if a REAL variable is equivalenced with a DOUBLE PRECISION variable, the REAL variable will share storage with only the first storage unit of the two-unit DOUBLE PRECISION variable.

EQUIVALENCE STATEMENT (Continued)

Array elements in EQUIVALENCE statements may be referenced by complete subscripts or a single subscript equal to the element's positional value.

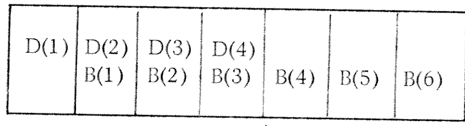
When an element of one array is equivalenced with an element of another array, that determines storage correspondence for all elements of the arrays.

Only one variable, array, or array element from a given EQUIVALENCE list can appear in a COMMON statement within the program unit.

When an array element appears in an EQUIVALENCE list with an array element that is in a common area, the equivalencing may lengthen the common area. Common may only be extended beyond the last assignment of storage made in a COMMON statement; no core storage is left empty to provide for EQUIVALENCE extensions in the other direction.

```
DIMENSION B(6), D(4)
COMMON D
EQUIVALENCE (B(1), D(2))
```

Storage in Blank Common Area



Extended Common

EXTERNAL STATEMENT

Format:

EXTERNAL $\underline{s}_1, \underline{s}_2, \ldots, \underline{s}_n$

where: each $\underline{s}$ is the name of a function subprogram or subroutine subprogram.

The EXTERNAL statement specifies subprograms as external to the program unit in which the specification is made.

The EXTERNAL specification must be given to names of functions, subroutines, and tasks that appear in the program unit as arguments to be passed to another subprogram. The EXTERNAL specification causes the argument to be recognized as a subprogram, rather than an array or variable. An address for the subprogram argument can then be passed to the called subprogram.

The data type of an EXTERNAL function subprogram may appear in a data type statement in the calling program.

EXTERNAL STATEMENT (Continued)

REAL ROOT EXTERNAL ROOT CALL MULT (A, B, ROOT)	Subroutine MULT is called with REAL function ROOT as the last argument.
SUBROUTINE MULT (Q, R, S) . . . Q = S(Q, R)	This generates a call to the function passed via dummy argument S.

COMPILER NOSTACK

Format:

```
COMPILER NOSTACK
```

The COMPILER NOSTACK statement may optionally appear as the first statement of a program unit or as the second statement if the COMPILER DOUBLE PRECISION statement is given.

The statement may be used with all versions of DGC FORTRAN IV except the 8K configuration. When given, the statement causes all non-COMMON variables and arrays to be placed in a fixed location in memory rather than on the run-time stack. It provides the following:

1) DATA initialization of non-COMMON variables.
2) All free variables are initialized to zero at load time.
3) Variables within a subprogram are available upon re-entry to the subprogram for the second and subsequent times.

Attributes 2 and 3, although not ANSI FORTRAN requirements, exist at many installations and are expected by many existent FORTRAN programs. If a working program compiles successfully using the DGC FORTRAN compiler without the NOSTACK option but does not run correctly, use the option to determine if the programmer was expecting either memory to be zeroed or variables to remain unchanged. If so, the program can be recoded to generate the most efficient code by placing just the necessary variables in a labeled COMMON and recompiling without the NOSTACK option.

CHAPTER 8

DATA INITIALIZATION

DATA STATEMENT

Format:

DATA $\underline{vlist_1}/\underline{clist_1}/\underline{vlist_2}/\underline{clist_2}/\ldots \underline{vlist_n}/\underline{clist_n}/$

where: each vlist is a list of names of variables, arrays, and array elements with constant subscripts.

each clist is a list of optionally signed constants.

A DATA statement defines initial values for variables and array elements. Variable and constant lists are paired in the statement. Constants are assigned to variables according to their positions in the paired list.

In general, arithmetic and logical variables are initialized with constants that have the same data type. COMPLEX variables are initialized with two single-precision real numbers; DOUBLE PRECISION COMPLEX variables are initialized with two double-precision real numbers.

Any variable, except COMPLEX and DOUBLE PRECISION COMPLEX, may be initialized with string data. Each character of a string constant will occupy one byte (two characters per 16-bit word). For example, an 8-character string constant will fill 4 INTEGER or LOGICAL variables, 2 REAL variables, or 1 DOUBLE PRECISION variable. A string constant will initialize any number of consecutive words, depending only upon the length of the string. No correspondence is required between data type and string length.

A string constant in a DATA statement is different from one in a FORMAT statement or elsewhere in the program in that a word of binary zeroes is not generated at the end of the constant when the character count is even. Elsewhere in a program, the word of zeroes is generated to indicate the end of a string. (See Chapter 6, Sw Field Desciptor).

The variable list may contain the names of variables, arrays, and array elements that are in a labeled common area.* Stack variables and variables in blank common may not appear in the list. Dummy arguments may not appear in the variable list.

If the name of an array appears in the variable list, the name is assumed to stand for the first element of the array unless it is the last name in the list. In the latter case, all remaining constants will be assigned to the sequential elements of the array.

*According to ANSI FORTRAN standard X3.9-1966, variables stored in labeled common may have the initial values assigned only if the DATA statement appears in a BLOCK DATA subprogram. This is not necessary in DGC FORTRAN IV.

DATA STATEMENT (Continued)

Within the constant list, a group of constants may be specified with a repeat count and multiplication symbol. The repeat count specifies the number of times the constant is to be assigned to variables of the variable list. For example:

```
DATA A,B,C,AR(1,1),AR(2,2),AR(3,3)/6*1.0/
```

causes the datum 1.0 to be assigned to the six variables of the variable list. A repeat count cannot be used with a string constant.

If the constant list is longer than the variable list, the constants will be placed in succeeding storage locations as long as their data type agrees with the type of the last variable in the list.

```
DATA I,A/1,7.0,382.0,5*3.0,0,0/
```

A is intialized to 7.0, and the next seven storage locations are initialized to the seven real constants following 7.0 in the constant list.

The contents of the DATA statement may consist of one or more paired lists of variables and constants.

```
DATA X,Y,I,L,S,P/2*1.1,0,.TRUE.,5HPRICE/

    is equivalent to

DATA X,Y/2*1.1/,I/0/,L/.TRUE./,S,P/5HPRICE/
```

Note the commas preceding I, L, and S. These are allowed for compatibility with other compilers but are not required for DGC FORTRAN IV.

BLOCK DATA SUBPROGRAM

Variables in labeled common may be initialized to values in a BLOCK DATA subprogram. The BLOCK DATA subprogram begins with the statement BLOCK DATA and terminates with an END line. It contains only DIMENSION, DATA, COMMON, data-type, and EQUIVALENCE statements.

All variables in a given labeled common block must be listed in the COMMON statement (or statements) in the BLOCK DATA program even if not all the variables are initialized to values in a DATA statement.

```
BLOCK DATA
COMMON/ELN/C,A,B/RMC/Z,Y
DIMENSION B(4), Z(3)
DOUBLE PRECISION Z
COMPLEX C
DATA B(1),B(2)/2*1.1/C/2.4,3.769/Z(1)/7.649D5/
END
```

CHAPTER 9

FUNCTIONS AND SUBROUTINES

FUNCTIONS

Functions have the following characteristics:

1. They are referenced by the appearance of the name of the function in an expression. The name is followed by any actual arguments to the function.

2. They return a single value for the function to the point of reference.

3. They have a data type.

DGC FORTRAN functions are:

1. Statement functions, which are single statements written and compiled as part of a program unit (internal).

2. Function subprograms, which are written and compiled as separate program units (external).

3. FORTRAN library functions, which are supplied with the compiler.

Statement Functions

Format:

$f(a_1, a_2, \ldots, a_n) = e$

where: f is the name given by the programmer to a function. Within a program unit, statement function names must be unique in their first five characters.

Each a is a dummy argument name.

e is an expression.

The expression on the righthand side of the statement function is evaluated and assigned to the function name on the lefthand side.

Statement functions follow the rules of data type assignment as given in Chapter 4. Function names can be explicitly typed using data-type statements or can be implicitly typed as REAL or INTEGER by applying the IJKLMN convention to the function name.

Where dummy argument names are identical to identifiers appearing in type declaration statements, the dummy arguments will have the type declared.

Besides the dummy arguments, the expression e can contain:

FUNCTIONS (Continued)

Statement Functions (Continued)

1. Constants of any type.
2. Variables stored in a COMMON area.
3. Function references to previously defined statement functions, to FORTRAN library functions, and to external functions.

The name of the function in a statement function is internal to the program unit and cannot appear in an EXTERNAL statement.

To use a statement function, the programmer places a reference to the function in an expression to be evaluated. The reference contains the function name and actual arguments to replace the dummy arguments. The actual arguments are passed to the statement functions, e is evaluated, and the value is returned to the reference point.

The actual arguments in a statement function reference must agree in order, number and type with the corresponding dummy arguments. Actual arguments in a reference may be any expression of the same type as the corresponding dummy argument.

The statement function is:

```
ROOT(A,B,C)=(-B+SQRT(B**2-4 *A*C) )/(2. *A)
      .
      .
      .
```

This statement function might be referenced by:

```
VAL = ROOT(D(6), 122.6, ABS(X-Y) ) + Z**3
```

In the example, D(6) replaces A, 122.6 replaces B, the absolute value of A-Y replaces C, and the expression:

```
(-122.6+SQRT(122.6**2-4. *D(6)* ABS(X-Y) ) )/(2. *D(6) )
```

is evaluated and returned to the assignment statement. Z**3 is added to the returned value, and the total is assigned to location VAL.

Function Subprograms

When a programmer needs a function that cannot be expressed as a single statement (statement function), he writes a function subprogram. A function subprogram is external (separately compiled). A function subprogram is referenced in the same manner as any function, returning a single value for the function to the referencing point.

A function subprogram is defined by the FUNCTION statement that begins the function subprogram. The FUNCTION statement has the format:

<u>type</u> FUNCTION <u>name</u> ($\underline{a}_1, \underline{a}_2, \ldots, \underline{a}_n$)

Function Subprograms (Continued)

where: type is INTEGER, REAL, COMPLEX, DOUBLE PRECISION, LOGICAL, DOUBLE PRECISION COMPLEX, or blank.

name is the name of the function subprogram.

each a is a dummy argument to be replaced by an actual argument when the function subprogram is referenced. The argument list may not be blank.

The function returns a value that is of the data type in the FUNCTION statement. If no data type is given, the function returns an INTEGER or REAL value depending upon the beginning letter of the function name (IJKLMN convention).

Each dummy argument of a function subprogram may be a variable name, an array name, or an external subprogram name (function or subroutine).

The name of the function subprogram must appear on the lefthand side of an assignment statement at least once in the function subprogram. DGC FORTRAN IV subprogram names must be unique within the first 5 characters. See Appendix B for specific names reserved for other purposes.

A value is returned for a function when a RETURN statement in the function subprogram is executed. The function subprogram must contain at least one RETURN statement

Function subprograms, like subroutines, can execute abnormal returns, as described in the section "Abnormal Returns", page 9-9.

Except for the FUNCTION statement itself, the name of the function subprogram cannot appear in any non-executable statement in the function subprogram.

Dummy argument names cannot appear in DATA, COMMON, or EQUIVALENCE statements in the function subprogram.

Through assignment of values to its arguments, the function subprogram can effectively return more than one value to the referencing program unit.

The function subprogram cannot contain statements that define other program units, e.e., it cannot contain another FUNCTION statement, a BLOCK DATA statement or a SUBROUTINE statement.

An example of a function subprogram is REAL function SWITCH:

```
    FUNCTION SWITCH (X)
    IF(X.LE.0.) GO TO 5
    IF(X.LT.1.) TO TO 10
20  SWITCH = 1.          ;FIRST ASSIGNMENT TO SWITCH
    RETURN
10  SWITCH = X
    RETURN
 5  RETURN
    END
```

Arguments of Function Subprograms*

When a function subprogram* is referenced, dummy argument names of a given structure are replaced by actual argument names of a similar structure as shown below:

1. Dummy Argument: Variable Name

   Actual Arguments: Variable Name
   Array Element Name
   Any Expression

   When the actual argument is an expression, its value is passed.

2. Dummy Argument: Array Name

   Actual Arguments: Array Name
   Array Element Name

   When an array name is passed:

   dummy length ≤ actual array length

   When an array element name is passed:

   dummy length ≤ actual array length + 1 - the actual array's subscript.

3. Dummy Argument: Name that can be used as a function call.

   Actual Argument: External Function Name

   The dummy argument cannot be defined or redefined in the function subprogram. *

4. Dummy Argument: Name that can be used as a subroutine name in a CALL statement.

   Actual Argument: External Subroutine Name

   The dummy argument cannot be defined or redefined in the function subprogram. *

As external function or subroutine name that is used as an actual argument in the referencing program unit must appear in an EXTERNAL statement in the referencing program unit.

If a function reference causes association of two dummy arguments in the function subprogram, neither dummy argument can be defined in the function subprogram.

FORTRAN Library Functions

The FORTRAN library functions are those functions supplied with the FORTRAN compiler. Library functions are referenced in the same way as other functions:

```
X = ABS (SIN(X) )                    ←function references
```

A list of the library functions is given on the following page. All angular quantities are in radians.

*The same correspondence of dummy to actual arguments holds for subroutine subprograms when the words "subroutine subprogram" are substituted for the words "function subprogram" as indicated by the asterisks.

Name	Function	Definition	Number of Arguments	Type of Argument	Type of Function	Functions Used For
ATAN ATAN2 DATAN DATAN2 DATN2*	arctan (arg) (quadrants 1 and 4) $\arctan(arg_1/arg_2)$ (all quadrants)	Arctangent	1 2 1 2 1	Real Real Double Double Double	Real Real Double Double Double	Trigonometric Operations
COS DCOS CCOS DCCOS	cos(arg)	Trigonometric Cosine	1	Real Double Complex DP Complex	Real Double Complex DP Complex	
SIN DSIN CSIN DCSIN	sin(arg)	Trigonometric Sine	1	Real Double Complex DP Complex	Real Double Complex DP Complex	
SINH	sinh(arg)	Hyperbolic Sine	1	Real	Real	
TAN DTAN	tan(arg)	Trigonometric Tangent	1	Real Double	Real Double	
TANH DTANH	tanh(arg)	Hyperbolic Tangent	1	Real Double	Real Double	
ABS IABS DABS	$\|arg\|$	Absolute Value	1	Real Integer Double	Real Integer Double	Arithmetic and Conversion Operations
AIMAG DAIMAG	y where: arg = x + yi	Obtain imaginary part of complex argument	1	Complex DP Complex	Real Double	
DINT AINT INT IDINT	Sign of arg times largest integer $\leq \|arg\|$	Truncation	1	Double Real Real Double	Double Real Integer Integer	
ALOG DLOG CLOG DCLOG	$\log_e$ (arg)	Natural Logarithm	1	Real Double Complex DP Complex	Real Double Complex DP Complex	

*DATN2 is used with the 8K FORTRAN compiler in place of DATAN, because DATAN and DATAN2 are identical in their first five characters. DATAN and DATAN2 can be used with the 12K compiler.

Name	Function	Definition	Number of Arguments	Type of Argument	Type of Function	Function Used For
$ALOG_{10}$ $DLOG_{10}$	$\log_{10}(\underline{arg})$	Common Logarithm	1	Real Double	Real Double	Arithmetic and Conversion Operations
AMAX0 AMAX1 MAX0 MAX1 DMAX1	$\max(\underline{arg}_1, \underline{arg}_2, \ldots)$	Choosing Largest Value	≥ 2	Integer Real Integer Real Double	Real Real Integer Integer Double	
AMIN0 AMIN1 MIN0 MIN1 DMIN1	$\min(\underline{arg}_1, \underline{arg}_2, \ldots)$	Choosing Smallest Value	≥ 2	Integer Real Integer Real Double	Real Real Integer Integer Double	
AMOD * MOD DMOD	$\underline{arg}_1(\text{mod } \underline{arg}_2)$	Remaindering*	2	Real Integer Double	Real Integer Double	
CABS DCABS	$x^2 + y^2$ where: $\underline{arg} = x + yi$	Modulus	1	Complex DP Complex	Real Double	
CMPLX DCMPLX	$\text{complex} = \underline{arg}_1 + i\underline{arg}_2$	Express 2 Real Arguments in Complex Form	2	Real Double	Complex DP Complex	
CONJG DCONJG	For: $\underline{arg} = x + yi$ $\text{conj} = x - yi$	Obtain Conjugate of Complex Argument	1	Complex DP Complex	Complex DP Complex	
DBLE	$\text{Double} = (\underline{arg}, 0)$	Express Single Precision Argument in Double Precision Form	1	Real	Double	
DIM IDIM	$\underline{arg}_1 - \min(\underline{arg}_1, \underline{arg}_2)$	Positive Difference	2	Real Integer	Real Integer	
EXP DEXP CEXP DCEXP	$e^{\underline{arg}}$	Exponential	1	Real Double Complex DP Complex	Real Double Complex DP Complex	

*The function $\left\{\begin{matrix}\text{AMOD}\\ \text{MOD}\\ \text{DMOD}\end{matrix}\right\}(\underline{arg}_1, \underline{arg}_2)$ is defined as: $\underline{arg}_1 - [\,\underline{arg}_1 / \underline{arg}_2\,]\,\underline{arg}_2$

where: $[\,\underline{arg}_1 / \underline{arg}_2\,]$ is the truncated value of that quotient.

Name	Function	Definition	Number of Arguments	Type of Argument	Type of Function	Function Used for
FLOAT DFLOAT	Float	Convert from Integer to Real	1	Integer Integer	Real Double	Arithmetic and Conversion Operations
IFIX	Fix	Convert from Real to Integer by Truncation	1	Real	Integer	
REAL	x where: arg = x + yi	Obtain Real Part of Complex Argument	1	Complex DP Complex	Real Double	
SIGN ISIGN DSIGN	sign of $arg_2 * arg_1$	Transfer of Sign	2	Real Integer Double	Real Integer Double	
SNGL	arg	Obtain Most Significant Part of Double Precision Argument	1	Double	Real	
SQRT DSQRT CSQRT DCSQRT	$(arg)^{1/2}$	Square Root	1	Real Double Complex DP Complex	Real Double Complex DP Complex	
IAND *	arg_1 arg_2	16-bit ANDing	2	Integer	Integer	Bit/Word Manipulation and Testing
IOR *	arg_1 arg_2	16-bit ORing	2	Integer	Integer	
NOT	arg	Logical Complement	1	Integer	Integer	
IEOR	arg_1 arg_2	16-bit Exclusive OR	2	Integer	Integer	
ISHFT	arg_1, arg_2	Shift arg_1 by the number of bits given in arg_2, where: $arg_2 < 0$ right shift $arg_2 = 0$ no shift $arg_2 > 0$ left shift	2	Integer	Integer	
ITEST BTEST	arg_1, arg_2	Test a Bit within the Word Given by arg_1. The Bit Tested Is 15-arg_2. The Result Returned Is: 0 if tested bit = 0 -1 if tested bit = 1	2	Integer	Logical	

SUBROUTINES

Subroutines, also called subroutine subprograms, are external (separately compiled). They return values to the calling program unit only through actual-dummy argument correspondence, and they return to the calling program unit at the statement following the subroutine call unless they execute a RETURN via a dummy argument.

A subroutine is defined by the SUBROUTINE statement that begins the subroutine and has the format:

SUBROUTINE name (a_1, a_2, . . . , a_n)

where: name is the name of the subroutine.

each a is a dummy argument to be replaced by an actual argument when the subroutine is referenced. The argument list may be blank.

Each dummy argument of a subroutine may be a variable name, array name, or an external subprogram name (function or subroutine). Dummy argument names cannot appear in COMMON, EQUIVALENCE, or DATA statements in the subroutine subprogram.

The correspondence between dummy argument names of subroutines and actual arguments passed to the subroutine when it is referenced is the same as that given for function subprograms (page 9-4).

Within the subroutine, name may only appear in the SUBROUTINE statement immediately following the word SUBROUTINE. Subprogram names must be uniquely distinguishable by their first five characters. See Appendix B for names reserved for other purposes.

Through assignment of values to its arguments, the subroutine can effectively return values to the referencing program unit.

The subroutine must contain at least one RETURN statement. Return is made to the referencing program unit when a RETURN statement is executed.

The subroutine cannot contain statements that define other program units, i.e., it cannot contain another SUBROUTINE statement, a BLOCK DATA statement, or a FUNCTION statement.

An example of a subroutine subprogram is:

```
      SUBROUTINE REV(ARRAY, I1, I2)
      DIMENSION ARRAY (100)
      I12 = I1 + I2
      MID = I12/2
      DO 50 I = I1, MID
      J = I12 -I
C     USE TEMPORARY TO REVERSE
C     ELEMENTS OF ARRAY
      TEMP = A(I)
      A(I) = A(J)
      A(J) = TEMP
 50   CONTINUE
      RETURN
      END
```

SUBROUTINES (Continued)

A subroutine subprogram is referenced by a CALL statement. (See Chapter 5.) If the SUBROUTINE statement contains dummy arguments, the CALL statement must contain actual arguments that replace the dummy arguments.

When the subroutine has been executed, normal return is made to the statement in the calling program unit immediately following the CALL statement. For example, subroutine subprogram REV might be called from another program unit as shown.

```
DIMENSION A(100)
       .
       .
       .
CALL REV (A, K1, K2)
```

ABNORMAL RETURNS

Normally, return from a subroutine is to the statement immediately following the CALL statement, and return from a function is to the point of function reference.

It is possible to return to some other statement in the calling program. To do so, the called function or subroutine must contain a dummy integer argument that is used as a variable in a RETURN statement.

```
SUBROUTINE SUB (DUM, I, R1, Q, K)
INTEGER Q
       .
       .
       .
RETURN Q
```

When the subroutine SUB is referenced, the calling program passes a statement label to replace the dummy integer argument. The statement label must be preceded by a dollar sign ($).

```
          .
          .
          .
      CALL SUB (A, K1, K2, $25, K3)
          .
          .
          .
 25   - - - - - - -
          .
          .
          .
```

If an abnormal RETURN statement in SUB is executed referencing the fourth dummy argument, return will be made to the statement labeled 25 in the calling program.

ABNORMAL RETURNS (Continued)

Abnormal returns from functions are made in the same way. Rather than returning to the point of the reference, the return will be made to a statement, whose label is passed as an argument replacing the integer variable in the RETURN statement of the function being referenced.

DGC FORTRAN IV LIBRARY

Certain functions and subroutines supplied with the FORTRAN IV library are described in brief in this manual. Functions are described on pages 9-5 to 9-7; Chapter 6 contains non-real time I/O calls; and PART II describes calls that provide the real time interface to RDOS. In addition the next section of this chapter describes three bit manipulation routines.

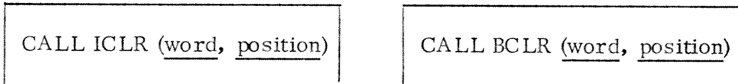
However, the functions and subroutines described in this manual are limited to those that can most commonly be used by programmers as well as by the system. For a full description of the FORTRAN IV library, see the FORTRAN IV Run Time Library User's Manual, 093-000068.

BIT/WORD MANIPULATION

Calls to run time routines permit bits of an integer variable to be accessed to change the setting or for testing.

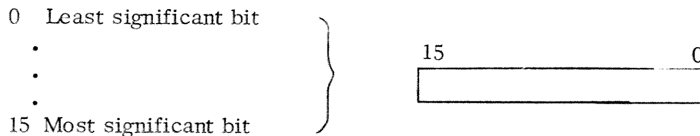
Clear a Bit (ICLR, BCLR)

A single bit in a word can be set to zero by executing a call to ICLR or to BCLR. The format of the call is:



where: word is an integer variable, one of whose bits is to be cleared.

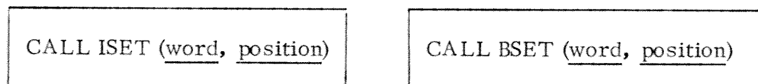
position is an integer constant or variable whose value specifies the bit position in the word to be set to zero:



Example:

Set a Bit (ISET, BSET)

A single bit in a word can be set to one by executing a call to ISET or to BSET. The format of the call is

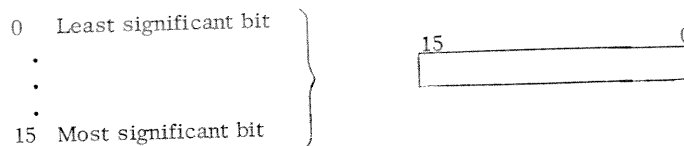


BIT/WORD MANIPULATION (Continued)

Set a Bit (ISET, BSET) (Continued)

where: word is an integer variable, one of whose bits is to be set to one.

position is an integer variable or constant whose value specifies the bit position in the word to be set to one:

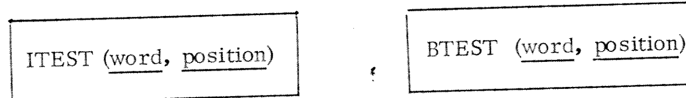


Example:

```
CALL ISET (MON, 0)          BIT  0 of MON will be set
```

Test a Bit (ITEST, BTEST)

A single bit in a word can be tested, using the integer function ITEST or BTEST. ITEST and BTEST are referenced by the following formats:



where: word is a logical variable, one of whose bits is to be tested.

position is an integer constant or variable whose value specifies the bit position to be tested. The bit tested is 15_8 - position.

The logical value returned by ITEST (BTEST), is a -1 (true) if the tested bit is one and zero if the tested bit is zero (false).

Example:

```
IF (ITEST (I, J ) )    GO TO 10
```

BIT/WORD MANIPULATION (Continued)

Shift a Word (ISHFT)

A word can be shifted a number of bits left or right using the integer function ISHFT. ISHFT is referenced by the following format:

```
ISHFT (word, bits )
```

where: word is an integer variable that is to be shifted.

bits is an integer constant or variable whose value specifies the number of bit postions to be shifted and:

a negative value represents a right shift.

a positive value represents a left shift.

Example:

ISHFT (J, -5)	Shift contents of J 5 bits to the right.

INDEX - PART I

A (assembly source code line) 1-2
A (in format conversion)) 6-10, 6-12
abnormal return 5-4
ABS library function 9-5
ACCEPT statement
  description of 2-5
  strings in 2-5
  input/output of 6-20
addition 3-1, 3-2
adjustable dimensions 2-6, 7-1
AIMAG library function 9-5
AINT library function 9-5
ALOG library function 9-5
ALOG10 library function 9-6
AMAX0 library function 9-6
AMAX1 library function 9-6
AMIN0 library function 9-6
AMIN1 library function 9-6
AMOD library function 9-6
angle brackets 2-6
.AND. 3-4
apostrophes 2-5
argument
  correspondence to dummy 9-4
  external subprogram 7-5
  of function Chapter 9
  of subroutine Chapter 9
arithmetic data
  assignment 4-1
  constant 2-1
  conversion Chapters 2, 3, 4
  expression 3-1
  IF statement 5-3
  I/O conversion Chapter 6
  operators 3-1
  mixed data types 3-2
  representation Chapter 2, Append. E
  storage Chapter 2, Appendix E
  variable 2-1
arithmetic statement function 9-1
arrays
  assigning values to 2-8
  adjustable dimensions of 2-6, 7-1
arrays (Continued)
  dimensions 2-6, 7-1
  boundless 7-2
  element of 2-6
  equivalencing Chapter 7
  format specification in Chapter 6
  input/output of 6-1, 6-2
  not declared in COMMON 1-1
  variable size of 7-2
ASCII characters 1-1
assembly source code 1-2
ASSIGN statement 5-2
assigned GOTO statement 5-2
assignment
  statements Chapter 4
  definition of 4-1
  illegal versions of 4-1
  rules for using 4-1, 4-2
  of arithmetic data 4-1
  of values to arrays 2-8
  of logical data 4-1
asterisk 1-3, 3-1, 3-2, 3-4, 6-8
ATAN library function 9-5
ATAN2 library function 9-5

basic field descriptor Chapter 6
BCLR routine 9-10, 9-11
binary I/O 6-1, 6-19
bit manipulation
  clear bit and set bit 9-10
  logical operations 3-5, 9-7
  test bit 9-7, 9-11
blank
  common 7-4
  descriptor Chapter 6
  in input conversion Chapter 6
  as space
    in GOTO 5-1
    in string constant 2-5
    in variable 2-1
    as delimiter 1-3
block data
  statement 8-2
  subprogram 1-1, 8-2

boundless arrays 7-2
BSET 9-10, 9-11
BTEST routine 9-7, 9-11

C (comment line) 1-1
CABS library function 9-6
CALL
  strings in 2-5
  statement 5-3
  to function Chapter 9
  to run time routines (see PART II)
  to subroutine Chapters 5, 9
carriage
  control tabulation 1-1, 6-10, 6-11
  use of slashes in 6-7, 6-14
  vertical spacing codes 6-15
  Z suppressor of 6-10, 6-16
  return 1-2, 2-4, 6-10
CCOS library function 9-5
CEXP library function 9-6
CHANTASK statement 1-2
character positions 1-1, 1-2
character set 1-3, 1-1
character string
  constant 2-4
  data initialization to 8-1
  I/O conversion Chapter 6
  storage 2-6, App. E
CHSAV routine 6-24
CHRST routine 6-24
CLOG library function 9-5
CMPLX library function 9-6
Comma 6-7
comment line
  following semicolon 1-2
  indicated by C 1-1
common
  blank 7-4
  effect on reentrant program 1-1
  effect when equivalencing Chap. 7
  extended 7-6
  labeled 7-4
  statement 1-2, 2-6, 7-3
  storage 1-1

COMPILER DOUBLE PRECISION 1-2, 7-3
COMPILER NOSTACK 1-2, 7-7
COMPLEX statement 1-2, 2-3
complex constant 2-2
complex
  data 2-4
  in unformatted I/O 6-4
computed GOTO statement 5-1
CONJG function 9-6
conjunction (logical) 3-4
constant
  complex 2-2
  definition 2-1
  double precision 2-3
  double precision complex 2-4
  Hollerity or string 1-3, 2-5
  integer 2-3, 2-1
  logical 2-2, 2-5
  real 2-3
  repetition 6-6, 6-13
continuation lines 1-2
CONTINUE statement 5-4
control statements
  arithmetic IF 5-3
  assigned GOTO 5-2
  ASSIGN statement 5-2
  CALL statement 5-3
  computed GOTO 5-1
  CONTINUE statement 5-4
  definition 5-1
  DO statement 5-5
  logical IF 5-3
  PAUSE statement 5-5
  RETURN statement 5-4
  STOP statement 5-5
  unconditional GOTO 5-1
conversion of data Chapters 3, 4
conversion, I/O Chapter 6
COS library function 9-5
CSIN library function 9-5
CSQRT library function 9-7

D (double precision) 2-3

D in format conversion 6-7 f
DABS library function 9-5
DAIMAG library function 9-5
data
  alphabetic 6-1 ff
  character string 2-4
  complex 2-4
  conversion Chapters 3, 4
  double precision 2-3
  double precision complex 2-4
  formatting of Chapter 6
  Hollerith 2-4
  initialization 8-1 f
  integer 2-1
  octal 2-2, 2-6
  precision of Chapters 2, 3, 4
  real 2-2
  statement
    description 1-2, 2-7, 8-1
    strings in 2-5
  - type
    statements Chapter 7
    specification of 7-3 ff
    description of 7-3 ff
    evaluating mixed 3-2
    of parameters 2-1
  typing of 2-1, 7-1
DATAN library function 9-5
DATAN 2 library function 9-5
DATN2 library function 9-5
DBLE library function 9-6
DCABS library function 9-6
DCCOS library function 9-5
DCEXP library function 9-6
DCLOG library function 9-5
DCMPLX library function 9-6
DCONJG library function 9-6
DCOS library function 9-5
DCSIN library function 9-5
DCSQRT library function 9-7
declaration statements
  COMMON 1-2
  COMPLEX 1-2
  DIMENSION 1-2
  DOUBLE PRECISION 1-2
  EQUIVALENCE 1-2
  EXTERNAL 1-2
  INTEGER 1-2
  LOGICAL 1-2
  REAL 1-2
define
  initial values for variables 8-1
  initial values for array elements 8-1
descriptor, field Chapter 6
DEXP library function 9-6
DFLOAT library function 9-7
diagnostics Appendix B
dimension
  adjustable 2-6, 7-1
  of an array 2-6, 7-1
  statement 1-2, 2-6
DIM library function 9-6
DINT library function 9-5
disjunction (logical 3-4
division 3-1, 3-2, 3-3
DLOG library function 9-5
DLOGIO library function 9-6
DMAXI library function 9-6
DMINI library function 9-6
DMOD library function 9-6
DO
  -implied list 6-1
  statement 5-5
    nesting of 5-6
    restrictions of 5-6
    extended range of 5-7
double precision
  constant 2-3
  data 2-4
  in unformatted I/O 6-4
  output conversion 6-8
  statement 1-2, 2-3

DOUBLE PRECISION COMPLEX
statement (data) 2-4
in unformatted I/O 6-4
DREAL library function 9-7
DSIGN library function 9-7
DSIN library function 9-5
DSQRT library function 9-7
DTAN library function 9-5
dummy argument Chapter 9

E format conversion 6-7
E (real data) 2-3
element of an array 2-6
end
of line 1-1
of program 1-1
of subprogram 5-2
of record indicators 6-5
END line 1-1
END FILE statement 6-24
.EQ. 3-3
equals sign 1-3
EQUIVALENCE statement 1-2,2-7,7-5
equivalencing of arrarys Chapter 7
error messages Appendix B
evaluating
expressions Chapter 3
mixed data types 3-2
exclamation point Chapter 6
executable statements 1-2
execution
of program Appendix D
resume after PAUSE 5-5
execution - time formatting Chapter 6
EXP library function 9-6
explicit data typing 7-1
expression
arithmetic 3-1
definition 3-1
evaluation Chapter 3
in non-executable statement 2=7
in executable statement 2-7
logical 3-4
mixed mode 3-2
operators for 3-1 ff
relational 3-4
extended common 7-6
extended DO range 5-7
extending core 1-1
EXTERNAL statement 1-2, 7-6

F format conversion 6-7
factor, scale 6-17
.FALSE. 2-5, 3-4
field descriptors 6-6 ff
field separators 6-6 ff
filename 6-24
fixed memory locations 7-7
fixed-point notation 2-2
FLOAT library function 9-7
floating-point notation 2-2
forcing values 7-3
format
I/O Chapter 6
strings in 2-5
strings constant 2-6
statement 1-2, 6-6
specification
purpose of 6-6
during run time 6-18
form feed 1-2, 6-15

FORTRAN
- assembler interface Appendix
- character set 1-3, 1-1
- compilation Appendix D
- execution Appendix D
- I/O Chapter 6
- library functions 9-4
- operators Chapter 3
- programs 1-1
- program units 1-1
- statements 1-2

FSEEK routine 6-24
full-word logical operations 3-3
function
- subprogram 1-1, 1-2, Chapter 9
- statement 1-2, 9-2
- names 9-1

G format conversion 6-7
.GE. 3-2, 3-3
GOTO
- unconditional 5-1
- computed 5-1
- assigned 5-2

.GT. 3-2, 3-3

H format conversion 6-10
hierarchy of operations 3-3
Hollerith string
- constants 1-3, 2-5
- in relational expressions 3-4
- in logical expressions 3-4
- in FORMAT specification 6-6

I format conversion 6-7
IABS library function 9-5
IAND library function 9-7
ICLR routine 9-10
IDIM library function 9-6
IDINT library function 9-5
IEOR library function 9-7
IF arithmetic 5-3
IF logical 5-3
IFIX library function 9-7
IJKLMN convention 2-3
initialization, of data 8-1
input/output
- types of 6-1
- statements Chapter 6
- programmed 6-1
- unformatted 6-4 ff
- formatted 6-6
- binary 6-1, 6-19

integer
- conversion Chapters 3, 4
- definition 2-1
- implied typing 2-1
- I/O conversion 6-6
- statement 1-2, 7-1
- storage 2-3
- field for (unformatted I/O) 6-4

INT library function 9-5
IOR library function 9-7
ISET routine 9-10
ISHFT library function 9-7, 9-12
ITEST library function 9-7, 9-11

K (following octal integer) 2-2

L format conversion 6-10
label
- character positions for 1-2
- leading zeroes in 1-2

labels 1-2, 5-1, Chapters 6, 9
labeled common 7-4
.LE. 3-3
leading zeroes in labels 1-2

library functions
- ABS 9-5
- AIMAG 9-5
- AINT 9-5
- ALOG 9-5
- ALOG10 9-6
- AMAX0 9-6
- AMAX1 9-6
- AMIN0 9-6
- AMIN1 9-6
- AMOD 9-6
- ATAN 9-5
- ATAN2 9-5
- BTEST 9-7
- CABS 9-6
- CCOS 9-5
- CEXP 9-6
- CLOG 9-5
- CMPLX 9-6
- CONJG 9-6
- COS 9-5
- CSIN 9-5
- CSQRT 9-7
- DABS 9-5
- DAIMAG 9-5
- DATAN 9-5
- DATAN2 9-5
- DATN2 9-5
- DBLE 9-6
- DCABS 9-6
- DCCOS 9-5
- DCEXP 9-6
- DCLOG 9-5
- DCMPLX 9-6
- DCONJG 9-6
- DCOS 9-5
- DCSIN 9-5
- DCSQRT 9-7
- DEXP 9-6
- DFLOAT 9-7
- DINT 9-5
- DLOG 9-5
- DLOG10 9-6

library functions (Continued)
- DMAX1 9-6
- DIMN1 9-6
- DMOD 9-6
- DREAL 9-7
- DSIGN 9-7
- DSIN 9-5
- DSQRT 9-7
- DTAN 9-5
- EXP 9-6
- FLOAT 9-7
- IABS 9-5
- IAND 9-7
- IDIM 9-6
- IDINT 9-5
- IEOR 9-7
- IFIX 9-7
- INT 9-5
- IOR 9-7
- ISHFT 9-7
- ISIGN 9-7
- ITEST 9-7
- MAX0 9-6
- MAX1 9-6
- MIN0 9-6
- MIN1 9-6
- MOD 9-6
- REAL 9-7
- SIGN 9-7
- SIN 9-5
- SINH 9-5
- SNGL 9-7
- SQRT 9-7
- TAN 9-5
- TANH 9-5

line
- comment 1-1, 1-2
- continuation of 1-2
- end of 1-1
- label within a 1-2
- of assembly code 1-2

list, definition 6-1

literal declaration 2-1, 6-12
logical
- assignment of 4-1
- conjunction 3-4
- data type 2-2, 2-5
- definition 2-2, 2-5
- disjunction 3-4
- expression 3-4
- field for 6-4
- input/output of 6-10
- negation 3-4
- operators 3-4
- statement 1-2, 7-1

logical IF statement 5-3
loop
- DO 5-5
- DO-implied 6-1

lower bound 7-1
.LT. 3-3

main program unit 1-1
mathematical data types
- INTEGER 2-2
- REAL 2-2
- DOUBLE PRECISION 2-2
- COMPLEX 2-2
- DOUBLE PRECISION COMPLEX 2-2

MAX0 library function 9-6
MAX1 library function 9-6
MIN0 library function 9-6
MIN1 library function 9-6
mixed data types 3-2
mixed made expression Chapters 3, 4
MOD library function 9-6
multiple record format 6-14
multiplication 3-1

name
- of subprogram 2-1, 9-3
- of variable 2-1

.NE. 3-3
negation (logical) 3-4
nested
- do loop 5-6
- do-implied list 6-1
- parenthesis in formatting 6-

.NOT. 3-4
numbers, in FORMAT 2-1
numerical conversion
- on imput 6-7
- on output 6-8

O format specifier 6-10
octal
- constant 2-2
- I/O conversion 6-10
- strings
  - in PAUSE 2-1
  - in STOP 2-1

operator
- arithmetic 3-1
- logical 3-4
- precedence 3-4
- relational 3-3

optionally compiled line 1-1
.OR. 3-4
order
- of statements 1-2
- of operator evaluation 3-3
- of assignment of values to array elements 2-5

output conversion of integers 6-8
OVERLAY statement 1-2

PARAMETER statement 1-2, 2-1
parenthesis 1-3, 3-1, 6-13, 6-15
PAUSE statement 5-5
positioning field descriptors 6-11
preassigned I/O channels 6-23

precedence
- of arithmetic oper. 3-1
- of relational oper. 3-4
- of logical oper. 3-4

precision of data Chapters 2, 3

program
- definition 1-1
- end of 1-1
- loop 5-5 ff
- reducing size of 7-3, 7-4
- stop execution of 5-5
- unconditional termination 5-5
- units 1-1
- reentrant 1-1

program units
- main 1-1
- subroutine subprogram 1-1
- function subprogram 1-1
- block data subprogram 1-1
- task subprogram 1-1
- source text of 1-1

quotation marks 2-5

radix 8 conversion 6-10

range of a DO loop 5-5, 5-7

READ statement
- free form 6-5
- description 6-1
- unformatted 6-5

READ BINARY statement 6-20

real
- data
  - field for 6-4
  - definition 2-3
  - output conversion of 6-8
- statement 1-2, 2-3

REAL library function 9-7

reentrant programs 1-1

reference
- a subroutine 5-4
- an array element 2-6 f
- common variables 7-4
- function Chapter 9

relational
- expression 3-3, 3-4
  - strings in 2-5
- operators 3-3

repetition constant 6-6, 6-13

resewed
- words 2-1
- library functions 2-1
- operator names 2-1

restore
- channel status 2-12

return
- abnormal 5-4, 9-9
- from function 5-4
- from subroutine 5-4, 9-8

RETURN statement 5-4, 9-8

REWIND statement 6-23

runtime stack 1-1

runtime format specifications 6-18

S format conversion 6-10, 6-11

scale factor 6-17

segmentation 1-1

semicolon
- indicating start of comment 1-2
- in a Holbrith constant 1-2

shift L 1-4

SIGN library function 9-7

SINH library function 9-5

SIN library function 9-5

slash in field description 6-7, 6-14

source
  text 1-1
  program, example 1-3
  code, assembly 1-2
SNGL library function 9-7
special symbols 1-3
specification statements
  definition 7-1
  DIMENSION 7-1
  COMPILER DOUBLE PRECISION 7-3
  COMMON 7-4
  EQUIVALENCE 7-5
  EXTERNAL 7-6
  COMPILER NOSTACK 7-7
SQRT library function 9-7
statement
  assignment Chapter 4
  control Chapter 4
  data initialization Chapter 8
  definition of 1-1
  format of Chapter 6
  functions Chapter 9
  I/O of Chapter 6
  label of a 1-1
  numbers 2-1
  order of 1-2
  specification Chapter 7
statement list
  ACCEPT 2-5
  ASSIGN 5-2
  BLOCK DATA 8-2
  CALL 5-3
  COMMON 1-2, 2-6, 7-3
  COMPILER DOUBLE
    PRECISION 1-2, 7-3
  COMPILER NOSTACK 1-2, 7-7
  COMPLEX 1-2, 2-3
  CONTINUE 5-4
  DATA 1-2, 2-7, 8-1
  DIMENSION 1-2, 2-6
  DO 5-5
statement list (Continued)
  DOUBLE PRECISION 1-2, 2-3
  DOUBLE PRECISION COMPLEX 2-4
  ENDFILE 6-24
  EQUIVALENCE 1-2, 2-7, 7-5
  EXTERNAL 7-6
  FORMAT 1-2, 6-5
  FUNCTION
  GOTO 5-1, 5-2
  IF 5-3
  INTEGER 1-2, 7-1
  ITEST 9-7, 9-11
  LOGICAL 7-1
  PARAMETER 2-1
  PAUSE 5-5
  READ
  READ BINARY
  REAL 2-3
  RETURN 5-4
  REWIND 6-23
  STOP 5-5
  SUBROUTINE
  TYPE 2-5
  WRITE 6-1, 6-23
  WRITE BINARY
STOP statement 5-5
storage
  allocation Chapter 7
  blank common, of 7-5
  blocks of 7-4
  common 7-4, 1-1
  sharing of 7-4
strings
  constants
    in CALL statement 2-5
    in DATA statement 8-1
    in TYPE statement 2-5
    blank space in 2-5
    description 2-5
    in source code 2-2
    with parameters 2-2

strings (Continued)
- data 2-1
- literals 6-12
- end of 2-6, 6-11
- I/O of 6-11, 6-12
- using S descriptor 6-10, 6-11

subprogram
- subroutine 1-1
- function 1-1
- block data 1-1
- task 1-1

subroutine
- subprogram 1-1, 1-2
- referencing a 5-4
- statement 1-2, 9-8

subscript
- bounds 7-1 f
- of an array
  - range of values 2-7
  - single 2-7
  - in multi dimensional 2-7

symbolic name 2-1

T format descriptor 6-10, 6-11
TAB key 1-2
tabulation
- to character position 8 1-1
- using T descriptor 6-10, 6-11

TAN library function 9-5
TANH library function 9-5
task
- subprogram unit 1-1
- statement 1-2

TASK statement 1-2
teletypewriter I/O 6-20
termination
- of program 1-1
- statement 1-1

transfer of control Chapter 5
.TRUE. 2-5, 3-4
truth values 3-4
type
- of file created 2-2

TYPE statement 2-5, 6-20

unconditional GOTO 5-1
unformatted I/O Chapter 6
unlabeled common 7-4
unsigned integers as labels 1-2
upper bound 7-1
v

variables
- blank space in 2-1
- common 7-4
- complex 2-4
- definition 2-1
- double precision 2-3
- double precision complex 2-4
- integer 2-2
- logical 2-5
- not declared as COMMON 1-1
- real 2-3
- size of array 7-2

vertical carriage control 6-15
WRBLK routine

WRBLK routine 2-11
WRITE statement 6-1, 6-23
WRITE BINARY statement 6-20

X (optionally compiled line) 1-1
X field descriptor 6-10, 6-11

Z field descriptor 6-10, 6-16

PART II

Chapter 1 - INTRODUCTORY CONCEPTS OF THE OPERATING SYSTEMS
Chapter 2 - SYSTEM AND DIRECTORY MAINTENANCE
Chapter 3 - FILE MAINTENANCE AND I/O
Chapter 4 - TASKING
Chapter 5 - SWAPPING, CHAINING, AND OVERLAYS
Chapter 6 - REAL TIME CLOCK AND CALENDAR
Chapter 7 - FOREGROUND/BACKGROUND PROGRAMMING

TABLE OF CONTENTS

CHAPTER 1 - INTRODUCTORY CONCEPTS OF THE OPERATING SYSTEMS

Discussion of Terms 1-1
Multitasking 1-1
Program Segmentation 1-2
System Directory 1-2
File Structures 1-2
Dual Programming 1-3
Mapping 1-3
User Status Table 1-3
FORTRAN IV Error Flags 1-3

CHAPTER 2 - SYSTEM, DIRECTORY, AND DEVICE CONTROL

Directories, Disks, and Disk Partitions 2-1
FORTRAN Calls Interfacing to System Directory Commands 2-2
Change the Current Directory (DIR) 2-3
Initialize a Directory (INIT) 2-3
Release a Directory (RLSE) 2-4
Get the Default Directory/Device Name (GDIR) 2-4
Create a Subdirectory (CDIR) 2-5
Create a Secondary Partition (CPART) 2-5
Get the Logical Name of the Master Device (MDIR) 2-5
Perform a Disk Bootstrap (BOOT) 2-6
Get the Name of the Current System (GSYS) 2-6
Device Control 2-7
Disable Console Interrupts (ODIS) 2-7
Enable Console Interrupts (OEBL) 2-7
Enable Spooling (SPEBL) 2-7
Disable Spooling (SPDIS) 2-8
Stop a Spool Operation (SPKIL) 2-8
User Interrupt Servicing 2-8
Identifying a User Interrupt Device (FINTD) 2-9
Remove a Service Interrupt Device (FINRV) 2-9

CHAPTER 3 - FILE MAINTENANCE AND I/O CONTROL

Files, File Names 3-1
Referencing a File 3-2
Referencing a File on Magnetic Tape or Cassette Units 3-2
Links, Link Entries 3-3
Linking Attributes 3-3
File Maintenance 3-4
Assign a New Name to the Multiple File Device (EQUIV) 3-4
Create an RDOS Disk File (CFILW) 3-4
Delete an RDOS Disk File (DFILW) 3-5
Delete a File (DELETE) 3-5
Rename a File (RENAME) 3-5
Create a Link Entry (DLINK) 3-6
Delete Link Entries in the Current Directory (DULNK) 3-6
Get File Directory Information for Given Channel (CHSTS) 3-6
Get Current File Directory Information (STAT) 3-7
Update Current File Size (UPDATE) 3-8

CHAPTER 3 - FILE MAINTENANCE AND I/O CONTROL (Continued)

File Attribute Maintenance 3-8
Examine the Attributes of a File (GTATR) 3-8
Change, Add, or Delete File Attributes (FSTAT) 3-9
Change or Add Link File Access Attributes (CHLAT) 3-9
File Input/Output 3-10
Get the Name of the Current Input/Output Console (GCIN, GCOUT) 3-10
Opening Files 3-10
Open a File (OPEN) 3-10
Open a File (FOPEN) 3-11
Open a File for Appending (APPEND) 3-12
Closing Files 3-13
Close a File (CLOSE) 3-13
Close a File (FCLOS) 3-13
Close All Open Files (RESET) 3-13
Reading and Writing Blocks and Records 3-14
Read a Series of Blocks (RDBLK) 3-14
Read a Series of Records (READR, RDRW) 3-14
Write a Series of Records (WRITR, WRTR) 3-15
Write a Series of Blocks (WRBLK) 3-15
Free Format Cassette and Magnetic Tape I/O 3-16
Open a Cassette or Magnetic Tape Unit for Free Format I/O (MTOPD) 3-16
Free Format Tape I/O (MTDIO) 3-17

CHAPTER 4 - TASKING

Multitasking Concepts 4-1
Task States 4-1
Task Control Blocks 4-1
Task Priorities 4-2
Task Scheduler 4-2
Task Execution Control 4-3
Number of Tasks 4-3
Task Activation (FTASK, ITASK, ASSOC) 4-4
Task Activation Based on Time of Day (FQTASK) 4-6
Start a Task after a Time Delay (START) 4-7
Execute a Task at a Specified Time (TRNON) 4-8
Task Suspension (SUSP, ASUSP, HOLD, WAIT, FDELY) 4-8
Ready a Task (ARDY, RELSE) 4-9
Task Priority Modification (PRI, CHNGE) 4-10
Task Termination (KILL, AKILL, ABORT, EXIT) 4-11
Obtaining a Task Status (STTSK) 4-11
Intertask Communication (XMT, REC, XMTW) 4-12
Task Operator Communication Module 4-14
Initializing the Task Operator Communication Module (IOPC) 4-14
Building a Program Table (IOPROG) 4-15
Sample Tasking Program 4-16

CHAPTER 5 - SWAPPING, CHAINING, AND OVERLAYS

Program Swapping and Chaining 5-1
Program Swapping (SWAP, FSWAP) 5-1
Restoring a Swapped Program (BACK, FBACK, EBACK) 5-3
Program Chaining (CHAIN, FCHAN) 5-4
Returning to Level Zero (STOP, EXIT) 5-4

CHAPTER 5 - SWAPPING, CHAINING, AND OVERLAYS (Continued)

Overlays 5-4
Numbering of Overlays within Overlay File 5-7
Overlays in a Single and Multiple Task Environment 5-8
Naming an Overlay (OVERLAY) 5-8
Opening an Overlay File (OVOPN) 5-9
Closing an Overlay File (CLOSE) 5-9
Loading Overlays in a Single Task Environment (OVLOD) 5-9
Loading an Overlay in a Multiple Task Environment (FOVLD) 5-11
Releasing an Overlay Area (FOVRL) 5-12
Releasing an Overlay (OVKIL, OVKIX, OVEXT, OVEXX) 5-13
The Overlay Loader 5-14

CHAPTER 6 - REAL TIME CLOCK AND CALENDAR

Introduction 6-1
Setting the Real Time Clock (FSTIM) 6-1
Setting the Real Time Clock (STIME) 6-1
Getting the Time of Day (TIME) 6-2
Getting the Time (FGTIM) 6-2
Getting the Date (DATE) 6-2
Setting the Date (SDATE) 6-3
User/System Clock Commands 6-3
Define a User Clock (DUCLK) 6-3
Remove a User Clock (RUCLK) 6-3
Examine the System Real Time Clock Frequency (GFREQ) 6-4

CHAPTER 7 - FOREGROUND/BACKGROUND PROGRAMMING

Introductory Concepts 7-1
Foreground/Background Considerations in an Unmapped Environment 7-1
Foreground/Background Considerations in a Mapped Environment 7-2
Foreground/Background Calls 7-2
Load a Foreground Save File (EXFG) 7-2
Load and Execute a Background Program (EXBG) 7-2
See If a Foreground Program is Running (FGND) 7-3
Define a Communications Area (ICMN) 7-3
Write a Message (WRCMN) 7-3
Read a Message (RDCMN) 7-4
Write an Operator Message (WROPR) 7-4
Read an Operator Message (RDOPR) 7-4

CHAPTER 1

INTRODUCTORY CONCEPTS OF THE OPERATING SYSTEMS

FORTRAN IV can be used in conjunction with DGC's three operating systems, namely, the Real Time Disk Operating System, the Real Time Operating System, and the Stand-alone Operating System. FORTRAN IV can also be executed in stand-alone mode (without the use of an operating system).

The Real Time Disk Operating System (RDOS) is a disk-oriented, modular, multitasking system. It is possible to segment FORTRAN IV programs under RDOS into overlays which are stored on disk and brought into a fixed area of core as needed at execution time. Any FORTRAN IV program under RDOS can suspend its own execution and either invoke another distinct program (called program chaining) or call for a new section of itself (called program swapping).

Two programs may be executed concurrently under RDOS, a foreground and a background program. The two programs may have equal priority or the foreground program may have the higher priority of the two. Foreground and background programs are hardware protected from each other and from the operating system when the system is a NOVA* 840 computer with a Memory Management and Protection Unit (MMPU). This hardware protection enables a debugged program to be run in the foreground, while a lower priority program is constructed and debugged in the background.

The other DGC operating systems (SOS and RTOS) provide the user with compatible subsets of RDOS. The Stand-alone Operating System provides a single program runtime facility in a non-disk environment, while RTOS provides a multitasking, memory-only real time system. All file structures, task concepts, and other features described for RDOS FORTRAN IV also apply to an RTOS system unless otherwise specified in this manual.

DISCUSSION OF TERMS

Multitasking

Multitasking provides an advanced method of having multiple execution paths through a user program. Assume a program A to be operating perhaps putting together statistical information. Somewhere along the line it determines that it needs another routine (or task) to be called in to assist it in performing some calculation. The second task may then be activated, while the first task may or may not cease its operation. In fact, both tasks could be operating concurrently, each contending for the system's resources (CPU time, I/O time, core storage, etc.).

Multitasking allows a user to coordinate many independent tasks by having each task share subroutines, the access of data buffers, and disk files. Programmable control of tasks is made possible by calls which activate, make ready, and suspend a task, and those that examine the status of tasks on a group or individual basis. Other calls to the task monitor allow independently executing tasks to be synchronized or to exchange information.

System resources in the form of CPU time and I/O peripherals are allocated to each task under a user-specified task priority structure. The user can also define task subprograms in assembly language for separate incorporation as task units.

---

* NOVA, SUPERNOVA, and NOVADISC are registered trademarks of Data General Corporation.

DISCUSSION OF TERMS (Continued)

Swapping and Chaining

An executing program may invoke another program that exists as a save file on disk. The invoking program is swapped out to disk and the invoked program executes. When the invoked program terminates execution, the calling program is restored to core. Up to five levels of program swaps are permitted.

In chaining, the invoking program is not saved. It simply executes until it invokes another program that exists as a save file on disk. There is no limit to the number of programs that may be chained.

Overlays

Another method of overwriting resident core images is the overlay facility. Unlike program swapping or chaining, the overlay facility associates disk files that are the user overlays with a root program that remains core resident. Overlays overwrite each other but do not overwrite the root program. Both the overlay facility and swapping and chaining are described in detail in Chapter 5.

System Directory

Each partition or subdirectory has a directory to the files of the partition or subdirectory; the directory is named SYS. DR. The information within every SYS. DR includes file names, the length in bytes of the files, and the file's attributes and characteristics.

The structure of SYS. DR for both system file directories and subdirectories is identical. That is, SYS. DR is a randomly organized file, and the first word in each block of the file is the number of files that are listed in this block of SYS. DR. Following this word is a series of 22 octal word entries, called user file descriptions or UFDs, which describe each file. The contents of the UFD differ somewhat for link entries; links are described in Chapter 3.

File Structures

There are three types of file organization: sequential, random, and contiguous. Each type of file consists of 256-word blocks. The organization of these files is described in detail in the RDOS Manual, 093-000075, and is only briefly discussed here.

Each block of a sequentially organized file has a 255-word data area followed by a word containing a pointer to the next block. The pointer is to the logical block address assigned by the system and derived from the physical sector/track address of the disk. Logical addresses need not be accessed sequentially; a sequentially organized file might have the last word of block 7 pointing to block 14 which in turn points to block 4, etc. Sequential I/O transfers are buffered, i.e., only whole blocks are transferred and each one is read into the buffer first.

Randomly organized files utilize all 256-words of the block for data. The blocks are accessed by a file index which is created when the random file is created. The file index is a sequentially organized file of pointers to the data blocks of the random file. Each random block is assigned a sequential positive integer by its position within the file. The first block is block 0. In processing randomly organized files, two disk accesses are generally all that is required for reading and writing of each block: one to access the file index and one for the block of data itself. If the index is main-memory resident (having previously been read into a system buffer), only one access is necessary.

Contiguously organized files use all 256 words of a block for data. These are files whose blocks may be accessed randomly but without need for a random file index. Contiguous files are composed of a fixed number of disk blocks, located at an unbroken series of disk block addresses. The files cannot be expanded nor

DISCUSSION OF TERMS (Continued)

File Structures (Continued)

reduced in size. Since the data blocks are at sequential logical block addresses, all that is needed to access a block within a contiguous file is the address of the first block (or the name of the file) and the relative block number within the file.

All I/O operations which can be performed on randomly organized files can be performed on contiguously organized files, but the size of the contiguous file remains fixed. Contiguously organized files have the advantage of usually requiring less time for accessing blocks within a file, since there is no need to read a file index.

Dual Programming

Dual programming, also referred to as foreground/background programming, allows two programs to execute concurrently, sharing system resources. One of these concurrently operating programs resides in the foreground and the other resides in the background. Either the foreground program has a higher priority than the background program, or the foreground and background programs may have equal priority in competing for system resources.

The division between the foreground and background programs may be either a software memory partition (created during the relocatable load process) or a hardware partition. The hardware partition exists when a Memory Management and Protection Unit (MMPU) is included with the Nova 840 system.

Mapping

When Nova 840 systems include an MMPU, there are two modes for addressing memory. The modes are: absolute mode and user (mapped) mode. In absolute mode memory addresses are unmapped, with only the lower 31K of memory addressable. In user (mapped) mode, the background and the foreground programs can each be allotted up to 31 blocks of memory of 1024 (decimal) words each. Addresses are mapped; each user program is aware of its portion of address space only and therefore, cannot reference locations outside its own logical address space.

User Status Table (UST)

Each UST contains information describing each user program including the program's length, the number of tasks required, and the number of I/O channels required. Each program has an associated UST.

FORTRAN IV ERROR FLAGS

Many of the run time calls contain as part of their format an integer variable error that returns an error code. The possible error codes returned in the integer variable error are:

0	Indeterminate error
1	Call successfully completed
2	System action in progress
≥ 3	RDOS system error code + 3

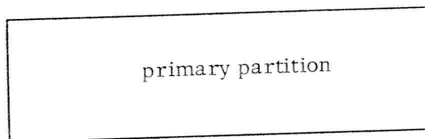
Error code 0 will be returned only as a result of a bug within the user program. Error code 1 indicates that the specified operation was successfully completed, therefore encountering no error condition. Error code 2 indicates system activity in progress; this is actually only momentarily placed within error during the time it takes to complete the operation. Error codes 0, 1, or 3 and higher are the only codes returned at completion of a call. An error code of 3 or higher indicates one of the RDOS system error codes, e.g., FORTRAN error 3 is RDOS error 0, FORTRAN error 4 is RDOS error 1, etc. A list of all error codes can be found in Appendix A.

CHAPTER 2

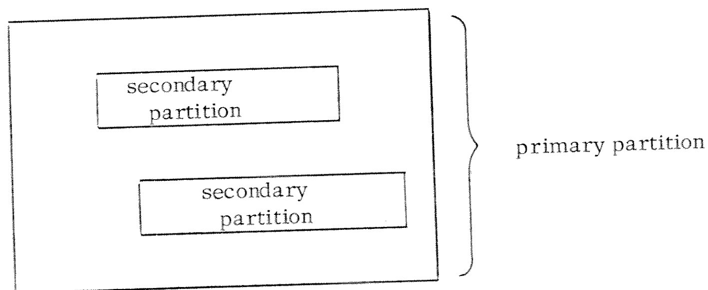
SYSTEM, DIRECTORY, AND DEVICE CONTROL

DIRECTORIES, DISKS, AND DISK PARTITIONS

A disk is a device (fixed or moving head) capable of storing information in the form of files. Total disk space (except the first six blocks which are reserved for HIPBOOT) is labeled the primary partition.



One or more portions of this primary partition may be designated as secondary partitions (created by a call to CPART).



Secondary partitions are fixed areas of contiguous file space. Within a partition a user may be allocated a subdirectory (created by a call to CDIR). Subdirectories allow users to share a partition's file space on a variable basis.

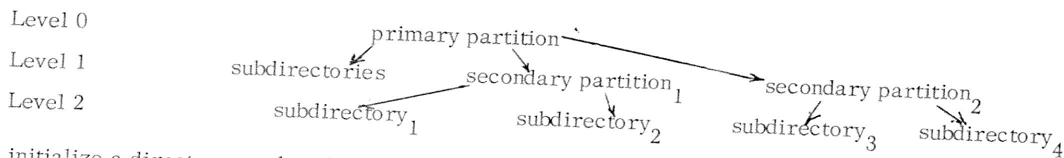
Each partition and subdirectory has a file directory called SYS.DR which contains the following types of information:

> SYS.DR for the primary partition contains information concerning each file contained within the primary partition, a list of each subdirectory associated with the primary partition, and a list of the names of each secondary partition.
>
> SYS.DR for each secondary partition contains information concerning each file within that secondary partition, and a list of all subdirectories associated with that secondary partition.
>
> Each subdirectory within a partition has its own SYS.DR containing a list of all files of that subdirectory.

DIRECTORIES, DISKS, AND DISK PARTITIONS (Continued)

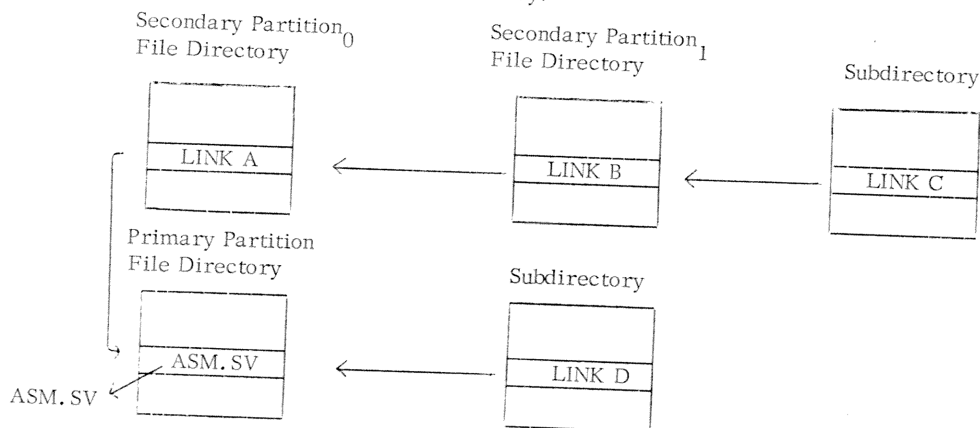
A bit allocation map (called MAP.DR) is contained within each partition. MAP.DR keeps a current record of which disk blocks are in use and which are free for data use within each partition. The primary partition's MAP.DR keeps a record of total disk file space except blocks 0 through 5. These six blocks contain the disk bootstrap program which can never be destroyed since the system is unaware of its existence. Subdirectories use the parent partition's MAP.DR as they do not have one themselves.

Since all directories (except the primary partition's directory) are listed in a parent directory, there is a hierarchy among directory specifiers. Primary partitions are at the highest level. Following these are primary partition subdirectories and secondary partitions. Lowest of all are the secondary partition subdirectories.



To initialize a directory so that the files it contains can be referenced, the user must initialize all directories in a path from the closest higher level directory which has already been initialized. If necessary, the primary partition itself must first be initialized. Thus if in the above illustration a user wished to access files in $subdirectory_1$ and no directory in this file space has been initialized, the following directories must be initialized in the following order: primary partition, secondary $partition_1$, $subdirectory_1$.

A user can also access a file using links, as described in Chapter 3. In brief, a common application of a link entry is to permit the conservation of disk file space by allowing a single copy of a commonly used disk file to be linked to by users in the same directory or partition, or in other partitions. Link entries may point to other link entries, with a depth of resolution of up to 10 decimal. The entry which is finally linked to is called the resolution entry.



FORTRAN CALLS INTERFACING TO SYSTEM DIRECTORY COMMANDS

FORTRAN calls to run time routines that interface to system directory commands are described in this section; they are:

BOOT	-	perform a disk bootstrap
CDIR	-	create a subdirectory
CPART	-	create a secondary partition
DIR	-	change the current directory
GDIR	-	get the current default directory name
INIT	-	initialize a directory or magnetic tape device
MDIR	-	get the name of the master device
RLSE	-	release a directory or magnetic tape device

Change the Current Directory (DIR)

Disk files are accessed by file name. Under RDOS a disk file name may reside in one of three kinds of directories: the primary partition's directory, the secondary partition's directory, or a subdirectory's directory. The primary partition is a fixed or moving head disk device. Secondary partitions and subdirectories result from partitioning of disk space of a primary partition among users.

When RDOS is bootstrapped (using HIPBOOT or the BOOT command as described in the RDOS Manual 093-000075), a current directory is established. Files in the current directory may be accessed by file name, e.g., if the current directory is DP0, then FILEY in DP0 is accessed by the name FILEY. However, files in directories other than the current directory can be accessed only by prefixing the file name with the specifiers of the directories in which they are found, e.g.,

DP1:PART2:FIX

where: DP1 is the primary partition and PART2 is the secondary partition containing the name of file FIX.

The user can change the current directory with a call to DIR. Thus if the current directory is changed from DP0 to PART2 of DP1, the user may access FIX without directory specifiers prefixed to the disk file name. A call to DIR also initializes the directory if it was not previously initialized. The call to DIR has the format:

CALL DIR (directoryname, error)

where: directoryname is the name of a file or device that is to become the new current directory.

error is an integer variable which will return one of the error codes upon completion of the call.

Examples of calls to DIR follow:

```
CALL DIR ("DP0", IER)
CALL DIR ('SPART', IER)
```

Initialize a Directory (INIT)

Before any file can be referenced, its directory must be initialized. Initialization opens a directory, specifying its name to the system for future access. Initialization can be either full or partial

Full initialization is used to introduce new disk packs or cartridges, magnetic tapes, or cassettes to the system; or it can be used to erase all existing files releasing their space from the partitions and subdirectories. Partial initialization is used to reintroduce to the system an entire unit with valuable file contents (i.e., a primary partition) or to reintroduce a portion of the primary partition, namely, a secondary partition or subdirectory.

Although more than one directory can be initialized at any one moment, there can be only one current default directory. The current default directory is the directory to which all file references are directed in the absence of additional directory specifier information.

A directory can be initialized by executing a call to INIT. The call has the format:

CALL INIT (directoryname, type, error)

where: directoryname is the name of the directory file or the directory device to be initialized.

Initialize a Directory (INIT) (Continued)

type is an integer constant or variable whose value determines the type of initialization to be performed:

-1 full initialization
0 partial initialization

error is an integer variable which will return one of the error codes upon completion of the call.

Partial initialization with overlays applies only to directory devices. Full initialization clears all previous files and information from the specified directory device. Subdirectories (non-device directories) are always partially initialized. See the RDOS User's Manual for further information on the initialization of directories. Examples of calls to INIT are:

```
CALL INIT ("DP1", 0, IER)
CALL INIT ("PARTITION", 0, IER)
CALL INIT ("MT0", -1, IER)
```

Release a Directory (RLSE)

The user can terminate access to files on a given directory by releasing the directory. (All files of a directory must be closed before the directory can be released.) The call to RLSE has the format:

CALL RLSE (directoryname, error)

where: directoryname is the name of the directory or directory device to be released.

error is an integer variable which will return one of the error codes upon completion of the call.

Releasing a directory is the reverse of initializing it. After a directory is released, files in that directory can only be accessed after initializing (CALL INIT) the directory again. An example of a call to RLSE is:

```
CALL RLSE "MT1", IER)
```

In the case of magnetic tape or cassette units, the RLSE call will rewind these tapes.

When the current directory is released, the master directory becomes the current directory until a new directory is specified explicitly.

Get the Current Default Directory/Device Name (GDIR)

A call to the routine GDIR returns the name of the current default directory/device. The format of the call is:

CALL GDIR (array, error)

where: array is the name of the array which will return the name of the current default directory/device. array must be large enough to accommodate 13 bytes.

error is an integer variable which will return one of the error codes upon completion of the call.

Get the Current Default Directory/Device Name (GDIR) (Continued)

An example of a call to GDIR is:

CALL GDIR (IAR, IER)

Create a Subdirectory (CDIR)

A subdirectory is a subset of the parent partition's file space. Unlike secondary partitions, subdirectories have no defined amount of file space. Instead, subdirectories take file space from the parent partition as required and release the space when it is no longer needed. A call to the routine CDIR causes an entry to be made in a primary or secondary partition's file directory for a subdirectory. The format of the call is:

CALL CDIR (name, error)

where: name is the name of the subdirectory

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to CDIR is:

CALL CDIR ("SDIR", IER)

Create a Secondary Partition (CPART)

RDOS permits the parceling of disk file space among several users, on both a fixed and semi-variable basis. Fixed parcels of a contiguous disk file are called secondary partitions.

A call to CPART creates a secondary partition and enters the name of the secondary partition in the primary partition's system directory. The primary partition can never be deleted; however, secondary partitions and subdirectories can be deleted. If a secondary partition is deleted, any subdirectories within that secondary partition are also deleted. The format of the call to CPART is:

CALL CPART (name, size, error)

where: name is the name to be assigned to the newly created secondary partition.

size is an integer constant or variable indicating the number of contiguous blocks in the secondary partition.

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to CPART is:

CALL CPART ("SECP", 14, IER)

Get the Logical Name of the Master Device (MDIR)

Since inter-device bootstrapping is possible under RDOS, the current master device may not be the master device which was defined at the time of system generation. A call to the routine MDIR permits the user to determine the current name for the current master device. The format of the call is:

CALL MDIR (array, error)

Get the Logical Name of the Master Device (MDIR) (Continued)

where: array is the name of an array which will return the name of the master device.

error is an integer variable which will return one of the error codes upon completion of the call.

The master directory device is a primary or a secondary partition which becomes the current directory device after either a full system initialization or a disk bootstrap. The master device contains all of the system overlays.

An example of a call to MDIR is:

CALL MDIR (IAR, IER)

Perform a Disk Bootstrap (BOOT)

A call to the BOOT routine causes all open files in a currently executing system (both foreground and background)* to be closed, all directories to be released, and all system I/O to be reset. Control is then transferred to HIPBOOT which will bootstrap a new operating system.

Bootstrapping may be performed with or without operator intervention. With operator intervention, the format of the call to BOOT is:

CALL BOOT (partition, error)

where: partition is the name of the partition containing HIPBOOT.

error is an integer variable that will return one of the error codes upon completion of the call.

An example of a call with operator intervention is:

CALL BOOT ("DP0", IER)

which will load HIPBOOT from moving head disk unit 0. When loaded, HIPBOOT queries the user with FILENAME?, requesting the name of the system to be bootstrapped. A system file response is then given by the user as described in Appendix D.

Without operator intervention, the following conditions must be fulfilled: (1) the operating system to be bootstrapped must be in a primary partition, (2) HIPBOOT must be in the same partition as the operating system, (3) the default operating system, SYS.SV, must be the system to be bootstrapped, (4) the user must have placed -1 in the CPU data switches (all switches in up position), and (5) the user must provide a save file named RESTART.SV that will perform whatever restart procedures are necessary to resume control of the real time process that was interrupted. When HIPBOOT bootstraps SYS.SV, the new system control will be chained to RESTART.SV. RESTART.SV must be in the same primary partition with SYS.SV. The time and date are not updated automatically and must be set by the user.

With operator intervention, the format of the call to BOOT is the same as without operator intervention, except that the partition specified must contain, HIPBOOT, SYS.SV and RESTART.SV.

Get the Name of the Current System (GSYS)

A call to the GSYS routine will return the name of the current operating system. The name returned will consist of the name plus its two-character extension terminated by a null terminator. The format of the call is:

CALL GSYS (array, error)

*BOOT should not be issued from the background when the foreground is active.

Get the Name of the Current System (GSYS)(Continued)

where: array is the name of the array which will return the name of the current operating system. The array must be large enough to accommodate 15 octal bytes.

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to GSYS is:

CALL GSYS (IAR, IER)

DEVICE CONTROL

Disable Console Interrupts (ODIS)

A call to the routine ODIS will permit the user to prevent console interrupts from occurring within his program environment. CTRL A, CTRL C, and CTRL F console interrupts may not occur unless reenabled by a call to OEBL. The format of the call is:

CALL ODIS

Enable Console Interrupts (OEBL)

By default, when a system is first bootstrapped, console interrupts CTRL A, CTRL C, and CTRL F are enabled. If console interrupts have been disabled by a call to the routine ODIS, this call re-enables them within its program environment. The format of the call is:

CALL OEBL

Enable Spooling (SPEBL)

Simultaneous peripheral operation on-line (spooling) has been implemented for the following devices:

$LPT	$LPT1
$PLT	$PLT1
$PTP	$PTP1
$TTO	$TTO1
$TTP	$TTP1

Spooling permits the queuing of data for one or more spoolable devices, making the CPU available for further processing while those devices receive the queued data. Spooling occurs only when no other system operations are ready. (System operations are given a higher priority than any user tasks.) Spooling is possible in a single program environment only if two or more system stacks have been allocated at SYSGEN time; a dual program environment requires three or more system stacks. When an insufficient number of system stacks is allocated, all spooling commands become no-ops. Since spooling requires disk buffers, the system will disable spooling if no free disk space is available at the time spooling is attempted. The user may re-enable spooling at some later time when sufficient disk buffer space becomes available.

A call to SPEBL will enable spooling on a device for which spooling had been previously disabled. The format of the call is:

CALL SPEBL (devicename, error)

DEVICE CONTROL (Continued)

Enable Spooling (SPEBL) (Continued)

where: devicename is the name of the device which the user wishes to be a spoolable device.

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to SPEBL is:

```
CALL SPEBL ("$LPT", IER)
```

Disable Spooling (SPDIS)

The call to SPDIS causes a spoolable device to discontinue spooling its output. If this call is issued while a device is spooling, execution of the call will be delayed until all data waiting to be spooled has been output. Data output to the device before the spooled data has been exhausted will itself be spooled to the output device, delaying execution of the call to SPDIS even longer. The format of the call is:

CALL SPDIS (devicename, error)

where: devicename is the name of the device which will no longer be a spoolable device.

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to SPDIS is:

```
CALL SPDIS ("$LPT", IER)
```

Stop a Spool Operation (SPKIL)

It is possible to stop a spool operation which is currently being performed, losing any data which was in the output queue. The format of the call to SPKIL is:

CALL SPKIL (devicename, error)

where: devicename is the name of the device currently spooling the output which is to stop.

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to SPKIL is:

```
CALL SPKIL ("$LPT", IER)
```

USER INTERRUPT SERVICING

Users who wish to incorporate non-SYSGENed devices into real time FORTRAN programs must provide for the interrupt servicing to be done in assembly language, and for the creation of a three-word device control table (DCT) as explained in the RDOS User's Manual, 093-000075.

Interrupt requests from special (non-SYSGENed) devices do not, for the most part, change the status of tasks in a FORTRAN multitask environment. Instead, such interrupts freeze the environment until servicing of the interrupt is completed and the multitask environment is unfrozen. Likewise, all other tasks will resume their former states when the environment becomes unfrozen, unless the user transmits a message to one of them by means of the transmit interrupt message command .IXMT.

It is still necessary, however, to identify the interrupt device to the system by means of a FORTRAN call (FINTD) and it is possible to remove this device from the system by means of another FORTRAN call (FINRV). Interrupt servicing and the FORTRAN run time routines FINTD and FINRV may be used in both single and multitask environments.

Identifying a User Interrupt Device (FINTD)

The FINTD routine is used to identify to the system a device that is capable of generating interrupt requests but which was non-SYSGENed. The format of the call to FINTD is:

CALL FINTD (device-code, dct)

where: device-code is an integer variable or constant which is the code of the user device, where device-code is less than 63.

dct is the name of a three-word device control table which may be a dimensioned array or an externally defined item. The dct entries are defined in the RDOS Manual, 093-000075. Since the third entry is an address, the dct is usually handled in assembly language and defined as an external to the FORTRAN program rather than an array.

Those devices that were not identified to the system at SYSGEN time must be made known to the system by the FINTD routine. FINTD causes an entry for the specified device code to be placed in the system interrupt vector table.

An example of a call to FINTD is:

CALL FINTD(62, IDDCT)

where: IDDCT is defined in the program either as

EXTERNAL IDDCT

or

DIMENSION IDDCT (3)

Note that if IDDCT is an array, it must always be accessible in case of an interrupt, i. e., it should be in labeled or unlabeled COMMON.

There is a special usage of a call to FINTD to provide for automatic restart of user-defined devices and system devices to which power-up service is not extended after a power failure. Users having the power monitor/automatic restart hardware may make use of this call to provide power-up service in a user-written routine. The call has the format:

CALL FINTD(63, name)

where: name is the name of an externally declared user-written routine that provides the power-up interrupt servicing.

63 is the device code of the CPU.

An example of this special usage call to FINTD is:

```
EXTERNAL IPRUP
CALL FINTD(63, IPRUP)
```

USER INTERRUPT SERVICING (Continued)

Remove a Service Interrupt Device (FINRV)

A previously added (FINTD) user interrupt device can be removed from the system interrupt vector table by a call to FINRV. The format of the call to FINRV is:

CALL FINRV (device-code)

where: device-code is an integer variable or constant which must be the device code of a previously identified user interrupt device.

If an attempt is made to remove a SYSGENed device or if the device code argument is not within the legal range of user interrupt devices (less than 63), a fatal run time error occurs and execution is terminated.

An example of a call to FINRV is:

CALL FINRV (23)

CHAPTER 3

FILE MAINTENANCE AND I/O CONTROL

FILES, FILE NAMES

A file is a collection of information or any device receiving or providing this information. All devices and disk files are accessible by file name. File names are byte strings of ASCII characters, packed left to right and terminated by a carriage return, form feed, space or null. Allowable ASCII characters are all upper case alphabetics, all numerics, the character dollar sign ($) and the character colon (:).

A file name may consist of any number of characters, but only the first ten are considered significant (in addition to a two-character extension preceded by a period). Therefore, file names must be unique within their first ten characters.

ABCDEFGHIJ   is equivalent to   ABCDEFGHIJKL

I/O devices are given reserved device names; the list of these reserved names is given below. Where second devices/controllers are allowed, the name appears in the second column.

Device	Reserved Device Name	
incremental plotter	$PLT	$PLT1
teletypewriter punch	$TTP	$TTP1
card reader	$CDR	$CDR1
teletypewriter printer or display unit screen	$TTO	$TTO1
teletypewriter or display unit keyboard	$TTI	$TTI1
80 or 132 column line printer	$LPT	$LPT1
high-speed paper tape reader	$PTR	$PTR1
high-speed paper tape punch	$PTP	$PTP1
teletypewriter reader	$TTR	$TTR1
magnetic tape unit n (n = 0 to 7)	MTn	MT1n
cassette unit n (n = 0 to 7)	CTn	CT1n
DGC NOVADISC fixed head unit	DK0	DK1
moving head disk unit n	DPn (n=0-3)	DPn (n=4-7)
input dual processor link	$DPI	
output dual processor link	$DPO	
multiprocessor communications adapter receiver	MCAR	MCAR1
multiprocessor communications adapter transmitter	MCAT	MCAT1
asynchronous data communications multiplexor	QTY:nn (nn = 0 - 64 = line number)	

Under FORTRAN IV, when writing a file name, that file name must appear either within quotation marks (quotes), or within apostrophes (sometimes referred to as single quotes). For example:

"PTP" 'CT0:6' "ABC" "TEST" 'TEST.SV'

In the call formats appearing on following pages, the variables filename and devicename often appear. For example:

CALL DFILW (filename, error)

CALL SPEBL (devicename, error)

FILES, FILE NAMES (Continued)

File names and device names follow the RDOS naming conventions, i.e., the name is a string of upper case ASCII alphabetic characters, numerals or the $ character. While the file name may be any length, only the first 10 are considered significant. A literal file or device name appearing in a FORTRAN IV call is enclosed in quotation marks or apostrophes or the file name may be passed as part of a string array.

Referencing a File

A file must be opened (i.e., associated with a channel number) before it can be accessed. The channel number may have been pre-assigned (see list below) or may be user-assigned in a call to OPEN or FOPEN. Any of the 64 channels (0-63) can be associated with any file or device in a call to OPEN or FOPEN (even if the file/device already has a pre-assigned number). The pre-assigned channel/device number is temporarily suspended for the duration of the call to OPEN or FOPEN (e.g., a call to CLOSE, FCLOS, or RESET will disassociate the channel/device number).

The pre-assigned channel numbers (with foreground associations listed within parentheses) are as follows:

Device	Channel
$PLT	6
$TTP	8
$CDR	9
$TTO ($TTO1)	10
$TTI ($TTI1)	11
$LPT	12
$PTR	13
$PTP	14
$TTR	15

Note that when issuing a TYPE statement, channel 10 is associated with either $TTO or $TTO1 (depending on whether executing in the background or the foreground) and when issuing an ACCEPT statement, channel 11 is associated with either $TTI or $TTI1. Both of these channel associations are made without issuing a call to the routine FOPEN or OPEN.

Referencing a File on Magnetic Tape or Cassette Units

Files are placed on tape in numeric order (0 - 99). A given file is referenced by the device name followed by a colon followed by the file number:

CT*n*:*m* (CT1*n*:*m*) or MT*n*:*m* (MT1*n*:*m*)

where: *n* is the unit number (0 to 7)

m is the file number (0 - 99)

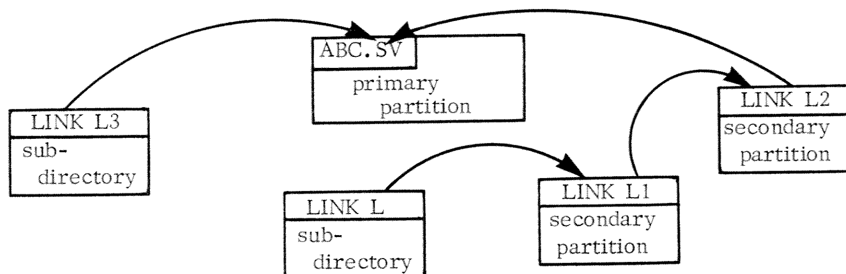
CT*n*, CT1*n*, MT*n*, and MT1*n* are the default names of the particular devices. It is possible to change these device names by a call to EQUIV.

FILES, FILE NAMES (Continued)

Links, Link Entires

Users can access any disk file, magnetic tape file, or cassette file by its name or by several different names (called aliases). By using link entries, users can access files outside their own directories. (For those readers not familiar with directories, turn to Chapter 2.) A single copy of a commonly used disk file can then be linked to by several users in the same or in different partitions, resulting in a conservation of total disk file space.

These link entries may in turn point to other link entries, and so on, up to a depth of ten (decimal). This depth is referred to as the depth of resolution, as the final file linked to is called a resolution file, or resolution file entry.



Link entries are created by a call to DLINK, and can be deleted by a call to DULNK.

Whenever a link is to be resolved (i.e., when the link is opened), the directory containing the resolution entry is initialized by the system if not already initialized. However, when the link entry to this file is created, the pertinent directory containing the resolution file need not be initialized, in fact, the resolution file need not even exist at this time. The link entry name must be unique within its own directory.

Looking at the diagram above, four links exist to the resolution entry for the file ABC.SV. In order for any given link to be resolvable, all intermediate links must be resolvable. Thus, if LINK L1 is unlinked, LINK L is no longer resolvable. LINK L2 and LINK L3 will be resolvable at that point, however.

Each resolution entry contains two kinds of attributes:

resolution file attributes
link access attributes

Resolution file attributes apply to direct users of files. Link access attributes specify file attributes for users linking to these files. The attributes for the resolution entry are set when the resolution file is created and could have been subsequently changed by the user (by a call to FSTAT). Link access attributes are initially set to zero, but are subsequently changed by the user by a call to CHLAT).

After a user has opened a file via a link entry, that file's attributes can be changed via a call to FSTAT. These attributes are in effect for only the length of time the file is open via the link entry.

FILE MAINTENANCE

Assign a New Name to a Multiple File Device (EQUIV)

A call to the routine EQUIV assigns a temporary name to a multiple file device, permitting unit independence during the execution of a FORTRAN program. Thus magnetic tape file references might be made to temporary name MTAPE in a FORTRAN program, with assignment to a specified magnetic tape transport unit (e.g., MT6) at run time by means of a call to EQUIV. No temporary name can be assigned to the master device.

The call must be issued before the device is intitialized. Temporary names persist until either a disk bootstrap, release, or new temporary name assignment is made. The format of the call is:

CALL EQUIV (name1, name2, error)

where: name1 is the reserved or most recently assigned name of the multiple file device.

name2 is the temporary name of the multiple file device.

error is an integer variable which will return one of the error codes upon completion of the call.

The devices which can be equivalenced are:

CT0 - CT7, CT10 - CT17
MT0 - MT7, MT10 - MT17
DP0 - DP7
DK0, DK1

An example of a call to EQUIV is:

CALL EQUIV ("MT6", "MTAPE", IER)

Create an RDOS Disk File (CFILW)

An RDOS disk file is created by executing a call to the CFILW routine. The call has the format:

CALL CFILW (filename, type { , size }, error)

where: filename is the name to be assigned to the new file.

type is an integer constant or variable whose value indicates the type of the file to be created, either:

1 Sequentially organized file
2 Randomly organized file
3 Contiguously organized file

size is an integer constant or variable giving the size in number of blocks (256 words) of a contiguously organized file. This argument is used only for type 3 (contiguous) files.

error is an integer variable which will return one of the error codes upon completion of the call.

FILE MAINTENANCE (Continued)

Create an RDOS Disk File (CFILW) (Continued)

The name filename must be unique with respect to all other file names in the system. The size of a contiguously organized file must be specified by size and cannot be changed after the file has been created. An example of a call to CFILW is:

```
CALL CFILW ("Y10", 3, 20, IER)
```

Delete an RDOS Disk File (DFILW)

An RDOS disk file may be deleted by issuing a call to DFILW. The file must be closed before being deleted. The call has the format:

CALL DFILW (filename, error)

where: filename is the name of the file to be deleted.

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to DFILW is:

```
CALL DFILW ("DATA12", IER)
```

Delete a File (DELETE)

Files may be deleted using the library routine DELETE (although DFILW is preferred to DELETE). The call takes the form:

CALL DELETE (filename)

where: filename is the name of the file to be deleted.

The file specified will be deleted from the system directory if it exists and is not open. Only closed files can be deleted. If the file is currently open, an error message will be returned. An example of a call to DELETE is:

```
CALL DELETE ("DATAFILE")
```

Renaming a File (RENAM)

A disk file may be renamed by executing a call to the RENAM routine. The format of the call is:

CALL RENAM (oldfilename, newfilename, error)

where: oldfilename is the file name that is to be changed.

newfilename is the new name which is to be assigned to the file.

error in an integer variable which will return one of the error codes upon completion of the call.

An example of a call to RENAM is:

```
CALL RENAM ("TEST", "SORT", IER)
```

Create a Link Entry (DLINK)

A call to the routine DLINK creates a link entry in the current directory to a file in another directory. The file being linked to (i.e., the resolution entry) may have the same name as that specified in the link entry, or the link and file names may differ (i.e., the link entry name is an alias.) No attributes are applied to a link except the link characteristic. All access rights to the linked file are determined by an inclusive OR of the resolution entry's attributes and the link access attributes of the resolution entry. (Link access rights of each resolution entry can be altered by means of a call to CHLAT.)

The format of the call to DLINK is:

CALL DLINK (name1, { name2, } error)

where: name1 is the name of the link entry.

name2 is the name of the alternate directory, alternate partition, or the alias name. name2 is omitted if the resolution entry has the name name1 in the current primary partition.

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to DLINK is:

CALL DLINK ("ABC.SV", "ABB.SV", IER)

Delete Link Entries in the Current Directory (DULNK)

This call deletes a link entry (created earlier by DLINK) in the current directory. This call does not delete other links of the same name in other directories. Care must be exercised to ensure that the link being deleted is not required by links further removed from the resolution entry, or else the deletion of this link will render these more remote links unresolvable. The format of the call is:

CALL DULNK (name, error)

where: name is the name of the link entry to be deleted.

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to DULNK is:

CALL DULNK ("ABC.SV", IER)

Get File Directory Information for a Given Channel (CHSTS)

A call to the routine CHSTS returns a copy of the current directory status information for whatever file is currently opened on a specified channel. The format of the call is:

CALL CHSTS (channel, array, error)

where: channel is an integer constant or variable whose value specifies the number of the channel on which the device was opened.

array is an integer array which will return a copy of the 22 octal word UFD for the current file. (See chart for STAT call.)

Get File Directory Information for a Given Channel (CHSTS) (Continued)

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to CHSTS is:

CALL CHSTS (ICHAN, IAR, IER)

Get Current File Directory Information (STAT)

A call to the routine STAT allows the user to get a copy of the current directory status information for a specified file. This call causes a copy of the 22 (octal) word UFD to be written into a user-specified area. The file whose UFD is being copied need not be open at the time this call is issued. If the file is open, however, the information returned is a "snapshot" of the UFD as it existed at the time of the most recent OPEN.

Following is a template of the UFD with displacement mnemonics:

Displacement	Mnemonic	Contents
0-4	UFTFN	File name
5	UFTEX	Extension
6	UFTAT	File attributes
7	UFTLK	Link access attributes
10	UFTBK	Number of the last block in the file
11	UFTBC	Number of bytes in the last block
12	UFTAD	Starting logical block address of the file (random file index for random files)
13	UFTAC	Year/day last accessed
14	UFTYD	Year/day created
15	UFTHM	Hour/minute created
16	UFTP1	UFD temporary
17	UFTP2	UFD temporary
20	UFTUC	User count
21	UFTDL	DCT link

Link UFDs assign mnemonics UFLAD and UFLAN to words 7 and 14. Also in link UFDs, words 7 - 13 and 14 - 20 are reserved for an alternate directory specifier (if any) and an alias (if any) respectively.

The format of the call to STAT is:

CALL STAT (filename, array, error)

where: filename is the name of the file for which status information is to be copied.

array is an integer array which will return the status information.

error is an integer variable which will return an error code upon completion of the call.

An example of a call to STAT is:

CALL STAT ("Y10", IAR, IER)

Update the Current File Size (UPDATE)

A call to the run time routine UPDATE permits a file's size information to be updated without first closing the file. Specifically, this call causes information in UFTBK and UFTBC in the UFD on disk to be updated with current information for the file opened on a specified channel, and it flushes all system buffers to ensure that the file contains all information which has been written into it by the user. The format of the call to UPDATE is:

CALL UPDATE (channel, error)

where: channel is an integer variable or constant which specifies the channel on which the file is currently open.

error is an integer variable which will return one of the error codes.

An example of a call to UPDATE is:

CALL UPDATE (ICHAN, IER)

FILE ATTRIBUTE MAINTENANCE

File attribute calls allow the user to determine the current attributes of a file or device and to change the file attributes if desired.

Examine the Attributes of a File (GTATR)

The GTATR call will obtain for examination by the user the attributes of a file. To obtain attributes, the file must first have been opened on the channel number specified within the GTATR command. The call to GTATR has the format:

CALL GTATR (channel, attributes, error)

where: channel is an integer constant or variable whose value specifies the number of the channel on which the file whose attributes are to be examined is opened.

attributes is an integer variable whose value is set to represent the attributes of a file.

error is an integer variable which will return one of the error codes upon completion of the call.

The representation of attributes is as follows:

Bit	Meaning
1B0	Read-protected file. Cannot be read.
1B1	Attribute-protected file. Attributes cannot be changed.
1B2	Save file (core image file).
1B3	Link entry.
1B4	Partition.
1B5	Directory file.
1B6	Link resolution (temporary). Some or all of the attributes persist for the duration of the open.
1B7	No link resolution allowed.
1B9	User attribute.
1B10	User attribute.
1B11	Reserved
1B12	Contiguous file.
1B13	Random file.
1B14	Permanent file. Cannot be deleted or renamed.
1B15	Write-protected file. Cannot be written.

FILE ATTRIBUTE MAINTENANCE (Continued)

Examine the Attributes of a File (GTATR) (Continued)

An example of a call to GTATR is:

CALL GTATR (5, IAT, IER)

Change, Add, or Delete File Attributes (FSTAT)

The call to FSTAT causes a file's attributes (or its resolution attributes, in the case of a link) to be changed as specified by the user. If this call is issued by a link user, his copy of the file attributes is temporarily changed until he closes the file; the resolution attributes persist. To change the attributes of a file, the file must first be opened. The format of the call to FSTAT is:

CALL FSTAT (channel, attributes, error)

where: channel is an integer constant or variable whose value specifies the number of the channel on which the file whose attributes are to be changed is opened.

attributes is an integer constant or variable whose value specifies the attributes to be assigned to the file.

error is an integer variable which will return one of the error codes upon completion of the call.

The representation of the attributes which may be assigned to attributes is the same as listed on page 3-8 for the GTATR call. An example of a call to FSTAT is:

CALL FSTAT (12, 1, IER)

Change or Add Link File Access Attributes (CHLAT)

This call causes the user's copy of the link access attributes word to be changed. When a file is opened via a link entry, the attributes of the file as seen by the user are formed by the inclusive OR of the resolution entry's attributes and the user's copy of the link access entry attributes. The link access entry attributes are zero by default. The format of the call is:

CALL CHLAT (channel, attributes, error)

where: channel is an integer constant or variable whose value specifies the number of the channel on which the file whose attributes are to be changed is opened.

attributes is an integer constant or variable whose value specifies the attributes to be assigned to the file.

error is an integer variable which will return one of the error codes upon completion of the call.

The representation of attributes is as shown on page 3-8 for the GTATR call. An example of a call to CHLAT is:

CALL CHLAT (5, IAT, IER)

FILE INPUT/OUTPUT

Get the Name of the Current Input/Output Console (GCIN, GCOUT)

Before opening the console device, the user might find it necessary to find out which device is the current console to be used for input and which is the current console to be used for output. This is accomplished by issuing a call to either the routine GCIN or the routine GCOUT. The format of the two calls is:

CALL GCIN (array) ←—— get the name of the current input console ($TTI or $TTI1)

CALL GCOUT (array) ←—— get the name of the current output console ($TTO or $TTO1)

where: array is the name of an integer array that will contain the console name requested.

An example of a call to GCIN and GCOUT is:

CALL GCIN (IAR)

CALL GCOUT (IARR)

Opening Files

Open a File (OPEN)

An RDOS disk file may be opened by executing a call to OPEN. The call has the format:

CALL OPEN (channel, filename, {mode / array}, error [, size])

where: channel is an integer variable or constant whose value specifies the number of the channel (0 - 63) on which filename is opened.

filename is the name of the file which is to be opened.

mode (an alternate argument in the command line) is an integer constant or variable whose value indicates the mode of the file being opened, either:

1 - open for reading only
3 - open for writing by one user but for reading by one or more users
other than 1 or 3 - open for user-shared reading and writing

array (an alternate argument in the command line) is a three-element integer array whose elements contain the following information:

First element: contains -1 (the array flag)

Second element: contains either:

1 - open for reading only
3 - open for writing by one user
other than 1 or 3 - open for user-shared reading and writing

Third element: contains the device characteristic mask (see next page)

error is an integer variable which will return one of the error codes upon completion of the call.

size is an argument which is an integer constant or variable specifying the number of bytes that are to make up a record of a random file. size must be given only if the file is random.

FILE INPUT/OUTPUT (Continued)

Opening Files (Continued)

Open a File (OPEN) (Continued)

When a call to OPEN is executed, the file with the specified name filename will be opened on the channel specified by the value of channel. If filename is a sequentially organized file to which information is to be written, all previous information contained in the file will be overwritten. Note that if a TYPE or ACCEPT statement was issued before the call to OPEN, channel numbers 10 or 11, respectively, are already open unless closed by the user.

The bit/characteristic correspondence used in setting the device characteristic mask is:

Bit	Meaning
1B0	Spooling enabled (0B0 is spooling disabled)*
1B1	80-column device
1B2	device changing lower case ASCII to upper case
1B3	device requiring form feeds on opening
1B4	full word device (reads or writes more than a byte)
1B5	spoolable device*
1B6	output device requiring line feeds after carriage returns
1B7	input device requiring a parity check; output device requiring parity to be computed
1B8	output device requiring a rubout after every tab
1B9	output device requiring nulls after every form feed
1B10	a keyboard input device
1B11	a teletype output device
1B12	output device without form feed hardware
1B13	device requiring operator intervention
1B14	output device requiring tabbing hardware
1B15	output device requiring leader/trailer

If an MCA line is being opened, the third element of the array cannot contain a characteristic inhibit mask. Instead, for receiver lines the word must be cleared to zero. If a transmitter line is to be opened and the default number of retries (requiring 655 seconds) is to be used, the element must again be cleared to all zeros. However, if a different timeout value is to be specified, bit 15 of the element must be set to one (and all other bits must be cleared). The actual specification of a retry count will be deferred to the time the call to WRITR is issued.

Examples of calls to OPEN follow:

CALL OPEN (3, "TEST", 2, IER, 128)

CALL OPEN (5, "X45", IAR, IER)

Open a File (FOPEN)

A call to the routine FOPEN will assign a specified channel number to a device or to a disk file. By default, a disk file will be opened in random mode and a device will be opened in sequential mode. The call to FOPEN has the following format:

CALL FOPEN (channel, filename [, "B"] [, recordbytes])

where: channel is an integer constant or variable with a value between 0 and 63_{10}.

* Cannot be changed by an OPEN command.

filename is a string constant or array name. The array is initialized to an ASCII string by a DATA statement or is input using the S (not the A) FORMAT descriptor. (When using the Stand-alone Operating System, filename is ignored and channel is opened to its pre-assigned device.)

"B" indicates that the file is opened with all device characteristics inhibited. (This inhibits such functions as outputting a rubout after a tab to the paper tape punch.)

recordbytes implies a random file record and is the length in bytes of the random file record referenced as an integer constant or a variable.

Examples of calls to FOPEN are:

```
CALL FOPEN (ICH, "RFILE", "B", 200)
CALL FOPEN (3, 'DATAFILE', 40)
CALL FOPEN (4, '$PTR', 'B')
```

Open a File for Appending (APPEND)

A file is opened for appending by executing a call to the APPEND routine. The call has the format:

CALL APPEND (channel, filename, {mode | array}, error [, size])

where: channel is an integer variable or constant whose value specifies the number of the channel (0 - 63) on which filename is appended to.

filename is the name of the file to be opened for appending.

mode (an alternate argument in the command line) is an integer constant or variable whose value indicates the mode of the file being appended to, either:

1	- open for reading only
3	- open for writing (by one user only, though one or more users may open it for reading.
other than 1 or 3	- open for user-shared reading and writing.

array (an alternate argument in the command line) is a three-element integer array whose elements contain the following information:

First element:	contains -1 (the array flag)	
Second element:	contains either:	
	1	opened for reading only
	3	opened for writing by one user
	other than 1 or 3	opened for user-shared reading and writing
Third element:	contains the device characteristics mask (as listed for the OPEN call).	

error is an integer variable which will return one of the error codes upon completion of the call.

size is an integer constant or variable specifying the number of bytes that are to make up a record of a random file. size must be given only if the file is random.

When a call to APPEND is executed, the end of the file filename is located and filename is opened on the specified channel. Subsequent output of the file is appended to the data already there. An example of a call to APPEND is:

```
CALL APPEND (5, "SQRT", 2, IERR, ISIZ)
```

Closing Files

Close a File (CLOSE)

An RDOS file may be closed by a call to the routine CLOSE (which is preferred to FCLOS). The call has the format:

```
CALL CLOSE (channel, error)
```

where: channel is an integer variable or constant whose value specifies the channel number associated with the file to be closed.

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to CLOSE is:

```
CALL CLOSE (14, IER)
```

Close a File (FCLOS)

The FCLOS routine may also be called to free a channel and to close a file on the specified channel. The format of the call is:

```
CALL FCLOS (channel)
```

where: channel is an integer constant or variable with a value between 0 and 63_{10} specifying the channel which the user wishes to free.

An example of a call to FCLOS is:

```
CALL FCLOS (10)
```

Close all Open Files (RESET)

All open files can be closed by issuing a call to RESET. The call has the format:

```
CALL RESET
```

Reading and Writing Blocks and Records

Read a Series of Blocks (RDBLK)

A series of blocks can be read from a contiguously or randomly organized file without utilizing a system buffer by executing a call to the RDBLK routine. The call has the format:

CALL RDBLK (channel, sblock, array, nblock, error { , iblk })

where: channel is an integer constant or variable whose value specifies the number of the channel on which the contiguously organized file to be read from is opened.

sblock is an integer constant or variable whose value specifies the number of the first block to be read.

array is the name of an integer array that is to receive the blocks that are read. The array must be nblock * 256 words in length. (No error check is made on the adequacy of the array length.)

nblock is an integer constant or variable whose value specifies the number of consecutive blocks to be read.

error is an integer variable which will return one of the error codes upon completion of the call.

iblk is an optional integer variable that will be set to return the number of blocks read on encountering an EOF.

An example of a call to RDBLK is:

CALL RDBLK (10, 100, IARR, 15, IER, IBLK)

Execution of this call causes 15 contiguous blocks to be read, starting from the 100th block, into array IARR. The IARR array must have been previously dimensioned to a length of 3840 words

Read a Series of Records (READR, RDRW)

A series of records can be read from a randomly organized file into an integer array by executing a call to READR or a call to RDRW. The calls have the formats:

CALL READR (channel, srec, array, nrec, error { , nbyte })

CALL RDRW (channel, srec, array, nrec, error { , nbyte })

where: channel is an integer variable or constant whose value specifies the number of the channel on which the random file to be read is opened.

srec is an integer constant or variable whose value specifies the number of the first record to be read.

array is the name of an integer array that is to receive the records to be read. (There is no check on the adequacy of the array length.)

nrec is an integer constant or variable whose value specifies the number of successive random records to be read.

Read a Series of Records (READR, RDRW) (Continued)

error is an integer variable which will return one of the error codes upon completion on the call.

nbyte is an optional integer variable that returns the byte count read if an EOF or disk full is encountered.

CALL READR (15, 0, IARR, 20. IERR)

Write a Series of Records (WRITR, WRTR)

A series of records can be written into a file by executing a call to the WRITR routine or to the WRTR routine. The calls have the formats:

CALL WRITR (channel, srec, array, nrec, error [, nbyte])

CALL WRTR (channel, srec, array, nrec, error [, nbyte])

where: channel is an integer constant or variable whose value specifies the number of the channel on which the file to be written is opened.

srec is an integer constant or variable whose value specifies the number of the first record to be written.

array is the name of the integer array that contains the information to be written. (No check is made on the adequacy of the array's length.)

nrec is an integer constant or variable whose value specifies the number of consecutive records to be written.

error is an integer variable which will return one of the error codes upon completion of the call.

nbyte is an optional integer variable that returns a partial byte count if an EOF or disk full is encountered.

CALL WRITR (12, IRECD, IRAR, NRECD, IER, NBYTE)

Write a Series of Blocks (WRBLK)

A series of blocks may be written into a contiguous or random disk file, without intermediate system buffering, from an integer array by executing a call to WRBLK routine. The call has the format:

CALL WRBLK (channel, sblock, array, nblock, error [, iblk])

FILE INPUT/OUTPUT (Continued)

Reading and Writing Blocks and Records (Continued)

Write a Series of Blocks (WRBLK) (Continued)

where: channel is an integer constant or variable whose value specifies the number of the channel (0 - 63) on which the file to be written into is opened.

sblock is an integer constant or variable whose value specifies the number of the first block to be written.

array is the name of an integer array that contains the blocks that are to be written. The array must be 256 * nblock words in length, but no check is made on the adequacy of the array's length.

nblock is an integer variable or constant specifying the number of blocks to be written.

error is an integer variable which will return one of the error codes upon completion of the call.

iblk is an optional integer variable that is set equal to the number of blocks written should a disk full occur.

An example of a call to WRBLK is:

```
CALL WRBLK (12, 200, IARR, IBLK, IERR, IBYTE)
```

FREE FORMAT CASSETTE OR MAGNETIC TAPE I/O

Open a Cassette or Magnetic Tape Unit for Free Format I/O (MTOPD)

Before free format reading or writing can be performed on either an initialized magnetic tape or cassette unit, the device must be opened and linked to a channel. The routine to open files or devices (OPEN) cannot be used to open a magnetic or cassette tape unit for free format I/O; only MTOPD can be used to open these devices for this purpose.

A call to MTOPD positions a free format tape to a desired file, since the file name argument given to MTOPD includes both the unit name and the file number (MTn:m or CTn:m). The format of the call is:

```
CALL MTOPD (channel, filename, mask, error)
```

where: channel is an integer constant or variable whose value specifies the number of the channel on which filename is to be opened.

filename is the name of the magnetic tape unit or cassette unit to be opened.

mask is the device characteristic mask.

error is an integer variable which will return one of the error codes upon completion of the call.

For a list of the bit/characteristic correspondences see the OPEN call.

Open a Cassette or Magnetic Tape Unit for Free Format I/O (MTOPD) (Continued)

An example of a call to MTOPD is:

CALL MTOPD (16, "MT6:1", 0, IER)

Free Format Tape I/O (MTDIO)

Before free format I/O can be performed on a tape unit, that unit must first have been opened for free format I/O by means of a call to MTOPD. The call to MTDIO permits the operation of magnetic tape and cassette units on a machine level: reading and writing of records in variable length records or to the start of a new data file, and performing of other similar machine level operations. Free format I/O is entirely under user control; the user must check for proper formatting when using MTDIO. The format of the call is:

CALL MTDIO (channel, commandword, I/O-array, status, error { , record-count / , word-count })

where: channel is an integer constant or variable specifying the channel number (0 - 63) on which the device was opened.

commandword is an integer constant or variable whose bits specify which operation is to be performed as follows:

Bit	Meaning
0	Parity bit (1 = even, 0 = odd)
1-3	0 - read (words) 1 - rewind the tape 3 - space forward (over records or over file of any size) 4 - space backward (over records or over file of any size) 5 - write (words) 6 - write end of file 7 - read device status word
4-15	Word or record count. If 0 on a space forward (or space backward) command, the tape is positioned to the beginning of the next (or previous) file on the tape. If 0 on a read or write command, 4096 words are read (or written) unless an end of record is detected.

I/O-array is an integer array used for transmitting and receiving data. (In many instances, MTDIO is not used for data transfer, e.g., when status is requested, for rewinding, etc. In these instances, I/O-array must be present but is a dummy.)

status is an integer variable that can return the following status information:

Bit	Meaning
0	Error (bits 1, 3, 5, 6, 7, 8, 10 or 14 set)
1	Data late
2	Tape is rewinding
3	Illegal command
4	High density if = 1 (always 1 for cassettes)
5	Parity error
6	End of tape

Free Format Tape I/O (MTDIO) (Continued)

Bit	Meaning
7	End of file
8	Tape is at load point
9	9-track if =1, 7-track if =0 (always 1 for cassettes)
10	Bad tape or write failure
11	Send clock (always 0 for cassette)
12	First character (always 0 for cassette)
13	Write protected or write-locked
14	Odd character (always set to 0 for cassettes)
15	Unit ready

error is an integer variable that returns one of the FORTRAN error flags (page 1-3), which is a positive value.

record-count or word-count is an optional integer variable that returns the number of words written or read on a write or read or returns the number of records spaced over on space forward or backward.

An example of a call to MTDIO is:

```
DIMENSION IRRAY (1024)
     .
     .
     .
CALL MTDIO (5, ICOM, IRRAY, ISTAT, IER)
```

CHAPTER 4

TASKING

MULTITASKING CONCEPTS

A task is defined as a logically complete execution path through a user program that demands use of system resources such as peripheral devices for I/O, system or user overlays, or simply CPU control: task execution may occur independently and asynchronously with other tasks. A FORTRAN IV program run under RDOS can consist of any number of these tasks.

When a running program consists of more than one task, it is said to be a multitasking program. In such a multitask environment, tasks compete simultaneously for the use of system resources. Only one task may receive CPU control and the desired resource at any single moment. This allocation is awarded to tasks according to their priority and readiness to use the resources. A task scheduler governs the transfer of control to each task.

FORTRAN IV uses the multitask programming facilities available under RDOS, which allow execution of various routines to be performed asynchronously as separate tasks.

Task States

At a given time during execution of a multitask program, a task can be in one of four states: dormant, suspended, ready, or executing. A dormant task is one that has not been activated yet or has been terminated. A ready task is one that can proceed when given control of the processor. A suspended task is one that is not ready to proceed but is still alive. Tasks are said to be active if they exist in either the ready, suspended, or executing states.

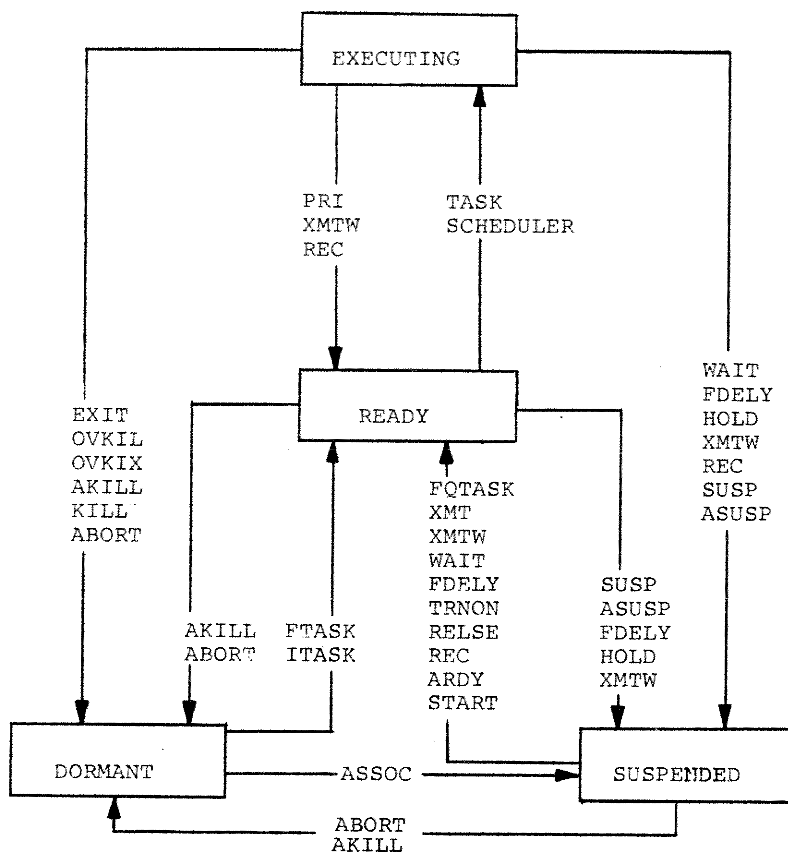
When a task is activated (FTASK or ITASK), it enters the ready state and competes with other ready tasks for control of the processor based on assigned priorities. When the task scheduler gives a ready task control of the processor, that task goes into the executing state and retains control until it has been completed or some event forces it to relinquish control. When a task cannot proceed until some event occurs, that task goes into the suspended state until that event occurs.

The diagram on the following page outlines the various task states pertaining to the calls which transfer a task from one state to another.

Task Control Blocks

A task control block (TCB) is a block of 13 or 14_8 locations used to store the status of an active task. Each active task, including the main program, has a single TCB maintained for it by the task scheduler. These TCB's are linked together to form the active chain. When a multitask program is loaded, a number of TCB's are created as specified in the CHANTASK statement (or in the RLDR command line format). During the execution of the program, those TCBs that are not being used are linked together to form the free chain.

When a task is activated, a free TCB is taken from the free chain and linked to the active chain. This TCB is then filled with status information for the newly activated task.



Task State Transitions

Task Priorities

When a task is activated it is assigned a priority number in the range 0 to 255 decimal. Tasks with the lowest numbers have the highest priorities. The task scheduler always gives control of the processor to the task in the ready state with the highest priority. The priority number assigned to a task can be changed while that task is executing a CALL PRI statement.

More than one task may be assigned the same priority number. The relative priority of ready tasks with a common priority number is determined by the relative positions of their respective task control blocks (TCBs) on the active chain maintained by the task scheduler. Each time a task relinquishes control to the scheduler, its TCB is moved to the end of the active chain. This gives ready tasks with a common priority number approximately equal opportunities to receive control of the processor.

Optionally, a task may have a task identification number (ID) which can be used in referencing that particular task. Task identification numbers have a default value of zero, or may be set with a value from 1 to 255. Only one task may be assigned to one identification number from 1 to 255, although many tasks may have the default identification number of zero.

Task Scheduler

During the execution of a multitask program, the task scheduler receives control of the processor when a task issues I/O or other system calls. The task scheduler searches the active chain for the TCB of the ready task with the highest priority number. This task is then given control of the processor to the task. If there are no tasks in the ready state, the task scheduler will wait until some event causes a task to be readied.

TASK EXECUTION CONTROL

Number of Tasks

In preparing a multi-task program for execution, the user must supply to RLDR a parameter specifying the maximum number of tasks which may be executing at any one time. This value may be supplied in one of three ways:

1. The first FORTRAN program unit contains the statement

   CHANTASK c,t

   where: c is the maximum number of channels.
   t is the maximum number of concurrently active tasks.

   When making channel and task specifications within a CHANTASK statement, the CHANTASK statement must precede all other statements in the main FORTRAN program, except a COMPILER DOUBLE PRECISION statement, COMPILER NOSTACK statement, or an OVERLAY statement.

2. The first assembly language program has

   .COMM TASK, t * 400 + c

3. The RLDR command line contains

   c/C (for channels)
   t/K (for tasks)

Task Control blocks (TCB's) will be allocated by RLDR in each case.

The first TCB is set up to cause activation of the main task at its starting address, with a priority of 0 (highest) and no task I.D. In FORTRAN IV this starting address is generally that of the runtime initializer .I, which in turn invokes the main FORTRAN program.

The two types of tasks under consideration here are "FORTRAN" and non-FORTRAN tasks. All program units written in FORTRAN require that they be executed as "FORTRAN" tasks. However, a "FORTRAN" task need not be written in FORTRAN and can, in fact, be written in assembly language.

A "FORTRAN" task is characterized by two features:

1. An extended save capability
2. FORTRAN stack facilities

The extended save capability causes the following page zero FORTRAN state variables to be saved and restored on task swaps:

.SVO	general return address temporary
.OVFL, NSP, .NDSP	number stack pointers
SP	runtime temporaries stack pointer
AFSE, QSP	procedural stack pointers

Whenever a task uses any of these variables, the task must be considered a FORTRAN task. The FORTRAN stack pointer FSP in location 16 is automatically saved, along with the accumulators and carry in the TCB.

These capabilities require memory space generally not required by self-contained non-FORTRAN ("Assembly") tasks. At program initiation memory must be parcelled out among FORTRAN tasks. The number of such tasks is denoted by the value of the symbol FRTSK which may be supplied by the user in an assembly language module such as

```
.ENT  FRTSK
FRTSK = f
.END
```

Memory will be divided equally f ways. If the user does not provide a definition of FRTSK, it will be assumed that all tasks are FORTRAN tasks and that memory will be divided t ways. F is less than or equal to t. f = t only if there are no non-FORTRAN tasks.

A non-FORTRAN task requires nothing to be saved on task swaps: no extended save capability is required; none of the FORTRAN state variables are modified, nor is any stack facility required.

Number of Tasks (Continued)

The number of tasks specified indicates the number of task control blocks that will be available at run time. If an attempt is made to activate a task when all available TCBs are in use, an error condition will result (see Activating a Task). Each task subprogram must begin with a TASK statement and end with an END line. The TASK statement has the following format:

TASK taskname

where: taskname is the name assigned to the task program unit. This name must be unique within its first five characters with respect to all function, subroutine, task, and overlay names.

A task name must be declared EXTERNAL in each external program unit that references it. Each task may be executed an arbitrary number of times during execution of a multitasking program.

Task Activation (FTASK, ITASK, ASSOC)

All tasks except the main FORTRAN program unit are activated by executing a call to either FTASK or ITASK. FTASK activates a task by task name; ITASK associates the identification number with the task name by which the task may later be referenced. A call to ASSOC associates a task name with an identification number and then puts the task in the suspended state; it can later be executed by calling START or TRNON. The call to FTASK has the format:

CALL FTASK (taskname, $error-return, priority-number [, IASM])

where: taskname is the name of the task to be activated. taskname is declared EXTERNAL in the calling task.

$error-return is a number of a statement in the calling program to which control is returned if the task cannot be activated (used when no TCB is available for the task).

priority number is an integer constant in the range 0 - 255 (decimal) specifying the priority assigned to the new task. (A priority of 0 indicates priority the same as the calling program.)

IASM is an optional parameter which must be in the argument list and set to nonzero if the task to be activated is written in other than FORTRAN IV (i.e., written in assembly language) and is not using the FORTRAN run time stack. Conversely, if IASM does not appear within the FTASK command line, taskname must be written in FORTRAN IV.

An example of a call to FTASK is:

```
      EXTERNAL PROG
      CALL FTASK (PROG, $14, 6)
   14 WRITE (10) "NOT ENOUGH TCBS"
```

A call to ITASK will, as well as activate a task, associate an identification number with the specified task. The format of the call to ITASK is:

CALL ITASK (taskname, identification, priority-number, error [, IASM])

where: taskname is the name of the task to be activated. taskname must be declared EXTERNAL in the calling task.

identification is the task identification number which is either an integer variable or an integer constant in the range 0 - 255; zero is the default value of the ID.

priority-number is an integer variable or constant in the 0 - 255 (decimal) range specifying the priority to be assigned to the newly activated task. A priority number of zero indicates that the task will have the same priority as the calling program.

error is an integer variable which will return one of the error codes upon completion of the call.

IASM is an optional parameter which must be in the argument list and set to non-zero if the task to be activated is written in other than FORTRAN IV (i.e., written in assembly language) and is not using the FORTRAN run time stack. Conversely, if IASM does not appear within the ITASK call, taskname, must be written in FORTRAN IV.

An example of a call to ITASK is:

```
.
.
.
EXTERNAL P1
CALL ITASK (P1, 10, 6, IER)
.
.
.
```

A call to ASSOC has the format:

CALL ASSOC (taskname, identification, priority-number, error [, IASM])

where: taskname is the name of the task to be put in the suspended state. taskname is declared EXTERNAL in the calling task.

identification is the task identification number which is either an integer variable or constant in the range 0 - 255 (decimal); zero is the default ID value.

priority-number is an integer variable or constant in the 0 - 255 (decimal) range specifying the priority to be assigned to the newly activated task. A priority number of zero indicates that the task will have the same priority as the calling task.

error is an integer variable which will return one of the error codes upon completion of the call.

IASM is an optional parameter which must be in the argument list and set to non-zero if the task to be activated is written in other than FORTRAN IV (i.e., in assembly language) and not using the FORTRAN run time stack. Conversely, if IASM does not appear within the ASSOC statement format, taskname must be written in FORTRAN IV.

The difference between a call to ITASK and a call to ASSOC is that both calls associate a task name with an identification number, but ITASK puts the task in the ready state while ASSOC puts the task in the suspended state. Routines activated by ASSOC may later be put into the ready state for execution by a call to START or a call to TRNON, both described on pages following.

Task Activation Based on Time of Day (FQTASK)

Tasks contained in overlays or resident in main memory can be executed periodically with an FQTASK call. If the task is contained in an overlay, it causes the overlay containing the task to be loaded so that execution of the task can proceed. Provision is also made to periodically execute core resident tasks. The call has the format:

CALL FQTASK (overlayname, task, array, error {, type})

where: overlayname is the name of the overlay containing the task subprogram to be executed. overlayname must be declared EXTERNAL in the calling task.

task is the name of the task subprogram (specified by TASK statement). task must be declared EXTERNAL in the calling task.

array is the name of an 11-element integer array that is unique for the task.

error is an integer variable which will return one of the error codes upon completion of the call.

type is an optional parameter which must be in the call if the task to be activated is written in other than FORTRAN IV (i.e., written in assembly language) or if the task is core resident. But, if type does not appear within the FQTASK call, task must have been written in FORTRAN IV and must be an overlay. In the case of non-overlay tasks, overlayname is a required dummy. type, when present, is an integer variable or constant specifying:

0	task is a FORTRAN overlay
-1	task is core-resident (non-overlay)
-2	task is non-FORTRAN overlay
-3	task is non-FORTRAN and core-resident non-overlay

Each task to be called by FQTASK must have a unique array. Before a call to FQTASK is executed, elements of array must have been assigned values as shown:

Element	Value
1	Used by the system
2	Number of times task is to be executed
3	Used by the system
4	Starting hour of the first task ($0 \leq hour \leq 23$)
5	Starting second within the hour of the first task execution ($0 \leq second \leq 3599$)
6	Task priority
7	Time (seconds) between successive task executions
8	Used by the system
9	Channel number on which the overlay file is opened (not used if task is core-resident)
10	Overlay conditional flag (0 = unconditional, 1 = conditional). Not used if task is core-resident.
11	Task ID. If no task ID is required, the element must be set to zero.

Task Activation Based on Time of Day (FQTASK) (Continued)

In addition to setting up array, the overlay file containing overlayname must have been opened via a call to OVOPN.

When a call to FQTASK is executed, overlayname is loaded at the time specified in array elements 4 and 5 and task is first executed. The task is executed periodically after each increment specified by element 7 until the task has been executed the number of times specified by array element 2.

While it is not necessary for FORTRAN resident tasks queued in by FQTASK to be terminated by a call to KILL or for overlay tasks to be terminated by OVKIL if they are queued in by a call to FQTASK, it is recommended that the user provide the terminating KILL or OVKIL calls respectively.

Assembly language resident tasks and overlays are handled in a slightly different manner, but they also need not be terminated by a call to KILL or OVKIL respectively if queued in by a call to FQTASK. Assembly language resident tasks do not need a call to KILL if the address stored in AC3 is saved on entry and jumped to on exit. Assembly language overlay tasks queued in do not need a call to OVKIL if the address stored in AC3 is saved on entry and jumped to on exit with the overlay number stored in AC0.

If the necessary overlay area for overlayname is not available or if there is no TCB available for the task, task execution is postponed until the resource is available. Examples of calls to FQTASK are:

CALL FQTASK (OV, TASK1, IAR, IER)

CALL FQTASK (DUM, TASK1, IAR, IER, -1)

Start a Task After a Time Delay (START)

A call to the routine START will cause a task, which has been activated and put into the suspended state by a call to ASSOC, to be put into the ready state for execution after expiration of a specified time delay. The format of the call is:

CALL START (id, time, unit, error)

where: id is the identification number of the task which is to be delayed then executed at the expiration of the delay.

time is an integer variable, constant or array element specifying the length of time (in units specified by unit) of the delay before execution of the task. (If time equals zero, id will be executed as soon as permissible.)

unit is an integer variable, constant or array element specifying the units of time as follows:

0	pulses of the real time clock
1	milliseconds
2	seconds
3	minutes

error is an integer variable which will be set equal to one of the error codes upon completion of the call.

Start a Task after a Time Delay (START) (Continued)

An example of a call to START is:

CALL START (26, 30, 3, IER)

Execution of the task with 26 as its identification number will commence in 30 minutes.

Execute a Task at a Specified Time (TRNON)

A call to the TRNON routine will ready a task that was activated and suspended by a call to ASSOC for execution at a specified time of day. The format of the call is:

CALL TRNON (id, array, error)

where: id is an integer variable, constant, or array element specifying the identification number of the task to be executed at a specified time.

array is an integer 3-element array specifying:

first element	-	hours
second element	-	minutes
third element	-	seconds

error is an integer variable which will return one of the error conditions upon completion of the call.

An example of a call to TRNON is:

CALL TRNON (32, IAR, IER)

Task Suspension (SUSP, ASUSP, HOLD, WAIT, FDELY)

The following may cause suspension of an executing task:

1. A CALL SUSP is executed.
2. A CALL HOLD is executed.
3. The task must wait for some I/O event.
4. A CALL FDELY is executed.
5. A CALL ASUSP is executed to suspend all tasks of the same priority as the executing task.
6. A CALL REC is executed to receive a message not yet sent.
7. A CALL XMTW is executed to transmit a message for which a corresponding CALL REC has not yet been received.
8. A CALL WAIT is executed.

Note that a task may be doubly suspended, e.g., by a call to ASUSP and I/O completion. In this case, two separate suspend bits are actually set and both must be reset before the task will be readied.

Execution of a call to SUSP causes the task in which it is executed to be suspended. The format of the call is:

CALL SUSP

Task Suspension (SUSP, ASUSP, HOLD, WAIT, FDELY) (Continued)

Execution of a call to ASUSP causes all tasks of a given priority (ready and executing) to be suspended. The format of the call is:

CALL ASUSP (priority-number)

where: priority number is a decimal integer (0-255) giving the priority number of the tasks to be suspended. A priority number of 0 indicates a priority equal to that of the caller's.

Execution of a call to HOLD causes the task having the identification number given in the call to be suspended. The format of the call is:

CALL HOLD (identification, error)

where: identification is an integer variable, constant, or array element specifying the identification number of the task.

error is an integer variable which will return one of the error codes upon completion of the call.

A call to the WAIT routine allows the executing task to voluntarily relinquish control of the system for a specified period of time. This enables lower-priority tasks to be executed for the duration of the delay. When execution is resumed, system resources will be as they were before the delay. The format of the call is:

CALL WAIT (time, units, error)

where: time is an integer variable, constant, or array element specifying the length of time (in terms of units) delay to elapse before execution is resumed.

units is an integer variable, constant, or array element specifying the unit of time to be used as follows:

0	pulses of the real time clock
1	milliseconds
2	seconds
3	minutes

error is an integer variable which will return one of the error codes upon completion of the call.

Execution of a call to FDELY will suspend that task for a specified amount of time. The format of the call is:

CALL FDELY (number-of-pulses)

where: number-of-pulses is a decimal integer, giving the number of real time clock pulses for which the task will be suspended.

Readying a Task (ARDY, RELSE)

When a task is activated, it is put into the ready state, and while active, remains in either the ready state, the executing state, or the suspended state. A suspended task can be readied under the following circumstances:

Readying a Task (ARDY, RELSE) (Continued)

1. A task suspended by execution of a call to SUSP, HOLD, or ASUSP may be readied by execution of a call to ARDY.

2. A task suspended for performance of I/O is readied automatically when I/O is completed.

3. A task suspended by execution of a call to FDELY is readied at the end of the time period specified.

4. A task suspended by execution of a call to REC is readied by the execution of a corresponding call to XMT or XMTW.

5. A task suspended by execution of a call to XMTW is readied by the execution of a corresponding call to REC.

Note that a doubly suspended call must be doubly readied. Execution of a call to ARDY causes all tasks of the priority specified in the call to be readied if they were previously suspended by a SUSP, ASUSP, or HOLD call. No other tasks are affected. The format of the call is:

CALL ARDY (priority-number)

where: priority-number is the priority number of the tasks to be readied.

Execution of a call to RELSE causes the task having the identification number given in the call to be readied if it was suspended by a SUSP, ASUSP, or HOLD call. The format of the call is:

CALL RELSE (id, error)

where: id is the identification number assigned to the task in an ITASK call.

error is an integer variable which will return one of the error codes upon completion of the call.

Task Priority Modification (PRI, CHNGE)

When a task is activated it is assigned a priority number. A call to PRI makes it possible to change the priority number of the task. The call has the format:

CALL PRI (priority-number)

where: priority-number gives the new priority of the task.

Execution of a call to PRI causes the priority number of the executing task to be changed. A task may change its priority any number of times while it is active. An example of a call to PRI is:

CALL PRI (37)

Execution of a call to CHNGE causes the priority number of the task having the identification number given in the call to be changed. The format of the call is:

CALL CHNGE (id, priority-number, error)

Task Priority Modification (PRI, CHNGE) (Continued)

where: id is the identification number of the task.

priority-number gives the new priority of the task.

error is an integer variable which will return one of the error codes upon completion of the call.

Task Termination (KILL, AKILL, ABORT, EXIT)

A task may be terminated (placed in the dormant state) by execution of a call to KILL, AKILL, ABORT, or EXIT. Execution of a call to KILL kills the executing task. The format of the call is

```
CALL KILL
```

Execution of a call to AKILL immediately terminates all ready or executing tasks of the priority number given in the call. Any suspended tasks having that priority number are killed immediately, unless they are awaiting an I/O event, in which case they are killed immediately after they are readied. The format of the AKILL call is:

```
CALL AKILL (priority-number)
```

Execution of a call to ABORT terminates the task having the identification number given in the call. The format of the call is:

```
CALL ABORT (id, error)
```

where: id is the identification number previously assigned in an ITASK call.

error is an integer variable which will return one of the error codes upon completion of the call.

Execution of a call to EXIT causes the executing task to be terminated and causes a return to the CLI. The format of the call is:

```
CALL EXIT
```

Obtaining Task Status (STTSK)

The user can obtain the current status of a given task (ready, suspended, or inactive) by a call to the STTSK routine. The format of the call is:

```
CALL STTSK (id, status, error)
```

where: id is the identification number of the task, assigned in a call to ITASK.

status is an integer variable for which a status code is returned.

error is an integer variable which will return one of the error codes upon completion of the call.

Obtaining Task Status (STTSK) (Continued)

The possible status codes that may be returned are:

0	Ready
1	Suspended by a .SYSTM call
2	Suspended by ASUSP, SUSP, HOLD
3	Wait due to XMTW or REC
4	Wait for overlay node
5	Suspended by ASUSP, SUSP, or HOLD and by a .SYSTM call
6	Suspended by XMT/REC and by SUSP, ASUSP, or HOLD
7	Wait for overlay node and suspended by ASUSP, SUSP, or HOLD
8	No tasks exist for this identification number.

INTERTASK COMMUNICATION (XMT, REC, XMTW)

Active tasks may communicate with each other through shared COMMON (labeled or blank). Information generated by one executing task can be retained in data or subprogram units until one or more other tasks are executing and can access this information. No synchronization of creation and use of information is implicit in this scheme. Unless precautions are taken, attempts may be made by tasks to use information not yet generated.

Synchronized transmission of one word messages between active tasks can be accomplished using three calls: CALL XMT, CALL XMTW, CALL REC. The format of the call to XMT is:

CALL XMT (message-key, message-source, $error-return)

where: message-key is an integer variable common to both the transmitting and receiving tasks.

message-source is an integer variable in the transmitting task containing the non-zero message to be transmitted.

error-return is the number of a FORTRAN statement (in the program unit containing the CALL XMT statement) to which control is returned if the message-key is non-zero when the CALL XMT is executed.

A one-word non-zero integer message can be transmitted by setting message-source equal to that value and then executing the CALL XMT. A message transmitted to XMT is received by execution of a call to the REC routine. The call has the format:

CALL REC (message-key, message-destination)

where: message-key is an integer variable common to both the transmitting and receiving tasks.

message destination is an integer variable accessible by the receiving task.

In the transmission of a message using corresponding CALL XMT and CALL REC statements, the order in which these statements are executed is unimportant. If CALL XMT is executed first, the value of message-source is assigned the variable message-key. Message-key should have the value of 0 when CALL XMT is executed: if it does not, a return is made to statement error-return and the value of message-key is left unchanged. When CALL REC is subsequently executed, message-destination is assigned the value of message-key and message -key is assigned the value of 0.

INTERTASK COMMUNICATION (XMT, REC, XMTW) (Continued)

If CALL REC is executed before the corresponding CALL XMT, the receiving task is suspended until CALL XMT is executed. When CALL XMT is executed, message-destination is assigned the value of variable message-source and the receiving task is is placed in the ready state.

A call to XMTW routine is used in place of CALL XMT when it is desired that the transmitting task be suspended until the receiving task receives the message. The call XMTW has the same format as CALL XMT, with XMTW merely replacing XMT. The transmitting task is suspended only if CALL XMTW is executed before the corresponding CALL REC.

An example of the intertask communication calls is:

```
          TASK SEG1
          COMMON KEY
                .
                .
                .
          CALL REC (KEY, MDEST)
                .
                .
                .
          END


          TASK SEG2
          COMMON KEY
                .
                .
                .
          CALL XMT (KEY, MSRCE, $17)
                .
                .
                .
  17      WRITE (10) "KEY ALREADY SET"
                .
                .
                .
          END
```

TASK OPERATOR COMMUNICATION MODULE

A small task Operator Communications Module (OPCOM) is available which permits certain specific operator commands to be executed immediately when entered from the operator console, TTI. OPCOM provides the capability to sample or change the status of tasks, and to run these tasks or queue them for periodic execution. It should be noted that OPCOM is unrelated to the CLI and remains an integral part of each save file where it is used.

A list of the OPCOM commands is shown below along with their function. For a complete description of these commands, see Chapter 5 of the RDOS User's Manual.

To kill a task:

<CTRL E> $\begin{Bmatrix} B \\ F \end{Bmatrix}$ *,KIL,task I.D.)

To change the priority of a task:

<CTRL E> $\begin{Bmatrix} B \\ F \end{Bmatrix}$ *,PRI,task I.D., new priority)

To queue a task for periodic execution:

<CTRL E> $\begin{Bmatrix} B \\ F \end{Bmatrix}$ *,QUE,program #, hour, minute, second, repeats, interval, {priority})

To ready a task:

<CTRL E> $\begin{Bmatrix} B \\ F \end{Bmatrix}$ *,RDY,task I.D.)

To initiate a task for execution:

<CTRL E> $\begin{Bmatrix} B \\ F \end{Bmatrix}$ *,RUN,program #, {priority})

To suspend a task:

<CTRL E> $\begin{Bmatrix} B \\ F \end{Bmatrix}$ *,SUS,task I.D.)

To display the status of a task:

<CTRL E> $\begin{Bmatrix} B \\ F \end{Bmatrix}$ *,TST,task I.D.)

Two FORTRAN task calls IOPC and IOPROG, when included in a program, will prepare the OPCOM package for entry of the OPCOM commands at the operator console. CALL IOPC, a mandatory call, will initialize the OPCOM package, making it accessible to the operator. CALL IOPROG, an optional call, will build a program table of task information for reference by the OPCOM commands RUN and QUE and need only be used if these commands will be entered at the console. IOPROG must be called for each task being described.

Initializing the Task Operator Communication Module (IOPC)

CALL IOPC ([program array, number of programs, queue array, number of queues,
overlay channel,] error)

where: program array is an empty array of at least 8 times the number of tasks to be defined. Program array must be declared as process global (common or static).

number of programs is the total number of tasks to be described.

Initializing the Task Operator Communication Module (IOPC) (Continued)

queue array is an empty array of at least 13 times the number of programs in size. Queue array must be declared process global.

number of queues is the total number of queues required: one for each concurrent RUN or QUE OPCOM command.

overlay channel is opened by a call to .OVOPN.

NOTE:

1. One TCB must be reserved for the OPCOM package.
2. When running without a program table (see IOPROG) set the first five arguments to zero.
3. IOPC may be called more than once. Additional calls will remove the previous program array. A new array must be given for each call.

Building a Program Table (IOPROG)

CALL IOPROG (program name, program number, task identifier, task priority, [overlay node/number, conditional load], error, [ASM])

where:

program name is the task name.

program number is the number associated with the program used in RUN and QUE commands.

task identifier is an integer from 0 to 255.

task priority is an integer from 0 to 255.

conditional load is 0 meaning conditional, or -1 meaning unconditional.

error will be 1, indicating successful completion or RDOS error code incremented by 3 indicating an error.

ASM is as follows:

If not included in the calling sequence, the task described is a FORTRAN task and in an overlay.

If included in calling sequence:

0	=	task described is a FORTRAN task and in an overlay.
-1	=	task described is FORTRAN task and core resident.
-2	=	task described is non-FORTRAN task and in an overlay.
-3	=	task described is non-FORTRAN and core resident.

NOTE: The only error message return on this call will be ER MEM indicating insufficient memory to include program description.

SAMPLE TASKING PROGRAM

The program following is an illustration of a FORTRAN IV program written for a multitask environment. The main program contains calls that activate two tasks, TIMPLT and QUAD, at priority levels 1 and 2 respectively. The main program then deactivates itself by issuing a call to KILL.

QUAD outputs to the teletypewriter solutions to quadratic equations from input values provided by the programmer. TIMPLT prints a counter on the line printer, one count per line, 55 lines per page. The counter is incremented once each second, given a real time clock cycle that is set to 100 milliseconds.

```
;
;C        TEST 3

;          EXTERNAL              TIMPLT,QUAD
; C******
;          IPU=12
; C******
;          WRITE (IPU,376)
; 376      FORMAT (1H1,31X,24HFORTRAN TEST PROGRAM -- ///)
;          WRITE (IPU,378)
; 378      FORMAT (17H0BEGIN TEST 3        )
;          WRITE(IPU)  '  START OF MAIN'
;          CALL ITASK (QUAD,10,110,IER)
;          CALL ITASK (TIMPLT,10,111,IER)
;        IF (IER.NE.1) GO TO 100
;          WRITE(IPU) '  EXIT FROM MAIN'
;          WRITE (IPU,377)
; 377      FORMAT (17H0END OF TEST 3      )
;          CALL KILL
; 100      WRITE (IPU)  '   ERROR!'
;          END
```

SAMPLE TASKING PROGRAM (Continued)

```
!
! C
! C       MTEST1A USED IN TEST3
! C
!         TASK QUAD
! C       GET QUADRATIC EQUATION COEFFICIENTS
!         A=.1Ø
!         B=2.Ø
!         C=3Ø
! 1ØØ     A=A+1
!         B=B-A
!         C=C+(B-A)
! C       F(X) = A*X**2+B*X+C
!         IF((B**2-4*A*C).LT.Ø)GOTO 1Ø
! C       FIND THE REAL ROOTS
!         X1R = (-B+(B**2-4*A*C)**.5)/(2*A)
!         X2R = (-B-(B**2-4*A*C)**.5)/(2*A)
! C       OUTPUT THE COEFFICIENTS AND THE REAL ROOTS
!         WRITE(12,1)A,B,C,X1R,X2R
! 1       FORMAT(1HØ,"A = ",F1Ø.4,"B = ",F1Ø.4,"C = ",
!        1         F1Ø.4,"X1 = ",F1Ø.4,"X2 = ",F1Ø.4)
!         CALL FDELY(5Ø)
!         GOTO 1ØØ
! 1Ø      WRITE(12,2)A,B,C
! 2       FORMAT(1HØ,"*** COMPLEX ROOTS***",
!        1         "A = ",F1Ø.4,"B = ",F1Ø.4,"C = ",F1Ø.4)
!         GOTO 1ØØ
!         END
```

```
!
! C
! C       MTEST1B USED IN TEST3
! C
!         TASK TIMPLT
! C       SET OUTPUT COUNTER TO ZERO
!         J=Ø
!         N = Ø
! 1       LINES = Ø
! C       RESET LINE COUNTER TO ZERO
! 2       LINES = LINES+1
!         N = N+1
!         CALL FDELY(1Ø)
! C       IF BOTTOM OF PAGE, GOTO TOP OF NEXT PAGE
!         IF(LINES.EQ.55)GO TO 1Ø
!         WRITE(12)N
!         GOTO 2
! 1Ø      WRITE(12)N
!         WRITE(12,2Ø)
! 2Ø      FORMAT(1H1)
!         J=J+1
!         IF(J.EQ. 2) CALL AKILL(1Ø)
!         GOTO 1
!         END
```

```
                              FORTRAN TEST PROGRAM --

BEGIN TEST 3
  START OF MAIN
  EXIT FROM MAIN

END OF TEST 3

*** COMPLEX ROOTS***A =       1.1000B =      0.9000C =     29.8000

*** COMPLEX ROOTS***A =       2.1000B =     -1.2000C =     26.5000

*** COMPLEX ROOTS***A =       3.1000B =     -4.3000C =     19.1000

*** COMPLEX ROOTS***A =       4.1000B =     -8.4000C =      6.6000

A =       5.1000B =    -13.5000C =    -12.0000X1 =      3.3495X2 =     -0.7025
        1
        2
        3
        4
        5

        6
A =       6.1000B =    -19.6000C =    -37.7000X1 =      4.5665X2 =     -1.3534
        7
        8
        9
       10

       11
A =       7.1000B =    -26.7000C =    -71.5000X1 =      5.5689X2 =     -1.8083
       12
       13
       14
       15

       16
A =       8.1000B =    -34.8000C =   -114.4001X1 =      6.4769X2 =     -2.1806
       17
       18
       19
       20

       21
A =       9.1000B =    -43.9000C =   -167.4001X1 =      7.3328X2 =     -2.5087
       22
       23
       24
       25

       26
A =      10.1000B =    -54.0000C =   -231.5001X1 =      8.1566X2 =     -2.8101
       27
       28
       29
       30

       31
A =      11.1000B =    -65.1000C =   -307.7002X1 =      8.9590X2 =     -3.0942
```

SAMPLE TASKING PROGRAM (Continued)

```
          32
          33
          34
          35

          36
A =        12.1ØØØB =    -77.2ØØØC =   -397.ØØØ2X1 =      9.7465X2 =     -3.3663
          37
          38
          39
          40

          41
A =        13.1ØØØB =    -9Ø.3ØØØC =   -5ØØ.4ØØ4X1 =     1Ø.5231X2 =     -3.63ØØ
          42
          43
          44
          45

          46
A =        14.1ØØØB =   -1Ø4.4ØØØC =   -618.9ØØ4X1 =     11.2916X2 =     -3.8873
          47
          48
          49
          5Ø

          51
A =        15.1ØØØB =   -119.5ØØ1C =   -753.5ØØ5X1 =     12.Ø538X2 =     -4.1398
          52
          53
          54
          55



          56
A =        16.1ØØØB =   -135.6ØØ1C =   -9Ø5.2ØØ7X1 =     12.811ØX2 =     -4.3887
          57
          58
          59
          6Ø

          61
A =        17.1ØØØB =   -152.7ØØ1C =  -1Ø75.ØØ1ØX1 =     13.5644X2 =     -4.6346
          62
          63
          64
          65

          66
A =        18.1ØØØB =   -17Ø.8ØØ1C =  -1263.ØØ1ØX1 =     14.3146X2 =     -4.8781
          67
          68
          69
          7Ø

          71
```

SAMPLE TASKING PROGRAM (Continued)

```
A =    19.1ØØØB =  -189.ØØØ1C = -1472.ØØ1ØX1 =     15.Ø622X2 =     -5.1198
      72
      73
      74
      75

      76
A =    2Ø.1ØØØB =  -21Ø.ØØØ1C = -17Ø3.ØØ1ØX1 =     15.8Ø76X2 =     -5.3598
      77
      78
      79
      8Ø

      81
A =    21.1ØØØB =  -231.1ØØ1C = -1955.2Ø1ØX1 =     16.5512X2 =     -5.5986
      82
      83
      84
      85

      86
A =    22.1ØØØB =  -253.2ØØ1C = -223Ø.5Ø1ØX1 =     17.2933X2 =     -5.8362
      87
      88
      89
      9Ø

      91
A =    23.1ØØØB =  -276.3ØØØC = -2529.9Ø1ØX1 =     18.Ø34ØX2 =     -6.Ø729
      92
      93
      94
      95

      96
A =    24.1ØØØB =  -3ØØ.4ØØ1C = -2854.4Ø2ØX1 =     18.7736X2 =     -6.3Ø89
      97
      98
      99


     1ØØ

     1Ø1
A =    25.1ØØØB =  -325.5ØØ2C = -32Ø5.ØØ2ØX1 =     19.5122X2 =     -6.5441
     1Ø2
     1Ø3
     1Ø4
     1Ø5

     1Ø6
A =    26.1ØØØB =  -351.6ØØ3C = -3582.7Ø2ØX1 =     2Ø.25ØØX2 =     -6.7787
     1Ø7
     1Ø8
     1Ø9
     11Ø
```

CHAPTER 5

SWAPPING, CHAINING, AND OVERLAYS

Program Swapping and Chaining

During run time, programs may be swapped or chained. In chaining, the currently executing program (the caller) issues a call to either FCHAN or CHAIN which causes the program to be overwritten in core by another program loaded from disk. The core image of the calling program is not saved. In program swapping, the currently executing program issues a call to FSWAP or SWAP which causes the current program's core image to be temporarily saved on disk and a new program to be loaded from disk for execution. The saved program can later be restored to core by a call to BACK, FBACK, or EBACK and continue its execution from the point of suspension.

The diagram on page 5-2 illustrates the results of the various program segmentation calls and statements concerning swapping and chaining.

When performing a program swap, the calling program is said to execute at a level higher than the called program. (The higher the level of execution of a program, the lower its associated level number is. The CLI is always at level number 0, an assembler or the FORTRAN IV compiler is usually at level number 1, etc.) When a program issues a call to FSWAP, the execution level number is incremented, the calling program is saved on disk, and the called program is brought into core for execution. When a call to FBACK, BACK, or EBACK is encountered, the execution level number is decremented and the calling program is restored to core. If an attempt is made to nest swaps to a level deeper than four, an RDOS error will result.

Program swapping allows core images of programs to be saved and called for execution more than once during a program's execution. Each program swapped to must contain a complete FORTRAN IV program consisting of a main program unit and all subroutines directly or indirectly linked to it.

When performing a program chain, the called program will replace the calling program at the same execution level. The calling program is not saved but is entirely overwritten by the called program. There is no limit on the number of chains performed. Program chaining can be used to subdivide an exceedingly large program that would exceed the limits of core if it were to reside in core in its entirety. Each chained-to file must contain a complete FORTRAN IV program consisting of one main program unit and all subroutines directly or indirectly linked to it.

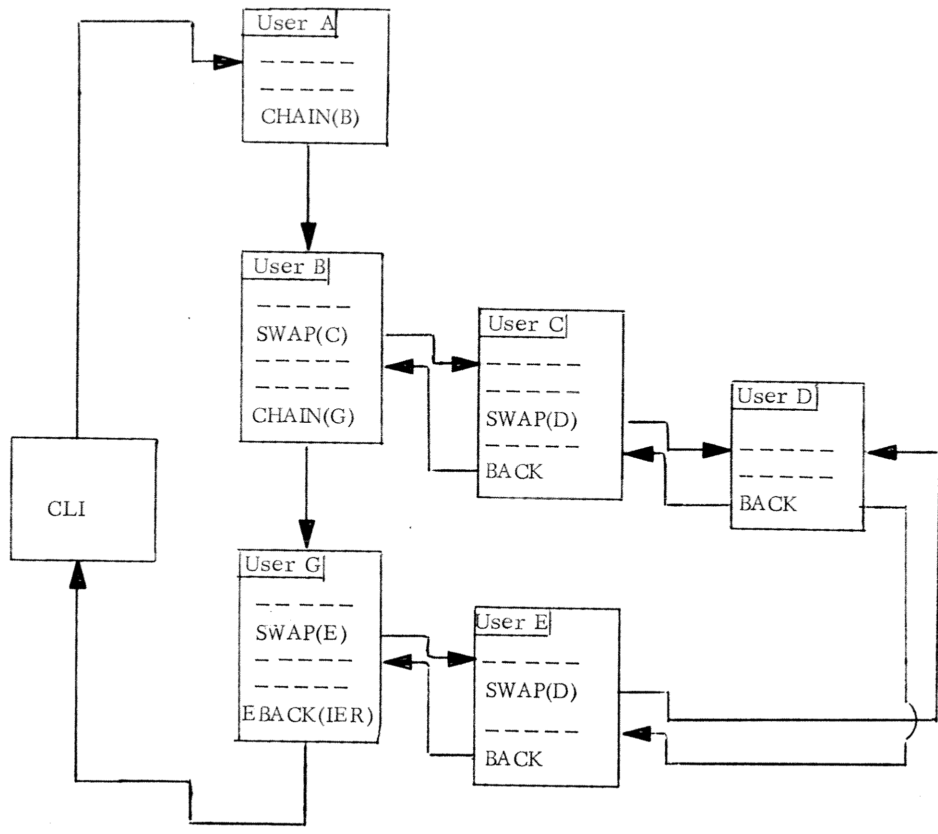
This chapter is divided into two sections, the first dealing with swapping and chaining. The second deals with overlays, defining what an overlay is, and how it is created, deleted, loaded, etc.

Program Swapping (SWAP, FSWAP)

An executing program can cause its core image to be temporarily saved on disk and another program to be loaded from disk for execution. This is accomplished by issuing a call to either the SWAP routine or to the FSWAP routine; the difference between the two calls being that SWAP contains an error location. The format of the two calls is:

CALL SWAP (filename, error)

CALL FSWAP (filename)



Calls and Statements	Change of Level
SWAP, FSWAP	Level n → level n+1
CHAIN, FCHAN	Level n → level n
STOP, EXIT, EBACK (chaining)	Level n → CLI
BACK, FBACK, EBACK (swapping)	Level n → level n-1

LEVELS OF SWAPPING AND CHAINING

Program Swapping (SWAP, FSWAP) (Continued)

where: filename is the name of the save file to be executed next.

error is an integer variable which will return one of the error codes upon completion of the call.

The calling program is suspended and its current status is saved in the current TCB. If the execution level of the calling program is n, filename executes at level n + 1 . An example of a call to SWAP and a call to FSWAP is:

```
CALL SWAP ("ABC", IER)

CALL FSWAP ("A2")
```

Restoring a Swapped Program (BACK, FBACK, EBACK)

An executing program can cause the last program to be swapped out to disk to be brought back into core for a resumption of execution. The executing program will, at that time, be swapped out to disk until called for again. Calls to the run time routines BACK, FBACK, and EBACK will perform this restoration of the last swapped program to disk. The format of the call to BACK (which brings back programs swapped by SWAP or FSWAP) is:

```
CALL BACK
```

The format of the call to FBACK (which brings back programs swapped by either SWAP or FSWAP) is:

```
CALL FBACK
```

The call to EBACK can be made from either a chained-to or swapped-to program and restores the program that is at the next higher level with a standard error return. The restored program is either the last program swapped out or in the case of chaining the next higher level program, e.g., the CLI. The format of the call to EBACK is:

```
CALL EBACK (error)
```

where: error is an integer variable which will return one of the error codes upon completion of the call.

An example of a calling sequence is:

```
CALL SWAP ("A2", IER)
     .                  A1, executing at level 1, swaps in A2
     .                  at level 2.
     .
CALL SWAP ("A3", IER)
     .                  A2, executing at level 2, swaps in A3
     .                  at level 3.
     .
CALL BACK
     .                  A3 at level 3 swaps to disk and brings
     .                  back A2 at level 2.
     .
CALL SWAP ("A4", IER)
     .                  A2 at level 2 swaps in A4 at level 3.
     .
     .
CALL EBACK (IER)
     .                  A4 at level 3 swaps to disk and brings
     .                  back A2 at level 2.
     .
```

Program Chaining (CHAIN, FCHAN)

The currently executing program can cause its core image to be overwritten by another program on disk when the user issues a call to either the CHAIN routine or the FCHAN routine. The formats of the CHAIN and FCHAN calls are:

CALL CHAIN ("filename", error)

CALL FCHAN ("filename")

where: filename is the name of the save file to be executed next. The execution level is the same as that of the caller's.

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to CHAIN and a call to FCHAN is:

CALL CHAIN ("AA", IER)

CALL FCHAN ("ABC")

Returning to Level Zero

The FORTRAN IV statement STOP, or call to the run time routine EXIT, will each cause the termination of a task or program and return to level zero, the CLI.

The format of the STOP statement is:

STOP [message]

where: message is an optional message which can be printed upon termination of the executing task or program.

The call to EXIT has the format:

CALL EXIT

OVERLAYS

Overlays may be used when core is not large enough to accommodate an entire user program. During loading of relocatable binaries, two files are created rather than a single save file that would have to be brought into core in its entirety for execution. One file is the save file which contains the root program to be brought into core. The other is an overlay file that will remain on disk. When an overlay is referenced either from the root program or from another overlay that was previously brought into core, the overlay will be brought into core.

The save file contains, in addition to the root program, a directory of the overlay file and a series of overlay areas. Each overlay area in the save file corresponds to an overlay segment in the overlay file. Each overlay area in the save file represents an area of core that will accommodate a single overlay. Each overlay segment in the overlay file may contain up to 256 (decimal) overlays. Each overlay area in the save file is large enough to accommodate the largest overlay in that overlay segment. Only one overlay of an overlay segment may reside in core at a given time. On the save file, up to 128 (decimal) overlay areas may be allocated. They are designated 0 through 127.

When loading relocatable binaries, those binaries that make up an overlay area are enclosed in square brackets in the RLDR command line. Separate overlays of the overlay area are

OVERLAYS (Continued)

indicated by commas. The format of the RLDR command when overlays are included is:

$$\text{RLDR } \underline{\text{rootname}}_0 \left\{\begin{matrix}\underline{\text{rootname}}_1 \\ [\underline{\text{overlay-area}}_0]\end{matrix}\right\} \cdots \left\{\begin{matrix}\underline{\text{rootname}}_n \\ [\underline{\text{overlay-area}}_{n-1}]\end{matrix}\right\} \underline{\text{libraries}}$$

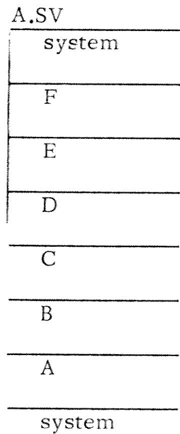
where: $\underline{\text{rootname}}_0$ is the name of the main FORTRAN program in relocatable binary.

other rootnames are names of relocatable binaries to become part of the save file program.

each overlay-area contains the names of relocatable binaries that are overlays or part of of a single overlay.

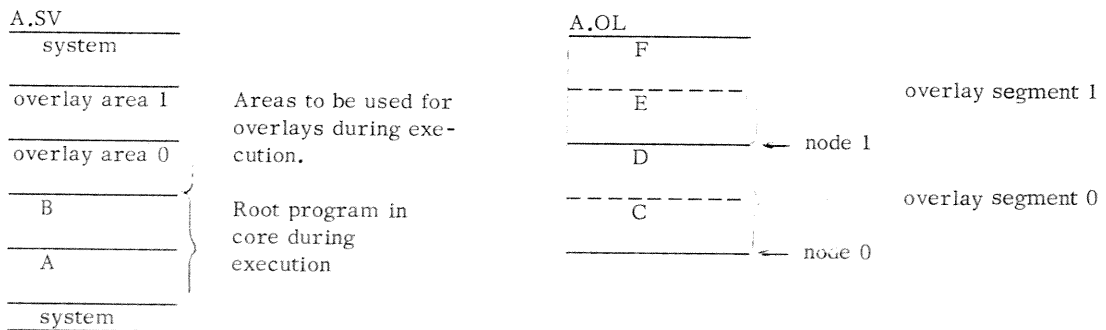
To see how save and overlay files are created, compare the following examples of RLDR commands:

RLDR A B C D E F libraries)



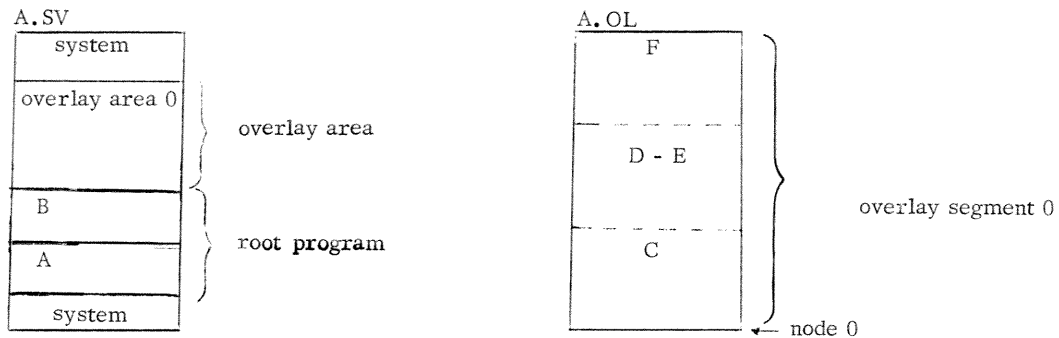
A, B, C, D, E, and F must all be in core during execution.

RLDR A B [C, D] [E, F] libraries)



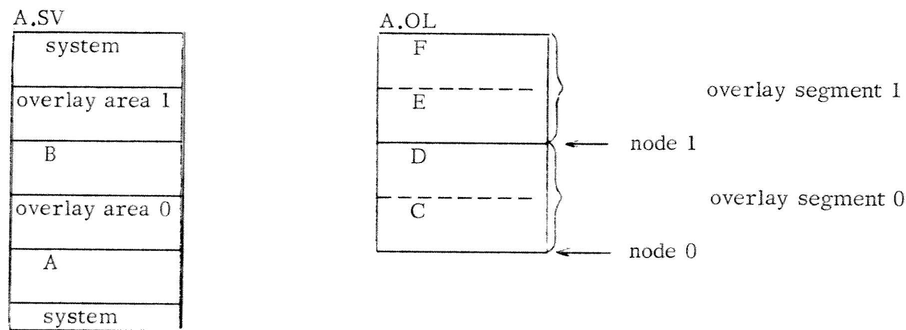
OVERLAYS (Continued)

RLDR A B [C, D E, F] libraries)



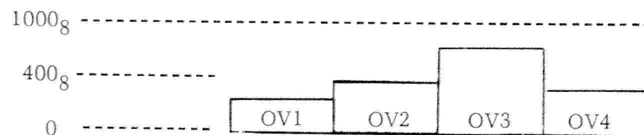
Note in the previous example that two or more relocatable binaries may be loaded as a single overlay within an overlay area. In this case, D and E are loaded as a single overlay, since there is no comma between the relocatable binaries in the command line.

RLDR A [C, D] B [E, F] libraries)

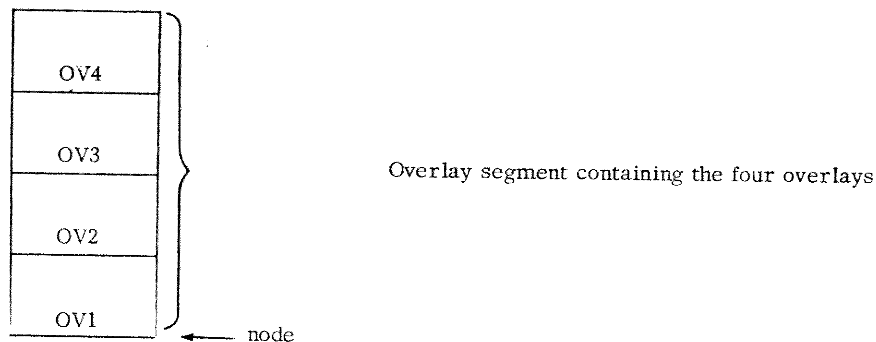


Note in the example above that tasks and overlay areas may be interspersed after the main FORTRAN program is loaded.

Within each overlay segment in the overlay file, each overlay occupies an equal area. The area is a multiple of 400 octal locations and is large enough to accommodate the largest overlay of the overlay segment. For instance, if there are four overlays, OV1, OV2, OV3, and OV4 in an overlay segment:



then each overlay will be allotted 1000 octal locations to accommodate the largest overlay:



The overlay file is created as a contiguous file. This allows the operating system to use multiple block reads (moving head disk)* for faster loading of overlays. As a result, each overlay within an overlay area is the same size. This is not a restriction on the user, however, as the relocatable loader will automatically adjust each overlay to be equal in size to the multiple of octal 400 that will accommodate the largest overlay within the overlay area. For better disk space utilization, though, the user should put overlays of approximately the same size within the same overlay area.

Overlays maintained in the overlay file are never altered during the execution of a program. Each time an overlay is loaded into core in the overlay area, it is in its original form whether or not it contains a non-reentrant routine. No part of an overwritten overlay is ever saved.

Once an overlay file has been loaded and resides on disk, it can be altered only by being reloaded using RLDR or, if desired, one or more overlays can be changed using the overlay loader, OVLDR, described in a later section.

Numbering of Overlays within an Overlay File

Overlays are numbered octally within an overlay file. There may be up to 128 decimal overlay segments within an overlay file (numbered 0 - 177_8). In a single-task environment there may be up to 256 decimal overlays within each overlay segment of the overlay file (numbered 0 - 377_8); in a multitask environment there may be up to 128 decimal overlays within each overlay segment of the overlay file (numbered 0 - 177_8). The overlay is referenced by a word that identifies the node (overlay area) and the overlay within the area. Thus overlay 1 of area 0 is numbered 1 while overlay 1 of area 2 is numbered 1001_8. The chart on the following page illustrates the numbering scheme in referencing a particular overlay within an overlay file in a single-task environment

In FORTRAN IV, each overlay of each segment is given a unique name in an OVERLAY statement (see page 5 - 8), and is referenced in calls by that name, so that it is not necessary to reference an overlay by number.

*While use of contiguous files enables faster loading from moving head disks, this does not imply that overlays are only used when the system configuration includes a moving head disk.

Numbering of Overlays Within an Overlay File (Continued)

Segment Number	Overlays Within This Segment (single-task environment)								
0	0	1	2	3	4	...	375	376	377
1	400	401	402	403	404	...	775	776	777
2	1000	1001	1002	1003	1004	...	1375	1376	1377
3	1400	1401	1402	1403	1404	...	1775	1776	1777
4	2000	2001	2002	2003	2004	...	2375	2376	2377
.									
.									
.									
126	77000	77001	77002	77003	77004	...	77375	77376	77377
127	77400	77401	77402	77403	77404	...	77775	77776	77777

Overlays in Single or Multiple Task Environments

Overlays may exist in either single or multiple task environments. In either environment, the overlay must be assigned a name in an OVERLAY statement, the overlay file must be opened by a call to OVOPN before an overlay file can be loaded into core, and the opened file is closed by a call to CLOSE.

However, in a multiple task environment, overlays and overlay areas can be shared by two or more tasks. This requires that checks be made upon loading the overlay to determine whether or not the overlay area is already in use. A task waiting for an overlay area that is in use must be suspended until the overlay area is released. Thus, different loading routines are called in single task and multiple task environments, and in a multiple task environment, a call to a routine that releases an overlay after use must be made.

Features common to both single and multiple task environment are discussed first in sections immediately following; then the differing features of single task loading and multiple task loading and the release of overlay areas are described.

Naming an Overlay (OVERLAY)

In both single and multiple task environments, each overlay must have an overlay name assigned to it. Overlay names are assigned in the OVERLAY statement, which has the format:

OVERLAY overlayname

where: overlayname is the name of an overlay.

An OVERLAY statement must be the first statement (except for possible COMPILER DOUBLE PRECISION, COMPILER NOSTACK, or CHANTASK statements) in one of the program units belonging to an overlay. If a single overlay was created from two or more relocatable binaries, each of which contained an OVERLAY statement, each overlayname specified in these statements is associated with that overlay. The overlay can then be referenced by any one of the names.

An overlay name is an external symbol (like the names of subprograms) and must be unique within its first five characters from all other external symbols and all reserved words. Overlay

Naming an Overlay (OVERLAY) (Continued)

names are referenced when loading overlays or releasing overlay areas. Each overlay name must be declared EXTERNAL in any program unit in which it is referenced.

Opening an Overlay File (OVOPN)

In both single and multiple task environments, the overlay file associated with a program using overlays must be opened by execution of a call to the OVOPN routine before any overlays can be loaded. The format of the call to OVOPN is:

CALL OVOPN (channel, filename, error)

where: channel is an integer variable whose value specifies the channel on which the overlay file is to be opened.

filename is the name of the overlay file to be opened (this file name should end with the extension . OL).

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to OVOPN is:

CALL OVOPN (JCHAN, "PGM.OL", IER)

If the value of JCHAN were 7, overlay file PGM.OL would be opened on channel 7, with IER receiving the error code upon completion of the call.

Closing an Overlay File (CLOSE)

In both single and multiple task environments, each overlay file is closed in the same way any file is closed. An overlay file is closed by execution of a call to the CLOSE routine. The call to CLOSE has the format:

CALL CLOSE (channel, error)

where: channel is an integer variable or constant whose value specifies the channel number of the overlay file to be closed.

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to CLOSE is:

CALL CLOSE (7, IER)

Loading Overlays in a Single Task Environment (OVLOD)

In a single task environment, an overlay is loaded by execution of a call to the OVLOD routine. The call has the format:

CALL OVLOD (channel, overlay, conditional-flag, error)

where: channel is an integer variable or constant whose value is the number of the channel on which the overlay file has been opened.

overlay is the name of the overlay which is to be loaded.

conditional-flag is an integer variable or constant whose value indicates conditional or unconditional loading.

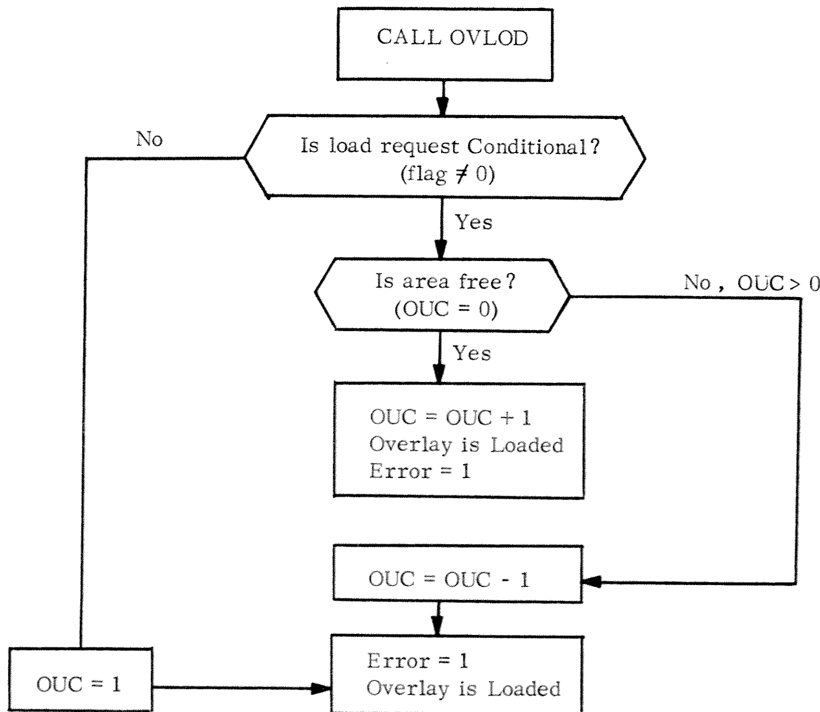
error is an integer variable which will return one of the error codes upon completion of the call.

An unconditional load loads a user overlay regardless of whether the overlay is present in core or not. This permits the initializing of non-reentrant code. A conditional overlay load request, on the other hand, causes a user overlay to be loaded only if it is not already core resident. Conditional loading saves time in some cases but should be used only when overlays are reentrant. For variable conditional-flag, the value zero specifies unconditional loading and a non-zero value specifies conditional loading.

Associated with each overlay is an overlay use count (OUC) that contains a value indicating whether or not the overlay is core resident. If a conditional load has been specified, the OUC is checked. If the OUC contains zero, the overlay may not reside in core or may be core resident but not in use; if the OUC contains one, the overlay is core resident. The conditions for loading an overlay depend upon the state of the OUC and the conditional flag as given below.

1. If the load request is conditional (flag≠ 0) and if the area is free (OUC= 0), the OUC is incremented, the overlay is loaded, and the error return is set to 1 to indicate the overlay has been loaded.
2. If the load request is conditional and if the overlay area already contains the requested overlay (OUC= 1), the overlay remains in the area, the OUC is decremented, and the error return is set to 1 to indicate that the overlay has been loaded.
3. If the load request is unconditional, the OUC is set to 1, the overlay is loaded and the error return is set to 1 to indicate that the overlay has been loaded.
4. If for any reason the overlay cannot be loaded, an appropriate error code is set and a return is made to the calling program.

The conditions specified above are shown in the following chart.



Loading Overlays in a Single Task Environment (OVLOD) (Continued)

An example of a call to OVLOD is:

CALL OVLOD (JCHAN, OV3, IFLAG, IERR)

Loading an Overlay in a Multiple Task Environment (FOVLD)

In a multiple task environment, an overlay is loaded by execution of a call to the FOVLD routine; the call has the format:

CALL FOVLD (channel, overlay, conditional-flag, error)

where: channel is an integer variable or constant whose value is the number of the channel on which the overlay file was opened.

overlay is the name of the overlay to be loaded.

conditional-flag is an integer variable or constant whose value indicates whether the load is to be conditional or unconditional.

error is an integer variable which will return one of the error codes upon completion of the call.

As in the single task environment, the loading of an overlay depends upon the state of the conditional flag and the overlay use count (OUC). However, since overlays and overlay areas can be shared by two or more tasks, the conditions for loading are somewhat more complex.

When a task causes an overlay to be loaded, the task is suspended until the loading process is completed. When a task tries to load an overlay into an overlay area and cannot because the overlay area is already in use, the task is suspended until the overlay area is freed and the desired overlay is then successfully loaded. If more than one task is suspended while waiting for an overlay area to be freed, the task with the highest priority waiting for the overlay area has its desired overlay loaded when the overlay area becomes free. The task then is readied when loading is complete.

The overlay use count is incremented each time a task requests an overlay load and is decremented each time a task causes the overlay to be released (see the section on FOVRL). Since more than one task can use an overlay, the OUC may be greater than 1. An overlay area is only free when the OUC goes to 0. The conditions for loading an overlay in a multiple task environment are as follows:

1. If the load request is conditional (flag ≠ -1) and if the area is free (OUC=0), then the OUC is incremented, the overlay is loaded, and the error return is set to 1 to indicate the overlay has been loaded.

2. If the load request is conditional, and if the area is not free but already contains the requested overlay, the overlay remains in the area, the OUC is incremented, and the error return is set to 1 to indicate that the overlay has been loaded.

3. If the load request is conditional and the area is not free and does not contain the requested overlay, the caller is suspended until the area is freed.

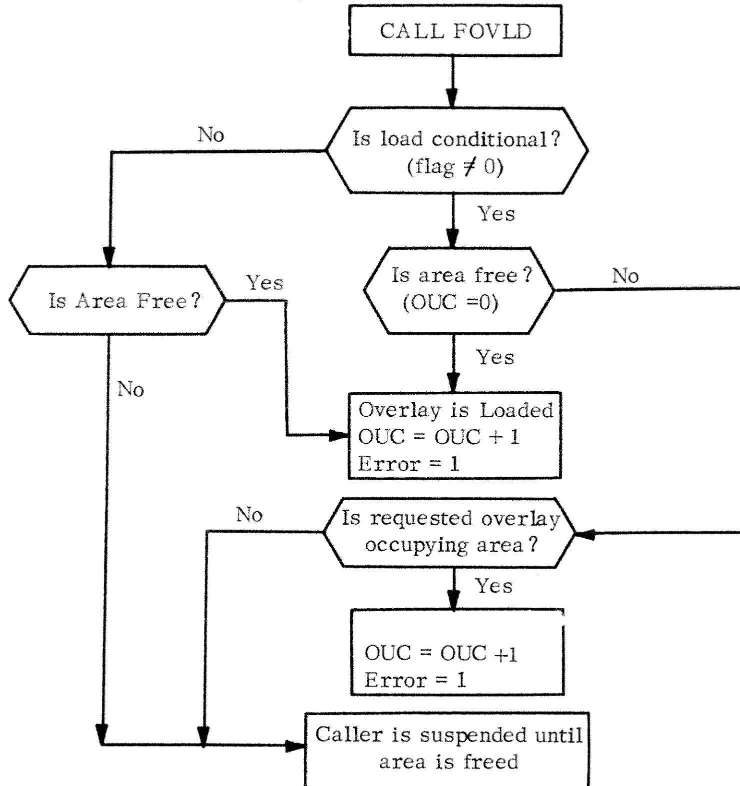
4. If the load request is unconditional (flag= -1), and if the area is free (OUC=0), the OUC is incremented and the overlay is loaded regardless of whether it is core resident or not.

5. If the load request is unconditional and if the OUC has not gone to zero freeing the area, the calling task is suspended until the area becomes free.

Loading an Overlay in a Multiple Task Environment (FOVLD) (Continued)

6. If for any reason the overlay cannot be successfully loaded, the error indicator will be set to the appropriate error code.

The conditions specified above are shown in the following flow chart.



When a task causes an overlay to be loaded, the task is suspended during the loading process as it would be for any other I/O operation. For those cases in which no loading occurs, and the task does not have to wait for an overlay area to become available, the task is not suspended.

An example of a call to FOVLD is:

CALL FOVLD (ICHAN, IOV, ICON, IER)

Releasing an Overlay Area (FOVRL)

All overlay loads (FOVLD) in the multiple task environment must eventually be paired with an overlay release or the area will be reserved indefinitely. An overlay area can be released from outside of the overlay by the execution of a call to the FOVRL routine. The call to FOVRL has the format:

CALL FOVRL (overlay, error)

where: overlay is the name (or any one of the names) of the overlay resident within the overlay area to be released.

error is an integer variable which will return one of the error codes upon completion of the call.

Releasing an Overlay Area (FOVRL)(Continued)

Execution of a call to FOVRL causes the OUC for that overlay area to be decremented. (The overlay area is only freed when OUC goes to zero.) If an overlay other than the one resident in the overlay area is named by overlay, an error condition results and the overlay area is not released. An example of a call to FOVRL is:

CALL FOVRL (AOVLY, IER)

Releasing an Overlay (OVKIL, OVKIX, OVEXT, OVEXX)

Overlays may be released from inside the overlay area, either from the routine in which the overlay was named or from some other routine within the overlay.

A call to OVKIL can be made from the routine in which the overlay was named (OVERLAY statement) and causes the overlay to be released and the task containing the overlay to be killed. The format of the call is:

CALL OVKIL (overlay)

where: overlay is the name of the overlay (specified in an OVERLAY statement).

A call to OVKIX is made from a routine outside that in which the overlay was named. The OVKIX routine causes the overlay to be released and the task containing the overlay to be killed. The format of the call to OVKIX is:

CALL OVKIX (overlay)

where: overlay is the name of the overlay.

A call to OVEXT can be made from the routine in which the overlay is named. It causes the overlay to be released and provides a return location. The format of the call is:

CALL OVEXT (overlay, external-label)

where: overlay is the name of the overlay.

external-label is the external label to which return is made upon completion of the call.

A call to OVEXX is made from outside the routine in which the overlay is named. OVEXX causes the overlay to be released and provides a return location. The format of the call is:

CALL OVEXX (overlay, external-label)

where: overlay is the name of the overlay to be released.

external-label is the external label to which return is made upon completion of the call.

The Overlay Loader (OVLDR)

It is possible to replace one or more overlays within an overlay file. To do so, a file of replacement overlays must be loaded using the overlay loader, which is invoked with the command OVLDR. When the replacement file of overlays has been loaded, overlays within the current overlay file may be replaced by overlays in the replacement file, using the command REPLACE. The replacement of overlays is described in Appendix D in thessection, OPERATION UNDER RDOS.

CHAPTER 6

REAL TIME CLOCK AND CALENDAR

Systems with a Real Time Clock (RTC) maintain a system clock and calendar for scheduling task activities on a time-of-day basis. Tasks may obtain or set the correct time in seconds, minutes, and hours or the current date in month, day and year. Tasks may also synchronize their activities with the real time clock for periods of time as short as one millisecond each.

Six calls are available to permit the system to keep track of the time of day and current date. Dates are alwavs referenced as month/day/year. The time is always given using a 24-hour clock. The six calls are:

CALL FSTIM - set the time of day
CALL STIME - set the time of day
CALL TIME - get the time of day
CALL FGTIM - get the time of day
CALL DATE - get the current date
CALL SDATE - set the current date

Setting the Real Time Clock (FSTIM)

The real time clock can be set using the run time routine FSTIM. Users may access the real time clock in both single and multiple task environments. The format of the call to FSTIM is:

CALL FSTIM (hour, minute, second)

where: hour is an integer variable or constant in the range 0 to 23.

minute is an integer variable or constant in the range 0 to 59.

second is an integer variable or constant in the range 0 to 59.

If an attempt is made to set a time outside the specified legal range, a run time error occurs. The clock used is a 24-hour clock. An example of a call to FSTIM is:

CALL FSTIM (7, 25, 11)

Setting the Real Time Clock (STIME)

A call to the STIME routine allows the user to set the Real Time Clock. The format of the call is:

CALL STIME (array, error)

where: array is a three-element integer array specifying the time to be set in the order of hours, minutes, and seconds.

Setting the Real Time Clock (STIME) (Continued)

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to STIME is:

CALL STIME (IAR, IER)

Getting the Time of Day (TIME)

The time of day can be obtained in the form of a three-element array by a call to TIME which has the format:

CALL TIME (time-array, error)

where: time-array is the name of a three-element integer array that is set equal to the time.

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to TIME is:

CALL TIME (ITAR, IER)

Getting the Time (FGTIM)

The real time clock can be accessed to obtain the time using the routine FGTIM. The format of the call to FGTIM is:

CALL FGTIM (hour, minute, second)

where: hour, minute, and second are integer variables which will return the current hour, minute and second.

The current time will be given in terms of a 24-hour clock. An example of a call to FGTIM is:

CALL FGTIM (IHR, IMIN, ISEC)

Getting the Date (DATE)

The date can be obtained in the form of a three-element array indicating month, day, and year using a call to the DATE routine, which has the format:

CALL DATE (date-array, error)

where: date-array is the name of a three-element integer array that is set equal to the date.

error is an integer variable which will return one of the error codes upon completion of the call.

The first, second, and third elements of array date-array are set equal to the date expressed as month, day, and year. An example of a call to DATE is:

CALL DATE (IAR, IER)

Setting the Date (SDATE)

The user can set the date by issuing a call to the SDATE routine which has the format:

CALL SDATE (array, error)

where: array is a three-element integer array specifying month, day and year in that order.

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to SDATE is:

CALL SDATE (IAR, IER)

User/System Clock Commands

The user may execute a call to the routine DUCLK to define a user clock and a call to RUCLK to remove a user clock. The user clock is a software clock that is controlled at predefined intervals by the operating system clock. This user clock allows control to be given to a user-specified routine when each predefined interval lapses. A call is provided to GFREQ which permits the user to examine the Real Time Clock frequency. It is assumed that the user who is considering the definition of a user clock is familiar with the Real Time Disk Operating System. The following descriptions of calls to run time routines concerning the user/system clock commands should be read in conjunction with the section User/System Clock Commands in the RDOS manual (093-000075).

Define a User Clock (DUCLK)

A call to the DUCLK routine permits the definition of a user clock. When an interrupt is generated by the user clock, the environment becomes frozen as is, and control passes to a user-specified routine at a user-defined location. (This routine cannot be a FORTRAN routine. Furthermore, no system or task calls may be issued from this routine, with the exception of .UCEX and .IXMT .) The format of the call to DUCLK is:

CALL DUCLK (ticks, name, error)

where: ticks is an integer variable or constant specifying the integer number of system RTC cycles which are to elapse between each user clock interrupt.

name is the name of a non-FORTRAN routine to which control is passed upon an interrupt and which must have been previously defined.

error is an integer variable which will return one of the error codes upon completion of the call.

In unmapped systems, the task call .UCEX must be issued to exit from a user clock routine. Refer to the RDOS User's Manual for further information concerning the task call .UCEX.

An example of a call to DUCLK is:

CALL DUCLK (100, IROR, IER)

Remove a User Clock (RUCLK)

To remove a previously defined user clock from the system, the user may issue a call to the RUCLK routine. The call has the format:

CALL RUCLK

Remove a User Clock (RUCLK) (Continued)

The user clock must have been previously defined before it may be removed.

Examine the System Real Time Clock Frequency (GFREQ)

A call to the GFREQ routine permits the user to examine the Real Time Clock frequency. The format of the call is:

CALL GFREQ (variable)

where: variable is an integer variable which will return the frequency of the Real Time Clock, either:

0 - no real time clock in system
1 - 10 HZ
2 - 100 HZ
3 - 1000 HZ
4 - line frequency(60 cycles per second)
5 - line frequency (50 cycles per second)

An example of a call to GFREQ is:

CALL GFREQ (IVAR)

CHAPTER 7

FOREGROUND/BACKGROUND PROGRAMMING

INTRODUCTORY CONCEPTS

As discussed in Chapter 4, a multitasking environment increases the potential utilization of system resources. Multitask environments were understood to exist in a single program environment, and a program was considered to be an orderly collection of tasks.

To increase system utilization still further, it is possible to have two programs sharing system resources concurrently, each with its own single or multi-tasking scheme. This sharing of system resources between two concurrently operating programs is called dual-programming.

In dual-programming, one program is designated as operating in the background, the other as operating in the foreground. The two programs are independent of each other, each containing its own Task Scheduler. The two programs may have equal priority, or the foreground program may be designated as having the higher priority of the two. When the foreground program has the higher priority, control is passed to the background program only when there are no ready tasks in the foreground.

Although the foreground and background programs are independent of each other, they may communicate with each other. This is accomplished by defining a communications area within each program to be used in sending and receiving these messages.

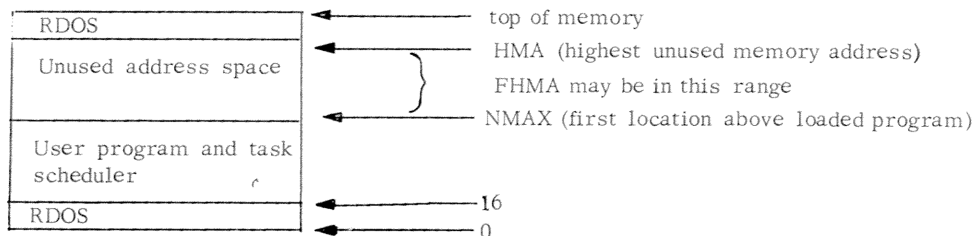
Foreground/Background Considerations in an Unmapped Environment

Systems lacking the MMPU use software memory partitions to separate foreground and background program areas. These boundaries are user-defined at load time, using switches.

In unmapped systems it is useful to include a user-written assembly language program, FHMA, to specify the highest memory location for a FORTRAN program in the background partition. This is necessary in determining the size of the run time stack and in insuring ample memory space for a foreground program. FHMA is loaded just before the run time library and has the source code:

```
.TITL  FHMA
.ENT   FHMA
FHMA = (n)         where n is a value between NMAX and the highest memory ad-
.END               dress available below RDOS
```

In an unmapped, background-only environment, memory can be represented as:



FHMA should be set to an address in the range between NMAX and HMA. Then FHMA to HMA will be reserved for the foreground.

Foreground/Background Considerations in an Unmapped Environment (Continued)

The RLDR command line (and all of its uses) is described in Appendix D, OPERATING PROCEDURES.

Foreground/Background Considerations in a Mapped Environment

Nova 840's, which can include a DGC 8021 Memory Management and Protection Unit (MMPU), provide an absolute hardware protection to separate foreground and background partitions. Foreground save and overlay files in a mapped environment are built in the same way that save and overlay files are built in a single program environment, since an entire page zero and NREL memory is available for both programs.

FOREGROUND/BACKGROUND CALLS

The following calls to be used in foreground/background programming are applicable to both mapped and unmapped systems, except for EXBG, which is used only in a mapped environment.

Load and Execute a Foreground Program

To load and execute a user program in the foreground, the user can issue a call to EXFG. This call can be made only from a background program; it is written in the following format:

CALL EXFG (filename, priority, error)

where: filename is the name of the file to be executed in the foreground.

priority is the integer variable or constant that indicates the priority of the new foreground program, either:

0	-	foreground is of higher priority than background.
1	-	foreground and background are of the same priority.

error is an integer variable that returns one of the error codes on completion of the call.

If the partition requirements set in the RLDR command line would cause any portion of the background program area to be overwritten, the foreground program is not loaded. An example of an EXFG call is:

CALL EXFG ("PROG1", 1, IER)

Checkpointing - Load and Execute a Background Program (EXBG)

Checkpointing is the practice of suspending one background program (the checkpointed program) temporarily so that a new background program can be loaded and executed. Only a foreground program may issue the checkpoint call, EXBG, and the call may only be issued in a mapped environment. To checkpoint a background program, it must be checkpointable, i.e., the program must perform no QTY I/O and must make no calls to RDOPR, DUCLK, FINTD, WAIT, or FDELY.

The checkpointed program is restored to execution when the new background program has been executed or when either a CTRL A or CTRL C keyboard interrupt is detected. There may be only one checkpointed program; nested checkpoints are not allowed.

If the program specified in EXBG is not given higher priority than the currently executing background program, the currently executing background program will complete execution before the program specified in EXBG is loaded and executed. Otherwise, the currently executing background program will be saved on disk until restored to execution.

Checkpointing - Load and Execute a Background Program (EXBG) (Continued)

The call to EXBG has the format:

CALL EXBG (filename, priority, error)

where: filename is the name of the file to be executed in the background.

priority is an integer variable or constant that indicates the priority of the new background program, either:

0 - same priority as present background program.
1 - foreground and background are of the same priority.

error is an integer variable that returns one of the error codes upon completion of the call.

An example of a call to EXBG is:

CALL EXBG ("MAIN6", 0, IER)

See if a Foreground Program is Running (FGND)

A call can be made to the FGND routine from a background program to determine whether or not a foreground program is currently running in the system. The call has the format:

CALL FGND (foreground)

where: foreground is an integer variable which returns a value of 0 or 1:

0 - a foreground program is executing
1 - a foreground program is not executing

An example of a call to FGND is:

CALL FGND (IRUN)

Define a Communications Area (ICMN)

A call to the routine ICMN permits an area to be defined within a user program's address space which will be used for sending or receiving messages to or from another user program. The foreground and the background may each define one communications area. The call to ICMN has the format:

CALL ICMN (array, length, error)

where: array is an array specifying the communications area.

length is an integer variable or constant specifying the size of the communications area in words.

error is an integer variable which will return one of the error codes upon completion of the call.

An example of a call to ICMN is:

CALL ICMN (A, 10, IER)

Write a Message (WRCMN)

A call to the routine WRCMN causes a message to be written by one program, either in the foreground or background, into the other program's communications area. The message that is sent may originate from anywhere within the sender program's address space. The format of the call to WRCMN is:

CALL WRCMN (array, start word, number of words, error)

where: array is an array specifying the origin of the message to be sent to the other program's communications area.

start word is an integer constant or variable specifying the word offset within the communications area to receive the message.

number of words is an integer constant or variable specifying the number of words to be sent.

error is an integer variable which will return one of the error codes upon completion of the call.

Note that start word represents an offset in words. For example, an offset of five will be the fifth element of an integer array but will be the first word of the third element of a real array.

An example of a call to WRCMN is:

CALL WRCMN (IAR, 6, INUM, IER)

Read a Message (RDCMN)

A call to the RDCMN routine causes a message to be read by one program, either in the foreground or background, from the other program's communications area. The message may be received anywhere within the receiving program's address space. The format of the call to RDCMN is:

CALL RDCMN (array, start word, number of words, error)

where: array is an array specifying the destination of the message sent from the other program's communications area.

start word is an integer variable or constant specifying the word offset within the communications area where the message originates.

number of words is an integer constant or variable specifying the number of words to be read.

error is an integer variable which will return one of the error codes upon completion of the call.

Write an Operator Message (WROPR)

A call to the WROPR routine causes an output string to be written from either the foreground or background program areas to the system console, $TTO. The message consists of an ASCII string, less than or equal to 129 characters in length including the required terminator (carriage return, form feed, or null). The system will prefix two exclamation characters and the alphabetic B to the text string, or two exclamation characters and the alphabetic F to the text string.

B = originating from the background
F = originating from the foreground

Write an Operator Message (WROPR) (Continued)

The format of the call to WROPR is:

CALL WROPR (array, error)

where: array is the name of the array containing the text string to be written to $TTO.

error is an integer variable which will return one of the error codes upon completion of the call.

The operator messages (text strings) output on the console after execution of this call will appear as follows:

!!Ftext string) or !!Btext string)

WROPR must not be used if RDOS calls .TRDOP or .TWROP are used or if the RDOS OPCOM package is used.

An example of a call to WROPR is:

CALL WROPR (IAR, IER)

Read an Operator Message (RDOPR)

A call to the RDOPR routine causes an operator message to be transmitted from the system console, $TTI, to either the foreground or the background program. The first character in this message must be a CTRL E character (this is echoed as an exclamation character); the second character must be either of the alphabetics F or B. These alphabetics (F and B) indicate which program is to receive the message:

B = background program is the receiver
F = foreground program is the reciever

If some character other than F or B is typed, no further text string is accepted until a F or B is typed. If the user should try to transmit an unsolicited message (i.e., one for which there is no outstanding read operator message call), the bell is sounded when CTRL E is depressed. The last character in the message string must be the carriage return terminator. The entire message string (including the terminator) can be up to 132 characters in length. The format of the call to RDOPR is:

CALL RDOPR (array, nchar, error)

where: array is an integer array element specifying the location of the message area.

nchar is an integer variable returning the number of characters transferrred (including the terminator). On an error, nchar is set to 0.

error is an integer variable which will return one of the error codes upon completion of the call.

PART II - INDEX

ABORT routine 4-11
absolute addressing mode 1-3
access
  files 3-1
  real time clock 1-2, Chapter 6
  real time calendar 1-2, Chapter 6
  resolution entry 3-3
active tasks 4-1
activating a task 4-4, 4-6
add attributes 3-9
AKILL routine 4-11
aliases 3-3
allocation map 2-2
alphabetics, in file name 3-1
APPEND routine 3-12
appending to a file 3-12
apostrophes 3-1
ARDY routine 4-9
ASCII characters
  changing from lower to upper case 3-11
  in file names 3-1
assign
  attributes 3-9
  new names to multiple file device 3-4
ASSOC routine 4-5
ASUSP routine 4-8
attributes
  changing 3-9
  examining 3-8
  list of 3-8
  setting 3-9
attribute protected file 3-8
automatic restart 3-20

BACK routine 5-3
background program 7-1
bit allocation map 2-2
blocks
  number of in contiguous file 3-4
  reading 3-14, 3-15
  task control 4-1
  writing 3-14, 3-15
bootstrap 2-6
BOOT routine 2-6

calendar commands Chapter 6
card reader ($CDR) 3-1
carriage return, terminating file name 3-1
cassette unit I/O 3-16ff
cassette unit names 3-1
CDIR routine 2-5
CFILW routine 5-4
CHAIN routine 5-4
chaining
  to level zero 5-4
  to programs 5-4, Chapter 1
change
  a file's name 3-5
  attributes 3-9
  current directory 2-3
channel
  associating to a file 3-2
  examining file on a 3-6, 3-7
  freeing a 3-13
  linking for free format I/O 3-16ff
  numbers 3-2
  restoring status (see PART I)
  saving status (see PART I)
CHANTASK statement 4-3
characteristics
  examining 3-8
  mask 3-10, 3-12
CHLAT routine 3-9
CHNGE routine 4-10

CHSTS routine 3-6
clock commands Chapter 6
CLOSE routine 3-13
closing
  a channel 3-13
  a file 3-13
  an overlay file 5-8
communications
  between programs 7-1
  between tasks 4-12
  defining area 7-3
  read a message for 7-4
  write a message for 7-3
concurrent program execution Chapter 7
console interrupts
  disable 2-7
  enable 2-7
contiguous file
  attribute 3-8
  creating 3-4
  organization 1-3
  reading a 3-14
  writing a 3-15
control, device 2-7
control table, device Chapter 3
core image attribute 3-8
CPART routine 2-5
CPU availability 4-1
create
  a file (CFILW) 3-4
  a link entry 3-6
  an overlay file 5-12
  a subdirectory (CDIR) 2-5
CTRL A 2-7
CTRL C 2-7
CTRL F 2-7
current system 2-6
current devices 3-10
current directory 2-3, 2-4

DATE routine 6-2
DCT link 3-7
define
  a user clock 6-3
  communications are 7-3
delete
  a file 3-5
  attributes 3-8
  link entries 3-6

DELETE routine 3-5
depth of resolution 3-3
device
  characteristics 3-10, 3-12
  control 2-7
  default names 3-2
  get current name of 2-4
  master 2-5
  names 3-1
  non-SYSGENed 4-13, 3-20
  spoolable 3-11

DFILW routine 3-5
DIR routine 2-3
direct block record I/O 3-14ff
directory
  get current name of 2-4
  change the current 2-3
  maintenance routines Chapter 2
  default device 2-3
  information, obtaining 3-6
  initialize a 2-3
  specifier 2-3
  file attribute 3-8
  releasing a 2-4
disable
  console interrupts 2-7
  spooling 2-8
discontinue,spooling 2-8

disk
- blocks, number of 3-4
- bootstrap 2-6
- creating a file on 3-4
- deleting a file on 3-5
- discussion of 2-1ff
- name of 3-1
- partitions 2-1ff

display unit ($TTO, $TTI) 3-1, 3-18
DLINK routine 3-6
dollar sign ($) 3-1
dormant state of a task 4-1
dual-program environment Chapter 7
dual-programming Chapter 7
DUCLK routine 6-3
DULNK routine 3-6

EBACK routine 5-3
enable
- console interrupts 3-18
- spooling 2-7, 3-11

environment
- foreground/background 7-1
- mapped 7-1
- multiple task 5-6, 7-1
- single task 5-6, 7-1

EOF 3-14ff
EQUIV routine 3-2, 3-4
error codes 3-2, 2-2
error flags 3-2, 2-2
error recovery procedures, warning 3-17
examine
- a device's characteristic 3-8
- a file's attributes 3-8
- real time clock frequency 6-4

execute a background program 7-2
executing state of a task 4-1
EXBG routine 7-2
EXFG routine 7-2
EXIT routine 4-11, 5-4
extensions
- file name mnemonic 3-7
- to file names 3-1

FBACK routine 5-3
FCHAN routine 5-4
FCLOS routine 3-13
FDELY routine 4-8
FGND routine 7-3
FGTIM 6-2
FHMA user-module 7-1
file
- attribute displacement mnemonic 3-7
- attribute maintenance 3-8ff
- change attributes of 3-9
- directory information 3-6, 3-7
- examine attributes of 3-8
- I/O 3-10ff
- maintenance 3-4ff
- name 3-1
- parceling of space in a 2-5
- name displacement mnemonic 3-7
- size information update 3-8
- structures 1-3

FINRV routine 2-9
FINTD routine 2-9
fixed head disk (DK0, DK1) 3-1
FOPEN routine 3-11
foreground/background programming
- in mapped environment Chapter 7
- introductory concepts Chapter 7
- in unmapped environment 7-1

foreground program, executing a 7-2
form feed, terminating a file name 3-1
FORTRAN IV error messages 3-2, 2-2
FOVLD routine 5-10
FOVRL routine 5-11
FQTASK routine 4-6
free format I/O 3-16ff

freeing a channel 3-13
frequency of the TRC 6-4
FSTAT routine 3-9
FSTIM routine 6-1
FSWAP routine 5-1
FTASK routine 4-4
full initialization 2-3

GCIN routine 3-10
GCOUT routine 3-10
GDIR routine 2-4
get
- attributes 3-8
- current date Chapter 6
- current default directory device name 2-4
- file directory information 3-6
- logical name of master device 2-5
- name of current I/O device 3-10
- time of day Chapter 6
- name of current system 2-6

GFREQ routine 6-4
global specifiers, temporary names
- names of 3-1
- temporary names 3-4

GSYS routine 2-6
GTATR routine 3-8
hardware partition 7-2
highest memory address (FHMA) 7-1
high-speed paper tape
- punch ($PTP, $PTP1) 3-1
- reader ($PTR, $PTR1) 3-1

HIPBOOT 2-6
HOLD routine 4-8

ICMN routine 7-3
identification numbers 4-3
identify user interrupt devices 2-8
incremental plotter ($PLT, $PLT1) 3-1, 3-18
indeterminate error (0) 3-2
INIT routine 2-3
initializing a directory 3-3, Chapter 2
input
- current device 3-10
- dual processor link ($DPI) 3-1

interrupts
- disable 2-9
- enable 2-9
- identifying device for 2-9
- message 2-8
- servicing user 2-8, 2-9

intertask communication 4-12
I/O, free format 3-16ff
ITASK routine 4-4
.IXMT routine 3-20

KILL routine 4-11
keyboard, interrupts 3-18, 3-20

levels of program segmentation 5-1, 5-2
link frequency 6-3
line printer ($LPT, $LPT1) 3-1, 3-18
link
- access attribute word 3-3, 3-7
- aliases 3-3
- chaining attributes 3-9
- entry 3-3, 3-6
- entry attribute 3-3, 3-8
- examining attributes 3-8
- resolution attribute 3-3, 3-8
- to file a 3-3

load
- a foreground save file 7-2
- and execute a background program 7-2
- overlays 5-8ff

magnetic tape I/O 3-16ff
MAP.DR 2-2
master device 3-4, 2-5
MDIR routine 2-5
memory management and protection
unit 7-1
MMPU 7-1
modes
for opening a file 3-10
modification of priorities 4-10
moving head disk 3-1
MTDIO routine 3-17
MTOPD routine 3-16
multiplexor (QTY) 3-1
multiprogramming Chapter 7
multitasking Chapter 4
multitask monitor Chapter 4

name
a file 3-1, 3-2
an overlay file 5-7
a task 4-3
of current system 2-6
of master device 2-5
node points 5-4ff
non-SYSGENed devices 2-9, 4-13
NREL memory 7-2
null, terminating a file name 3-1
numbering
of bytes in last block 3-7
of last block in file 3-7
of significant file name characters 3-7
of user overlays 5-6
numerics, in file name 3-1

obtaining
file directory information 3-6, 3-7
task states 4-11

ODIS routine 2-7
OEBL routine 2-7
OPEN routine 3-10
opening
an overlay file 5-8
cassette unit for free format
I/O 3-16
open
a file (OPEN) 3-10
a file (FOPEN) 3-11
a file for appending (APPEND 3-12
OUC 5-10
output
current devices 3-10
dual-processor link ($DPO) 3-1
spooling of 3-11, 3-18
OVERFLOW routine E-3
OVERLAY statement 5-7
overlay use count (OUC) 5-8ff
overlays 5-4ff
overlay loader 5-14
OVEXT routine 5-12
OVEXX routine 5-12
OVKIL routine 5-12
OVKIX routine 5-12
OVLDR command 5-14
OVLOD routine 5-8
OVOPN routine 5-8
page zero memory 7-2
paper tape
reader ($TTR, TTR1) 3-1
punch ($TTP, $TTP1) 3-1, 3-18
parceling of disk file space 2-5
parent partition 2-5
partial initialization 2-3
partition
attribute 3-8
discussion 2-1ff
primary 2-1ff
secondary 2-1ff, 2-5

perform a disk bootstrap 2-6
periodic execution
  of an overlay 4-6
  of a task 4-6
permanent file attribute 3-8
power failure, restart after 2-9
power-up service 2-9
preassigned channel numbers 3-2
prevent
  interrupts 2-9
  spooling 2-7
PRI routine 4-10
priorities
  modification of 4-10
  numbers 4-3
  of foreground/background programs
    programs 7-1
  of tasks 4-1, 4-3
program
  called 5-1
  calling 5-1
  chaining 5-1, 5-4
  overlays 5-4ff
  segmentation Chapter 5
  swapping 5-1
programmable control of tasks Chap. 4
pulses of the RTC 6-3

QTY 3-1
queuing of output data 2-7, 3-11
quotation marks 3-1

random file
  attribute 3-8
  creating a 3-10
  organization 1-3
  reading a 3-14
  writing a 3-15
random mode 3-10
random record size 3-10, 3-12
RDBLK routine 3-14
RDCMN routine 7-4
RDOPR routine 7-4
RDOS
  discussion Chapter 1, Chapter 7
  file names 3-1
  operating procedures App. D
  system error codes App. B
RDRW routine 3-14
reading
  a message from a program 7-4
  an operator message 7-4
  blocks 3-14
  -only mode 3-10, 3-12
  records 3-14
read-protected file attribute 3-8
READR routine 3-14
ready state of a task 4-1
Real Time Clock Chapter 6
Real Time Disk Operating System
    (see RDOS)
Real Time Operating System (see RTOS)
REC routine 4-12
referencing
  channel numbers 3-2
  files 3-2
  files on magnetic tape 3-2
  files on cassette tape 3-2
release an overlay area 5-11, 5-12
release a directory 2-4
RELSE routine 4-9
remove
  a user clock 6-3
  interrupt devices 2-8
RENAM routine 3-5
renaming a file (RENAME) 3-5
REPLACE command 5-15
reserved
  channel numbers 3-2
  device names 3-1

RESET routine 3-13
resolution
  entry 3-3
  file attributes 3-3
RESTART.SV 2-6
restoring a swapped program 5-2
RLDR command line 4-3, 5-2, 7-2
RLSE routine 2-4
root program 5-4
RTC Chapter 6
RTOS
  discussion Chapter 1
  file names 3-1
RUCLK routine 6-3
run time stack 7-1
run time errors App. B
save

save file attribute 3-8
saving channel status (see PART I)
secondary partition 2-1ff, 2-5
segment of an overlay file 5-5
SDATE routine 6-3
sequential
  creating 3-4
  opening 3-10
  mode 3-10, 3-4
  organization Chapter 1
servicing, user interrupts 3-20
set
  attributes 3-9
  calendar Chapter 6
  time of day 6-1
sharing system resources 7-1
significant characters in file name 3-1
simultaenous peripheral operation on-line (see spooling)
size
  of contiguous file 3-4
  of overlay file 5-4
  of random record 3-10
  of run time stack 7-1
software memory partition 7-1
SOS Chapter 1, App. D
space, terminating a file name 3-1
SPDIS routine 2-8
SPEBL routine 2-7
SPKIL routine 2-8
spooling
  device 2-7, 2-8, 3-11
  disable 2-8
  enable 2-7
  stop 2-8
Stand-alond Operating System (see SOS)
start a task 4-7
starting logical address of a file 3-7
START routine 4-7
STAT routine 3-7
status
  information from file directory 3-6, 3-7
  information on free format I/O 3-16ff
  information on tasks 4-1
STIME routine 6-1
STOP statement 5-4
stop, a spool operation 3-19
STTSK routine 4-11
subdirectory 2-1ff, 2-5
SUSP routine 4-8
suspended state of a task 4-1
SWAP routine 5-1
swapping programs 5-1
SYS.DR 1-3, 2-1ff
system clock commands 6-3, 6-4
system maintenance routines Chap. 2

system directory
- addition to 3-4
- deletion from 3-5
- discussion Chapter 1, 2-5

system errors App. B
system, get name of current 2-6
system utilization, improve upon Chapter 4, Chapter 5

TASK statement 4-4
task concepts 4-1, 4-2, 4-3
task control block 4-1
task scheduler 4-1ff, 7-1
task states 4-1
task status, obtaining 4-11
task identification numbers 4-3
task priority levels Chapter 4
TCB 4-1
temporary names of devices 3-4
terminating
- a file name 3-1
- a task 4-11

TIME routine 6-2
TRNON routine 4-8
TYPE statement 3-2
type
- contiguous 3-4
- of a file 3-4
- of initialization 2-4
- random 3-4
- sequential 3-4

UFD 3-7
unmapped memory Chapters 7 and 1
unresolvable links 3-6
update current file's size information 3-8
UPDATE routine 3-8

user
- addressing mode Chapter 1
- attribute 3-8
- clock commands 6-3, 6-4
- file directory 3-7
- interrupt device
  - identifying 3-20
  - servicing 3-20
  - removing 3-21
- servicing routines 3-20
- -shared
  - reading 3-10, 3-12
  - writing 3-10, 3-12

vector table 3-21

WAIT routine 4-9
WRBLK routine 3-15
WRCMN routine 7-4
write-protected file attribute 3-8
write
- a message between programs 7-3
- a multitask program 4-3
- an operator message 7-4
- blocks 3-14, 3-15
- -only mode 3-10, 3-12
- records 3-14

WRITR routine 3-15
WROPR routine 7-4
WRTR routine 3-15

XMT routine 4-12
XMTW routine 4-12

APPENDICES

APPENDIX A	Summary of FORTRAN IV
APPENDIX B	Error Messages
APPENDIX C	DGC FORTRAN Variations from Standard FORTRAN
APPENDIX D	Operating Procedures
APPENDIX E	Data Storage and Handling
APPENDIX F	Assembler/FORTRAN Interface
APPENDIX G	FORTRAN IV Runtime Re-Entrance at Interrupt Time

CONTENTS OF APPENDICES

APPENDIX A - FORTRAN IV SUMMARY

FORTRAN IV Statement Summary A-3
FORTRAN IV Run Time Call Summary A-9

APPENDIX B - ERROR MESSAGES

Compiler Error Messages B-1
FORTRAN IV Run Time Error Messages B-5
System Error Messages B-6

APPENDIX C - DGC FORTRAN VARIATIONS FROM STANDARD FORTRAN

APPENDIX D - OPERATING PROCEDURES

Operation under RDOS D-1
  Compilation and Assembly D-2
  Compile-time Options D-3
  Loading Procedures D-4
  Loading in a Single Task Environment D-4
  Loading in a Multitask Environment D-5
  The Overlay Loader D-6
  Loading in Foreground/Unmapped Environment D-8
  Loading in Background/Unmapped Environment D-8
  Examples of RLDR Command Lines D-8
  Undefined Symbols D-9
  Debugging D-9
  Merging Library Files D-9
  The CLG Command D-10
Operation under RTOS D-12
Operation under SOS D-14
  Operation under RDOS-Compatible SOS D-14
  Compilation D-14
  Assembly D-16
  Loading D-17
  Execution and Restart Procuedures D-18
  Producing a Trigger D-18
  Possible Error Messages D-19
  Examples D-19
Operation under DOS-Compatible SOS D-21
  Compile-time Options D-21
  Assembly D-22
  Loading D-22
  Restart Procedure D-23
  Execution D-23
Operation of 8K Stand-alone FORTRAN IV D-24
  Operating Procedures D-24
  Language Limitations D-24
Smaller Object Programs D-25
Disk Bootstrapping D-26

APPENDIX E - DATA STORAGE AND HANDLING

Storage of Data E-1
Integers E-1
Real Numbers E-1
Double Precision Numbers E-1
Complex Numbers E-2
Double Precision Complex Numbers E-2
String Data E-2
Logical Data E-3
Data Handling E-3
Number Stack E-3
Byte Manipulation E-3
Overflow Checking E-3

APPENDIX F - ASSEMBLER/FORTRAN INTERFACE

Addressing F-1
Assembler Code Generated by FORTRAN IV F-3
Description of Generated Code F-17
Calling and Receiving Sequences F-20
User Symbols F-22

APPENDIX G - FORTRAN IV RUN TIME REENTRANCE AT INTERRUPT TIME

APPENDIX A

FORTRAN IV SUMMARY

The following pages contain a summary of each call to a run time routine which can be made under FORTRAN IV and each statement which is a part of the FORTRAN IV programming language. Beside each statement or call description is a row of five boxes specifying respectively:

whether or not the call/statement is used under RDOS
whether or not the call/statement is used under RTOS
whether or not the call/statement is used under SOS
a page reference within the manual

An X signifies a positive answer as to whether or not a particular call/statement can be used under a particular operating environment. The page reference column is divided into two sections, the first corresponding to PART I and the second corresponding to PART II.

A summary of the format descriptions used is as follows:

upper case letters	-	are essential parts of the format description and must be used exactly as they appear.
lower case letters	-	are variable portions of the format descriptions; the user, when writing to correspond to the format, will insert his own variable name, device name, file name, etc.
=	-	equals sign is a necessary part of the format description
,	-	comma is an argument delimiter and is a necessary part of the format description.
[ ]	-	broken square brackets indicate optional portions of the format description.
( )	-	parentheses are used to delimit all arguments from the command word. (They are a necessary portion of the format description.)
" "	-	quotation marks are used as ASCII string delimiters and are necessary parts of the format description.
{ }	-	braces are used to denote alternate portions of the format description.
. . .	-	three dots indicate that portions of the format description have been omitted. When they appear, the user should readily see what portions of the format are missing.

FORTRAN IV STATEMENT SUMMARY

Statement Format	RDOS	RTOS	SOS	Page Ref. Part I	Page Ref. Part II
functionname (argument, argument, . . . , argument) = expression assigns the value of an expression to a specified function.	X	X	X	9-1	
variable = expression assigns the value of an expression to a specified variable.	X	X	X	4-1	
ACCEPT list values appearing within the list of the ACCEPT statement are input from the console.	X	X	X	6-20	
ASSIGN statementnumber TO variable causes a subsequent assigned GOTO statement to transfer control to the statement number specified within the ASSIGN statement	X	X	X	5-2	
BLOCK DATA defines a subprogram which contains only DIMENSION, DATA, COMMON, data-type, and EQUIVALENCE statements.	X	X	X	8-2	
CALL subroutine (argument, argument, . . . , argument) references a specified subroutine, replacing dummy arguments with actual arguments.	X	X	X	5-3	
CALL subroutine references a specified subroutine.	X	X	X	5-3	
CHANTASK number-of-channels, number-of-tasks specifies the number of channels that may be open at any one time, and the number of tasks which can be simultaneously active at one time.	X	X			4-3
COMMON name . . . name specifies names of variables and/or arrays to be placed in blank common. The arrays may be dimensioned in the statement.	X	X	X	7-4	
COMMON block-name/list of names . . . /block-name/list of names specifies lists of arrays and/or variables to be placed in labeled common areas defined by block names.	X	X	X	7-4	

FORTRAN IV STATEMENT SUMMARY

Statement Format	RDOS	RTOS	SOS	Page Ref. Part I	Page Ref. Part II
COMPILER DOUBLE PRECISION forces all REAL variables and constants to DOUBLE PRECISION and all COMPLEX to DOUBLE PRECISION COMPLEX.	X	X	X	7-3	
COMPILER NOSTACK causes all non-COMMON variables and arrays to be placed in a fixed location in memory rather than on a run time stack.	X	X	X	7-7	
COMPLEX variable, variable, . . . , variable specifies single precision complex variables and/or arrays. The arrays may be dimensioned in the statement.	X	X	X	7-3	
CONTINUE causes continuation of the normal execution sequence.	X	X	X	5-4	
DATA variable-list/constant-list/... variable-list/constant-list/ defines initial values for variables and array elements.	X	X	X	8-1	
DIMENSION arrayname (subscript bounds), . . . , arrayname (subscript bounds) specifies the subscript bounds of arrays for allocation of storage to the arrays.	X	X	X	7-1	
DO statementnumber variable = integer, integer [,integer] sets up a programming loop.	X	X	X	5-5	
DOUBLE PRECISION variable, variable, . . . , variable specifies double precision variables and/or arrays. The array may be dimensioned in the statement.	X	X	X	7-3	
DOUBLE PRECISION COMPLEX variable, variable, . . . , variable specifies double precision complex variables and/or arrays. The arrays may be dimensioned in the statement.	X	X	X	7-3	
ENDFILE channel causes the file associated with the specified channel to be closed.	X	X	X	6-24	

FORTRAN IV STATEMENT SUMMARY

Statement Format	RDOS	RTOS	SOS	Page Ref. Part I	Page Ref. Part II
EQUIVALENCE (list-of-names), (list-of-names),... (list-of-names) determines shared storage for variables and/or arrays.	X	X	X	7-5	
EXTERNAL subprogram-name, ..., subprogram-name specifies subprograms as external to the program unit in which the specification is made.	X	X	X	7-6	
statementnumber FORMAT (specification) allows for the formatting of input and output data according to a specification.	X	X	X	6-6	
type FUNCTION name (argument, ... , argument) defines a function subprogram.	X	X	X	9-2	
GOTO statement-number causes transfer to a specified statement number.	X	X	X	5-1	
GOTO variable causes transfer to the address which is the current value of the specified variable.	X	X	X	5-1	
GOTO (statement-number1, statement-number2,... statement-numbern), variable causes possible transfer to one of several statement numbers depending on the value of the specified variable.	X	X	X	5-1	
GOTO variable (statement-number1, statement-number2, . . . , statement-numbern) causes transfer to one of several possible statement numbers depending on the value of the specified variable after the last execution of an ASSIGN statement.	X	X	X	5-2	
IF (logical expression) statement causes either execution or bypassing of the specified statement depending on the specified logical expression being true or false.	X	X	X	5-3	

FORTRAN IV STATEMENT SUMMARY

Statement Format	RDOS	RTOS	SOS	Page Ref. Part I	Page Ref. Part II
IF (expression) statement-number1, statement-number2, statement-number3 causes transfer to one of three statement numbers depending on the value of the specified expression.	X	X	X	5-3	
INTEGER variable, variable, ... variable specifies integer variables and/or arrays. The arrays may be dimensioned in the statement.	X	X	X	7-3	
LOGICAL variable, variable, ... variable specifies logical variables and/or arrays. The arrays may be dimensioned in the statement.	X	X	X	7-3	
OVERLAY overlayname names an overlay.	X				5-8
PARAMETER variable = constant, ... variable = constant assigns values to symbolic names, which may then be used like constants throughout the program.	X	X	X	2-1	
PAUSE [string] causes the program to cease execution with an optional message printed at the console.	X	X	X	5-5	
READ (channel) [list-of-variables] READ (channel, format) [list of-variables] reads from a device or file the data associated with the variables in the list; formatting may be preset (unformatted I/O) or in accordance with a format specified by the user.	X	X	X	6-1	
READ (channel, [format,] {ERR / END} = statementnumber) [list] READ (channel, [format,] {ERR / END} = statementnumber, {ERR / END} = statementnumber) [list] reads information (as in READ description above) and also allows the user to gain control after an end-of-file or an I/O error at the driver level has been detected.	X	X	X	6-23	

FORTRAN IV STATEMENT SUMMARY

Statement Format	RDOS	RTOS	SOS	Page Ref. Part I	Page Ref. Part II
READ BINARY (channel) list transfers binary data from an external medium.	X	X	X	6-20	
REAL variable, . . . , variable specifies real variables and/or arrays. The arrays may be dimensioned in the statement.	X	X	X	7-3	
RETURN { variable } indicates the logical end of a subprogram, by default, causing a normal return when executed. Optionally, the user may cause an abnormal return.	X	X	X	5-4	
REWIND channel causes the file associated with the specified channel to be positioned at the initial record.	X	X	X	6-23	
STOP { string } causes an unconditional termination of a program's (or a task's) execution, and optionally causes a message to be printed at the console.	X	X	X	5-5	
SUBROUTINE name (argument, . . . , argument) defines a subroutine subprogram unit.	X	X	X	9-8	
TASK taskname assigns a name to a task program unit.	X	X			4-4
TYPE list causes output of the values of the variables specified in the statement.	X	X	X	6-20	
WRITE (channel) { list-of-variables } WRITE (channel, format) { list-of-variables } write to a device or file the data associated with the variables in the list; formatting may be preset (unformatted I/O) or in accordance with a format specified by the user.	X	X	X	6-1	

FORTRAN IV STATEMENT SUMMARY

Statement Format	RDOS	RTOS	SOS	Page Ref. Part I	Page Ref. Part II
WRITE (channel, [format,] {END/ERR} = statementnumber) [list] WRITE (channel, [format,] {END/ERR} = statementnumber, {END/ERR} = statementnumber [list] writes information (as in WRITE description on previous page) and also allows the user to gain control after an end-of-file or after an I/O error has been detected.	X	X	X	6-23	
WRITE BINARY (channel) list transfers data in binary to an external medium.	X	X	X	6-20	

FORTRAN IV RUN TIME CALL SUMMARY

Call Format	RDOS	RTOS	SOS	Page Ref. Part I	Page Ref. Part II
CALL ABORT (id, error) causes termination of the task with the specified identification number.	X	X			4-11
CALL AKILL (priority-number) terminates all ready or executing tasks with the specified priority number.	X	X			4-11
CALL APPEND (channel, filename, {mode, / array,} error [, size]) opens a file for appending	X	X			3-12
CALL ARDY (priority-number) causes all tasks of a given priority number to be readied.	X	X			4-10
CALL ASSOC (task, id, priority-no., error [, IASM] associates a task with an identification number, without causing that task to go to ready state.	X				4-5
CALL ASUSP (priority-number) causes all tasks of a given priority number to be suspended.	X	X			4-9
CALL BACK causes the last program that was swapped out to disk to be restored to core.	X				5-3
CALL BCLR (word, position) sets a single bit in a word to zero.	X	X	X	9-10	
CALL BOOT (device, error) perform a disk bootstrap.	X				2-6
CALL BSET (word, position) sets a single bit in a word to one.	X	X	X	9-10	
CALL CDIR (name, error) creates a subdirectory with a specified name.	X				2-5
CALL CFILW (filename, type, [, size], error) creates an RDOS disk file.	X				3-4

FORTRAN IV RUN TIME CALL SUMMARY

Call Format	RDOS	RTOS	SOS	Page Ref. Part I	Page Ref. Part II
CALL CHAIN (filename, error) causes the current program's core image to be overwritten by another program loaded from disk.	X				5-4
CALL CHLAT (channel, attributes, error) causes a change, addition, or deletion of link file access attributes.	X				3-9
CALL CHNGE (identification, priority-number, error) causes the priority number of a specified task to be changed.	X				4-10
CALL CHRST (channel, start-word) restores previously saved channel status to enable rereading and rewriting of records.	X			6-25	
CALL CHSAV (channel, start-word) saves the status of a channel to enable rereading or rewriting of records.	X			6-24	
CALL CHSTS (channel, array, error) returns a copy of the current directory status information for a file on the specified channel.	X	X	X		3-6
CALL CLOSE (channel, error) closes a file.	X	X	X		3-13, 5-9
CALL CPART (name, size, error) creates a secondary partition.	X				2-5
CALL DATE (date-array, error) gets the current date.	X	X			6-2
CALL DELETE (filename) deletes a file.	X				3-5
CALL DFILW (filename, error) deletes an RDOS disk file.	X				3-5
CALL DIR (directoryname, error) changes the current default directory device.	X				2-3

FORTRAN IV RUN TIME CALL SUMMARY

Call Format	RDOS	RTOS	SOS	Page Ref. Part I	Page Ref. Part II
CALL DLINK (name1, [name2,] error) creates a link entry in the current directory to a file in another directory.	X				3-6
CALL DUCLK (ticks, address, error) permits the definition of a user clock.	X	X			6-3
CALL DULNK (name, error) deletes a link entry in the current directory.	X				3-6
CALL EBACK (error) returns the last swapped program back to disk, or if there is no such program, causes return to level 0 - the CLI.	X				5-3
CALL EQUIV (name1, name2, error) assigns a new name to the multiple file device.	X				3-4
CALL EXBG (name, priority, error) loads and executes a program in the background.	X				7-3
CALL EXFG (name, priority, error) loads and executes a program in the foreground.	X				7-2
CALL EXIT causes termination of executing task.	X	X			5-4, 4-11
CALL FBACK causes the last program that was swapped out to disk to be restored to core.	X				5-3
CALL FCHAN (filename) causes current program's core image to be overwritten by another program loaded from disk.	X				5-4
CALL FCLOS (channel) closes a file on a specified channel and frees the channel.	X	X	X		3-13
CALL FDELY (number-of-pulses) suspends a task for a specified amount of time.	X	X			4-9

FORTRAN IV RUN TIME CALL SUMMARY

Call Format	RDOS	RTOS	SOS	Page Ref. Part I	Page Ref. Part II
CALL FGND (foreground) determines whether or not a foreground program is running.	X				7-3
CALL FGTIM (hour, minute, second) gets the current time.	X	X			6-2
CALL FINRV (device-code) removes a user interrupt device from the system interrupt vector table.	X	X			2-10
CALL FINTD (device-code, dct) specifies a device which is capable of generating interrupt requests.	X	X			2-9
CALL FOPEN (channel, filename [, "B"] [, recordbytes]) assigns a specified channel to a file (device) and opens that file or device.	X	X	X		3-11
CALL FOVLD (channel, overlay, condition-flag, error) loads overlays in a multiple task environment.	X				5-11
CALL FOVRL (overlay, error) releases a specified overlay.	X				5-12
CALL FQTASK (overlayname, task, array, error [, type] causes periodic execution of a task or overlay.	X				4-6
CALL FSEEK (channel, recordnumber) positions a random file to a given record.	X			6-24	
CALL FSTAT (channel, attributes, error) sets or changes the attributes of a file.	X				3-9
CALL FSTIM (hour, minute, second) sets the real time clock.	X	X			6-1
CALL FSWAP (filename) causes the current program's core image to be saved on disk, and another program to be loaded from disk.	X				5-1

	RDOS	RTOS	SOS	Page Ref. Part I	Page Ref. Part II
CALL FTASK (taskname, error-return, priority-number {, IASM}) activates a task by task name.	X	X			4-4
CALL GCIN (array) obtains the current input device name.	X	X			3-10
CALL GCOUT (array) obtains the current output device name.	X	X			3-10
CALL GDIR (array, error) returns the name of the current default directory/device name.	X				2-4
CALL GFREQ (variable) examines the Real Time Clock (RTC) frequency.	X	X			6-4
CALL GSYS (array, error) gets the name of the current system.	X				2-6
CALL GTATR (channel, attributes, error) examines the attributes of a file.	X				3-8
CALL HOLD (identification, error) causes the task with the specified identification number to be suspended.	X	X			4-9
CALL ICLR (word, position) sets a single bit in a word to zero.	X	X	X	9-10	
CALL ICMN (array, length, error) defines an area in a program's address space which will be used for sending and receiving messages.	X				7-3
CALL INIT (directoryname, type, error) causes a directory to be initialized.	X	X			2-3
CALL IOPC ([program-array, number of programs, queue array, number of queues, overlay channel,] error) Initializes OPCOM package.	X				4-14

	RDOS	RTOS	SOS	Page Ref. Part I	Page Ref. Part II
CALL IOPROG (program name, program number, task identifier, task priority, [overlaynode/number, conditional load], error [ASM]) builds a program table of task information.	X				4-15
CALL ISET (word, position) sets a single bit in a word to one.	X	X	X	9-10	
CALL ITASK (taskname, id, priority-number, error {, IASM}) activates a task and associates an id number with the task name.	X	X			4-4
CALL KILL kills the executing task.	X	X			4-11
CALL MDIR (array, error) obtains the current master device name.	X				2-5
CALL MTDIO (channel, commandword, I/O-array, status, error {, count} permits the operation of magnetic tape and cassette units on a machine level.	X	X			3-17
CALL MTOPD (channel, filename, mask, error) opens a magnetic tape or cassette unit for free format I/O.	X	X			3-16
CALL ODIS disables console interrupts.	X				2-7
CALL OEBL enables console interrupts.	X				2-7
CALL OPEN (channel, filename, {mode, / array,} error {, size}) opens a file.	X	X	X		3-10
CALL OVERFLOW (statement1, statement2 {"N" / "S"}) checks for floating point overflow and returns to statement1, or statement2 depending upon whether or not overflow occurred.	X	X	X		E-3

	RDOS	RTOS	SOS	Page Ref. Part I	Page Ref. Part II
CALL OVEXT (overlay, return-location) causes an overlay to be released and provides a return location.	X				5-13
CALL OVEXX (overlay, return-location) causes an overlay to be released and provides a return location.	X				5-13
CALL OVKIL (overlay) causes an overlay to be released and the task containing the overlay to be killed.	X				5-13
CALL OVKIX (overlay) causes an overlay to be released and causes the task containing the overlay to be killed.	X				5-13
CALL OVLOD (channel, overlay, conditional-flag, error) loads overlays in a single task environment.	X				5-8
CALL OVOPN (channel, filename, error) opens an overlay file.	X				5-8
CALL PRI (priority-number) changes the priority number of an executing task.	X	X			4-10
CALL RDBLK (channel, sblock, array, nblock, error {, iblk}) causes a series of blocks to be read from a contiguously or randomly organized file.	X	X			3-14
CALL RDCMN (array, start-word, number-of-words, error) reads a message from another program's communication area.	X				7-4
CALL RDOPR (array, nbyte, error) reads an operator message.	X				[illegible]5
CALL RDRW (channel, srec, array, nrec, error {, nbyte}) causes a series of records to be read from a file into an array.	X				3-14

	RDOS	RTOS	SOS	Page Ref. Part I	Page Ref. Part II
CALL READR (channel, srec, array, nrec, error {, nbyte}) causes a series of records to be read from a file into an array.	X				3-14
CALL REC (message-key, message-destination) receives a one-word message.	X	X			4-12
CALL RELSE (identification, error) causes the task with the specified identification number to be readied.	X	X			4-10
CALL RENAM (oldfilename, newfilename, error) renames a disk file.	X				3-5
CALL RESET closes all open files.	X	X	X		3-13
CALL RLSE (directoryname, error) closes and releases all files of a given directory.	X	X			2-4
CALL RUCLK removes a previously defined user clock.	X	X			6-3
CALL SDATE (array, error) sets the date.	X	X			6-3
CALL SPDIS (devicename, error) disables spooling on a specified device.	X				2-8
CALL SPEBL (devicename, error) enables spooling on a specified device.	X				2-7
CALL SPKIL (devicename, error) stops a spool operation which is currently being performed.	X				2-8
CALL START (id, time, unit, error) starts a task after a specified time delay.	X	X			4-7
CALL STAT (filename, array, error) obtains current status of a given file.	X				3-7

	RDOS	RTOS	SOS	Page Ref. Part I	Page Ref. Part II
CALL STIME (array, error) sets the time of day.	X	X			6-1
CALL STTSK (id, status, error) obtains current status of a task.	X	X			4-11
CALL SUSP causes an executing task to be suspended.	X	X			4-8
CALL SWAP (filename, error) causes the current program's core image to be saved on disk, and another program to be loaded into core from disk.	X				5-1
CALL TIME (time-array, error) gets the current time of day.	X	X			6-2
CALL TRNON (id, array, error) executes a task at a specified time.	X	X			4-8
CALL UPDATE (channel, error) permits the current file's size information to be updated.	X				3-8
CALL WAIT (time, units, error) causes executing task to be suspended for specified amount of time.	X	X			4-9
CALL WRBLK (channel, sblock, array, nblock, error {, iblk}) causes a series of blocks to be written into a disk file from an integer array.	X	X			3-15
CALL WRCMN (array, start-word, number-of-words, error) causes a message to be written by one program into another program's communication area.	X				7-4
CALL WRITR (channel, srec, array, nrec, error {, nbyte}) causes a series of records to be written into a file.	X				3-15
CALL WROPR (array, error) writes an operator message.	X				7-4

	RDOS	RTOS	SOS	Page Ref. Part I	Page Ref. Part II
CALL WRTR (channel, srec, array, nrec, error {, nbyte}) causes a series of records to be written to a file.	X				3-15
CALL XMT (message-key, message-source, error-return) transmits one-word messages between active tasks.	X	X			4-12
CALL XMTW (message-key, message source, error-return) transmits a one-word message between active tasks and waits until the message has been received.	X	X			4-13

APPENDIX B

ERROR MESSAGES

COMPILER ERROR MESSAGES

Error checking by the FORTRAN IV compiler is quite extensive. Syntax, identifier usage conflict, and allowable variable types in arithmetic expressions are all thoroughly checked.

Whenever possible, the statement scan is continued after an error is detected and noted. This is done for non-syntactic errors in declaration statements and expression evaluation. In the scan of FORMAT statements, recovery will be attempted under certain conditions.

Obviously, one error may lead to later error messages because information which should have been available to the compiler at this later point is not available.

An error message consists of one or two lines. The FORTRAN source line is typed preceding the first error detected, followed by the error code (s). Sometimes the FORTRAN source line given in the message is not the line containing the error but the succeeding one. This occurs because some errors are not detected until it has been verified that the line following is not a continuation line; by that time, the erroneous line is not available for output.

In specification statements, certain errors are detected when all declarations are being resolved and the first non-declaration line is in the buffer. Error messages resulting will be qualified by a second line specifying at least one of the identifiers involved in the error detected.

Error messages 61 and 76 will be qualified with the statement number in question.

Error messages are output to the teletype in all cases and to the listing device if different from the teletype. Error messages are always preceded by semicolons. A semicolon indicates to the assembler that the remainder of the line is a comment. Its use permits the listing and output devices (or the error and output devices) to be the same.

Each error message terminates with a decimal character count. This refers to the last character scanned and indicates that the source error occurred somewhere within the statement at or prior to the character given in the character count. Character count does not equal the column number, except when no tabs precede the character in question.

Some examples of error messages are:

```
;             DATA CP1/1,1.DØ/CP2/2.D-5,.Ø1D2/
;***050*** △CHR△15
```

(Presume in the example above that CP1 and CP2 are double precision complex variables.)

```
;1            FORMAT(1H0,1P3E15.4,F8.2)
;***051***△CHR△03
;    L2
```

(The error, as indicated by the variable, L2, occurred in a specification statement preceding the FORMAT statement.)

```
;             L1 = R3+1..GE.*R4
;***013***△CHR△14
```

In the list of error messages that follows:

N - Means that the syntax error is not necessarily fatal.

C - Means the scan of the statement is continued if the error is a syntax error. The continued scan applies only to syntax errors; errors at a diffferent level may or may not allow the scan to continue.

In FORMAT statements, the error is generally fatal. In declaration statements, if a conflict occurs, the last declaration for the identifier is ignored.

FORTRAN IV ERROR MESSAGES

Code	Meaning
00	Working space exhausted. Fatal, but compiler continues.
01	Multiply-defined parameter.
02 N	Mixed precision operands.
03 N	Unknown statement type.
04 N	Something other than blanks at statement end.
05	Syntax error in DATA variable list.
06	Syntax error in DATA literal list.
07	Syntax error in statement function.
10 C	Missing integer in FORMAT.
11	Error in parameter list of CALL.
12	Array identifier not followed by a left or right parenthesis or comma.
13	Illegal element in expression.
14	Improper use of array name.
16	Missing operator.
17	Illegal sequence of adjacent operators.
20	Illegal element/operator when "(" or literal or variable expected.
21	Premature statement end for an IF.
22	Trailing "." missing in operator such as .EQ.
23	Illegal continuation line (after comment or having label).
24	"." not followed by letter or number.
25 C	Format error.
26 C	Format error after repeat count. (Errors 25 and 26 together indicate an illegal character. These errors may repeat on one statement.)
27	Abnormal end to FORMAT statement.
30	Expression didn't close at end of statement.
31	Multiply-defined error.
32	Variably-dimensioned array not a dummy.
33	Variable list longer than value list in DATA.
34 C	Identifier in more than one type declaration.
35	Unclosed DO loop in program.
36	Common variable previously declared EXTERNAL, subprogram or dummy.
37 C	Dummy identifier predefined.
40 C	Dimension error.
41	Improper statement terminating DO loop.
42 C	Variable dimension for main program array.
43	Array size is greater than 32K.
44	Parentheses don't close before statement end.
45	Expected numeric operand for unary minus.
46	Expected logical operand for .NOT.

FORTRAN IV ERROR MESSAGES (Continued)

Code	Meaning
47 N	Illegal operand types for current operator.
50 C	Data statement error; types don't match.
51	Both members of equivalence pair in common.
52	Beginning of common extended by equivalence.
53	Irrecoverable format error.
54	Statement function name in conflict with previous declaration.
55	Multiply-defined dummy identifier in statement function.
56 C	Too few subscripts in DATA or EQUIVALENCE.
57 C	Subscripts out of bounds in DATA or EQUIVALENCE.
60 C	Formal syntactical structure of statement is in error, punctuation is missing or an identifier is of the wrong variety.
61	Undefined label.
62	Attempt to load or store external or array.
63	Array element can't be specified for a dummy array.
64 C	Identifier in EXTERNAL previously declared in other than type declaration.
65 C	A variable dimension is not a dummy.
66	Variable on DATA list not in labeled COMMON.
67	Two variables, neither in COMMON, are equivalenced.
70	A subscript is not type integer.
71	Wrong number of arguments for reserved name function.
72	Wrong type of arguments for a reserved name function.
73 N	Non-digit in label field.
74 N	Carriage return in label field.
75	Improper statement in block data subprogram.
76 N	Unreferenced label.
77	Stack variable referenced in statement function.
100 C	Variable stack has no room for all run time variables.
101	Undeclared identifier in statement function expression.
102	RETURN statement in main program.
104	$ followed by something other than a digit.
105	End of file without END.
106	Wrong number of subscripts.
111	Hollerith constant not ended at statement end.
112 C	Truncated integer. Magnitude greater than 2**15-1 .
114 C	Exponent error in real.
115 C	Exponent error in double precision.
116 C	Illegal character for FORTRAN statement.
120	Literal error of one of the following types: (a) two operands not both literals, (b) two literals of different types, or (c) source line is (literal, literal operator where: operator is not a right parenthesis.
140-160	Compiler errors for debugging only.

FORTRAN IV RUN-TIME ERROR MESSAGES

Error Number	Meaning
1	Stack overflow
2	Computed GOTO error
4	Division by zero
5	Integer overflow
6	Integer power error (illegal or overflow)
7	Floating point underflow.
8	Floating point overflow
9	Illegal format syntax
11	Logic conversion error
13	Number conversion error
14	I/O error
15	Field error (i.e., F5.10, E5.4, etc.)
16	Square root of negative number
17	Log of negative number
18	Channel not open
19	Channel already open
20	No channels available
21	System exceptional status *
24	Exponential over/underflow
25	Array element out of bounds
26	Negative base for floating-point power
27	Number stack overflow
28	BACKSPACE not implemented
29	Attempt to restore status of channel when the status was not saved.
30	Queued task error.
31	Seek on a non-random file.
32	Overlay aborted
33	Illegal argument
34	Delete error (file open)
35	Overlay error in overlay kill.
36	Undefined entry. **

---

* This error is generated when a system-related function (e.g., setting time) encounters an error (e.g., invalid time) and has no way to return an error indication to the FORTRAN program. Note that CALL FSTIM (hour, min, sec) cannot indicate the error. However, CALL STIME (array,ierror) provides for an error and consequently, processing continues.

** This error occurs when an attempt is made to call a subroutine that was not loaded.

SYSTEM ERROR MESSAGES

FORTRAN Code	RDOS Code	Meaning
0		Indeterminate error
1		Call successfully completed
2		Activity in progress
3	0	Illegal channel number
4	1	Illegal file name
5	2	Illegal system command
6	3	Illegal command for device
7	4	Not a saved file
8	5	Attempt to write an existent file.
9	6	End of file.
10	7	Attempt to read a read-protected file.
11	10	Attempt to write a write-protected file.
12	11	Attempt to create an existent file.
13	12	Attempt to reference a non-existent file.
14	13	Attempt to alter a permanent file.
15	14	Illegal attempt to change file attributes.
16	15	Attempt to reference an unopened file
17	16	(not assigned)
18	17	(not assigned)
19	20	(not assigned)
20	21	Attempt to use a channel already in use
21	22	Line limit exceeded on read or write line
22	23	Attempt to restore a non-existent image.
23	24	Parity error on read line
24	25	Trying to push too many levels
25	26	Attempt to allocate more memory than available
26	27	Out of file space
27	30	File read error
28	31	Unit not properly selected
29	32	Illegal starting address
30	33	Attempt to read into system area
31	34	File accessible by direct block I/O only
32	35	Files specified on different directories
33	36	Illegal device code
34	37	Illegal overlay number
35	40	File is not accessible by direct block I/O
36	41	Attempt to set illegal time or date
37	42	Out of TCB's
38	43	Message address is already in use
39	44	File already squashed error
40	45	Device already in system
41	46	Insufficient number of free contiguous disk blocks
42	47	QTY error
43	50	Illegal information in task queue table.
44	51	Attempt to open too many devices or directories

FORTRAN Code	RDOS Code	Meaning
45	52	Illegal directory specifier
46	53	Directory specifier unknown
47	54	Directory is too small
48	55	Directory depth is exceeded
49	56	Directory in use
50	57	Link depth exceeded
51	60	File is in use
52	61	Task ID error
53	62	Common size error
54	63	Common usage error
55	64	File position error
56	65	Insufficient room in data channel map
57	66	Directory/device not initialized
58	67	No default directory
59	70	Foreground already exists
60	71	Error in partition set
61	72	Directory in use by another program
62	73	Not enough room for UFTs
63	74	Illegal address
64	75	Not a link entry
65	76	Program to be checkpointed is not checkpointable, or attempt to create two outstanding checkpoints
66	77	Error detected in SYS.DR
67	100	Error detected in MAP.DR
68	101	Ten second disk time-out occurred
69	102	Entry not accessible via a link
70	103	MCA request outstanding
71	104	Incomplete MCA transmission/request
72	105	System deadlock
73	106	Input terminated by channel close
74	107	Spool file(s) active
75	110	Task not found for ABORT

APPENDIX C

DGC FORTRAN VARIATIONS FROM STANDARD FORTRAN

1. Comments may be placed on the same line with statements. The syntactical scan of the line ends at a semicolon (;) and comments may follow the semicolon delimiter.

2. Variables may be typed DOUBLE PRECISION COMPLEX.

3. When declaring arrays, upper and lower bounds may be given for subscripts of arrays; thus the lower bound of an array subscript does not have to be zero but can be any integer including negative integers. A colon delimits the lower from the upper bound.

4. An array may have up to 128 dimensions.

5. Subscripts of array elements in executable statements (other than lists of I/O statements) may be any form of expression whose value is type integer.

6. String constants enclosed in quotation marks or in apostrophes may be used instead of Hollerith constants.

7. Formatting includes the tabulation format descriptor, T*w*, tab to column *w*.

8. Abnormal returns are allowed from subprogram units.

9. All variables not stored in COMMON are placed on a run-time stack. Any program that does not alter COMMON storage is therefore a reentrant program.

10. Program units must be ordered as follows:

    a. COMPILER DOUBLE PRECISION and COMPILER NOSTACK statements.
    b. OVERLAY and CHANTASK statements.
    c. PARAMETER statements.
    d. FUNCTION, SUBROUTINE, or TASK statement.
    e. Declaration statements, which begin with the keywords: COMMON, COMPLEX, DIMENSION, DOUBLE, EQUIVALENCE, EXTERNAL, INTEGER, LOGICAL, or REAL.
    f. Statement functions. (FORMAT statements and DATA initialization statements may be given in this area.)
    g. Executable statements. (FORMAT statements and DATA initialization statements may be given in this area.)

11. Imbedded blanks are significant except when they appear in the name of a program variable or in the statement identifier GOTO (GO TO).

12. Statement identifiers, operator names, and names of library functions are reserved and cannot be used as program variables. The reserved names are:

12. (Continued)

.AND.
.EOT.
.EQ.
.FALSE.
.GE.
.GT.
.LE.
.LT.
.NE.
.NOT.
.OR.
.TRUE.
ABS
ACCEPT
AIMAG
AINT
ALOG
ALOG10
AMAX10
AMAX1
AMIN0
AMIN1
AMOD
ASSIGN
ATAN
ATAN2
BINARY
BLOCK DATA
CABS
CALL
CCOS
CEXP
CHANTASK
CLOG
CMPLX
COMMON
COMPILER
COMPLEX
CONJG
CONTINUE
COS
CSIN
CSQRT
DABS
DAIMAG
DATA
DATAN
DATAN2
DATN2
DBLE
DCABS
DCCOS
DCEXP
DCLOG
DCMPLX
DCOS
DCSIN
DCSQRT
DEXP
DFLOAT
DIM
DIMENSION
DLOG
DLOG10
DMAX1
DMIN1
DMOD
DO
DOUBLE PRECISION
DREAL
DSIGN
DSIN
DSQRT
DTAN
DTANH
END
ENDFILE
ENTRY
EQUIVALENCE
ERR
EXP
EXTERNAL
FLOAT
FORMAT
FUNCTION
GOTO
IABS
IAND
IDIM
IDINT
IEOR
IF
IFIX
INT
INTEGER
IOR
ISHIFT
ISIGN
ITEST
LOGICAL
MAX0
MAX1
MIN0
MIN1
MOD
NOSTACK
NOT
OVERLAY
PARAMETER
PAUSE
READ
REAL
RETURN
REWIND
SIGN
SIN
SINH
SNGL
SQRT
STOP
SUBROUTINE
TAN
TANH
TASK
TO
TYPE
WRITE

Names identical to DGC extended assembler mnemonics are not available for use as subprogram names.

13. An assigned GO TO is treated as an unconditional GO TO.

14. Statements with an X in column 1 are compiled only if the X option is true at compile time.

15. Generated code treats logical variables as full words, thus providing for 16-bit logical operations. When testing for a truth value, any non-zero word = .TRUE.

16. Octal numbers can be read and written under FORMAT control.

17. Binary data can be read and written using READ BINARY and WRITE BINARY statements.

18. Unformatted I/O leaves all conversion between internal and external forms up to the I/O processor.

19. Variable names may be up to 31 characters in length.

20. Hollerith strings are permitted in the lists of I/O statements.

21. Specific verbs, TYPE and ACCEPT, are used for teletype I/O.

22. Combined input and output is allowed in the ACCEPT statement.

23. Sw string field descriptor is accepted in FORMAT specifications.

24. Mixed arithmetic expressions combining integer with real and/or double precision quantities are accepted.

25. Hollerith data may appear in integer arithmetic expressions and will be interpreted as integer data.

26. Octal constants can be specified in the FORTRAN source program as ± d . . . dK, where each d is an octal digit.

27. DATA initialization is provided for labeled COMMON only.

28. Only COMMON variables can be EQUIVALENCed.

29. DATA initialization of labeled COMMON is possible in any FORTRAN program or subprogram.

30. Subprogram names must be unique within the first five characters (ANSI standard is six).

31. A repeat count cannot be used with a Hollerith constant in a DATA initialization statement.

32. PARAMETER statements can be used to define names for constants.

33. Under the Real Time Operating System, a multitasking environment is provided as well as a single task environment. A task is a FORTRAN program unit and is defined in source language beginning with a TASK statement and terminating with an END. The FORTRAN task scheduler is used in multitasking and all tasking functions are handled at run time by run time tasking routines and the scheduler.

34. Under RDOS, FORTRAN run time routines allow the user to identify to the RDOS system a device capable of generating interrupts.

35. Under RDOS, FORTRAN run time routines provide access to the real time clock.

APPENDIX D

OPERATING PROCEDURES

There are several operating procedures within this appendix, they are:

Operation under RDOS .... Page D-1
Operation under RTOS .... Page D-12
Operation under RDOS-compatible SOS .... Page D-14
Operation under DOS-compatible SOS .... Page D-24
Operation of 8K Stand-alone FORTRAN IV .... Page D-25
Operation of HIPBOOT .... Page D-26

Turn to the appropriate procedures corresponding to the environment your FORTRAN IV program will be operating in.

OPERATION UNDER RDOS

The FORTRAN IV compiler is supplied to the user in the form of two dumped tapes.

Dump Tape 1, FIV. SV -088-000032

Dump Tape 2, FORT. SV, CLG. SV -088-000033

Before invoking the compiler, the user must create save files from the tapes using the LOAD command. After the compiler has been LOADed, the FORTRAN library tapes must be transferred to the disk using the XFER command. The library tapes to be transferred are:

RTIOS (099-000072)	The Real Time I/O system library. (Transfer the tape first when using Analog-to-Digital equipment; otherwise, the tape may be ignored.)
DFT. LB (099-000082)	Transfer first when using the Discrete Fourier Transform; otherwise, the tape may be ignored.
CSP. LB (099-000085)	Transfer first when using the FORTRAN Commercial Subroutine Package.
FMT. LB (099-000034)	The FORTRAN IV multitask library (unmapped environment)
MFMT. LB (099-00058)	The FORTRAN IV multitask library (mapped environment)
FORT1. LB (099-000035)	FORTRAN IV Run Time Library 1
FORT2. LB (099-000036)	FORTRAN IV Run Time Library 2
FORT3. LB (099-000037)	FORTRAN IV Run Time Library 3
FORT4. LB (099-000055)	SMPYD. LB - Software multiply/divide
(099-000056)	HMPYD. LB - Hardware multiply/divide (Nova 800's, Nova 1200's Supernova)
(099-000057)	NMPYD. LB - Hardware multiply/divide (Nova)
FSYS. LB (099-000083)	An optional FORTRAN IV library to be used only if certain run time routines are to be utilized in the user's program. The list of the concerned run time routines is shown following.

If any of the following run time routines are to be CALLed from the user's program, the FORTRAN IV library, FSYS. LB, must have been loaded.

BOOT	DUCLK	GCOUT	MDIR	RUCLK
CDIR	DULNK	GDIR	MTDIO	SPDIS
CHLAT	EQUIV	GFREQ	ODIS	SPEBL
CHSTS	EXBG	GSYS	OEBL	SPKIL
CPART	EXFG	GTATR	RDCMN	STAT
DIR	FGND	ICMN	RDOPR	UPDATE
DLINK	GCIN	INIT	RENAME	WRCMN
			RLSE	WROPR

OPERATION UNDER RDOS (Continued)

Once the compiler and library tapes are loaded onto disk, the FORTRAN IV compiler can be invoked using the FORT command followed by appropriate arguments. Unless the user specifies a /A switch (see Compile-Time Options) the compiler expects the save file ASM.SV to be resident on disk.

Each FORTRAN main program, external subroutine, or external function is separately compiled. When the main program and its external subroutines and functions have been successfully compiled, the programs are loaded using the RLDR command. The FORTRAN libraries must always be loaded with the programs.

A series of commands for compiling, loading, and running a FORTRAN program is shown following:

```
FORT MAIN )
FORT XSUB1 )
FORT XFUN )
FORT XSUB2 )

RLDR/D MAIN XSUB1 XFUN XSUB2 FORT1.LB FORT2.LB↑ )
                   FORT3.LB FORT4.LB )

MAIN )
```

Compilation and Assembly

The Command Line Interpreter command FORT is used to compile a FORTRAN IV source program file. The format of the FORT command line is:

FORT { global switches } inputfilename { outputfilename { local switches } })

where: global switches can be appended to the command word, FORT. (These are discussed within the next section.)

inputfilename is the name of the source file the user wishes to be compiled.

outputfilename is an optional file name specifying the name of the file to be output as a result of compilation. (By default, the name of the file to receive the output is inputfilename.)

local switches are optional switches which can be appended to the optional output file name. (These switches are described within the next section.)

By default, the CLI will search for the FORTRAN source file with the specified name inputfilename.FR (or, inputfilename if no file name with the .FR extension is found).

If compilation is successful, an intermediate source file is produced. This file is the output of compilation which is used as the input to assembly. Once the assembly process has been successfully completed, the intermediate source file is deleted. Output from the default form of the command line is a relocatable binary file called inputfilename.RB. Or, if an output file name is specified in the FORT command line, the output file will be a relocatable binary file with the name outputfilename.RB.

Compile-Time Options

Output of compilation may be a relocatable binary file (by default), an intermediate source file, a listing file, or a combination of these files. The type of output received is determined by use of local and global switch options. In addition, switches are used to determine whether or not statements with an X appearing in column 1 are to be compiled and whether FORTRAN variable names and statement numbers are to be equivalenced to symbols acceptable to the assembler.

OPERATION UNDER RDOS (Continued)

Compile-Time Options (Continued)

The global switches which may be appended to the command word FORT are:

/A	-	assembly is suppressed, the source file will only be compiled (and intermediate source file is deleted by default).
/B	-	brief listing (the compiler source program will be the only resultant listing).
/E	-	error messages from the compiler are suppressed at the $TTO. (Assembler error messages, though, are not suppressed.)
/F	-	FORTRAN variable names and statement numbers are equivalenced to symbols which are acceptable to the assembler.
/L	-	the listing will be written to a file named inputfilename. LS .
/N	-	no relocatable binary will be produced.
/P	-	process only 72 characters per record/line (punched card).
/S	-	save the intermediate source output file; by default, this file is deleted.
/U	-	causes user symbols to be output in the assembly phase (must be used with /F).
/X	-	compile statements with an X appearing in column 1. (X indicates an optionally compiled line.)

Local switches are appended to the appropriate outputfilename in the FORT command line. Note that there may be more than one outputfilename within the command line. The local switches are:

/B	-	the relocatable binary output is directed to outputfilename. This switch overrides the global /N switch.
/E	-	the resultant error messages are directed to outputfilename. The local /E switch overrides the global /E switch.
/L	-	listing output is directed to outputfilename. This switch overrides the global /L switch.
/S	-	intermediate source output is directed to outputfilename.

Some examples of FORT command lines are:

FORT/L PROG)

produces a relocatable binary file with the name PROG. RB, and a compiler and assembler listing written to the file PROG. LS .

FORT/N DP1:PROG1 $LPT/L APROG1/S)

compiles the file PROG1 from disk pack unit 1 and produces compiler source and assembly listings on the line printer and intermediate source output file, named APROG1, to the default directory. This command line will not produce a relocatable binary file from the assembly.

Loading Procedures

All loading is accomplished via the RLDR command line. The format of the command line, though, is different depending upon the particular RDOS system configuration and environment. Procedures are outlined on following pages for loading in:

OPERATION UNDER RDOS (Continued)

Loading Procedures (Continued)

a single task environment
a multiple task environment
overlays, creating an overlay file
a foreground/background environment
a mapped/unmapped environment

Global and local switches may be appended to the command word or file name where pertinent. These switches are listed under the procedures for loading in a single task environment, but they should be remembered and referred to when reading the other loading procedures detailed within this section.

In general, loading proceeds as follows:

Main FORTRAN program
User subprograms and optional user modules such as FHMA and FRTSK.
Specific, optional, DGC supplied FORTRAN libraries such as RTIOS. LB (Real Time I/O System) or DFT. LB (Discrete Fourier Transform).
Required FORTRAN Libraries in the order given in sections following (FMT. LB, FORT1. LB, etc.).

Loading in a Single Task Environment

The RLDR command line is used to load relocatable binary output produced from compilation. The format used for loading in a single task environment is:

```
RLDR [global switches]  mainprogram [local switches] [subprograms] [local switches] ↑)
     [ FHMA]  FORT1. LB [FSYS. LB]  FORT2. LB  FORT3. LB  FORT4. LB )
```

where: global switches which can be appended to the command word RLDR are:

/A - produce an additional symbol table listing with symbols ordered alphabetically.
/C - cause loading to be compatible with RTOS/SOS conventions.
/D - load the symbolic debugger.
/E - output errors to the error file (console, by default).
/H - output all numerics in hexadecimal format (radix 16). By default, all numeric output is in octal format.
/N - inhibit search of SYS. LB.
/S - symbol table left at the high end of memory.
/Z - start save file at location zero. (CAUTION must be exercised if this switch is used.)

mainprogram is the name of the FORTRAN IV main program unit.

local switches are switches which may be appended to an input file name or octal number, these are:

/C - preceding octal number specifies number of channels required.
/E - error messages are output to given file name.
/F - preceding octal value is the foreground NREL partition address (used only when loading in foreground/background unmapped environment).

Loading in a Single Task Environment (Continued)

/K	preceding octal value specifies the number of tasks required. (This switch is not used in a single task environment).
/L	listing of the symbol table is written to the given file name.
/N	NMAX is forced to an absolute address.
/S	file specified will be labeled with the .SV extension.
/U	user symbols are loaded from relocatable binary file specified.
/Z	preceding octal value is the foreground ZREL partition address. (Used only in a foreground/background loading environment.)

subprograms are the optional names of one or more FORTRAN subprograms to be used by the main FORTRAN program unit.

FHMA is an optional user module which defines the highest memory address accessible. The default value is at the bottom of the system.

FORT1. LB, FORT2. LB and FORT3. LB are three FORTRAN IV Run Time Libraries which must be loaded. FORT4. LB is one of three tapes depending upon the system configuration, either:

099-000056	hardware multiply/divide (HMPYD. LB) (used with Nova 800's, Nova 1200's and the Supernova)
099-000057	hardware multiply/divide (NMPYD. LB) (used with the Nova)
099-000055	software multiply/divide (SMPYD. LB)

FSYS. LB is a FORTRAN library which is loaded only if one or more of the CALLs listed on page D-1 are to be issued from within the user program.

The main program is always loaded first, followed by any external subprograms, followed by the FORTRAN IV library files.

Loading in a Multitask Environment

In a multitasking environment, the multitask library (called FMT. LB) must be loaded before any of the other FORTRAN libraries. The switches and comments which applied to single task loading apply also to multitask loading. The format of the RLDR command line used in a multitask environment is:

```
RLDR main [{ taskname          ... }] [number/C] [number/K] [FHMA] [FRTSK] FMT.LB ↑)
          [{ [overlay-area]    ... }]
     FORT1.LB [FSYS.LB] FORT2.LB FORT3.LB FORT4.LB )
```

where: main is the name of the FORTRAN main program unit.

taskname is the name of a relocatable binary compiled from a task written in FORTRAN IV or assembled from an assembly language program.

overlay-area is a bracketed list of relocatable binaries to become part of the overlay file, main. OL; relocatable binaries separated by blanks are part of the same node while those separated from the preceding by a comma belong to another node.

Loading in a Multitask Environment (Continued)

number/C specifies the number of system channels required.

number/K specifies the number of tasks to be used. (The numbers specified by the C and K switches overwrite the values specified in the CHANTASK statement. If these values are unspecified, the default values are a single task environment with 8 channels required.)

FHMA is an optional user module which defines the highest memory address accessible. The default value is at the bottom of the system.

FRTSK is an optional user-supplied module specifying the number of tasks written in FORTRAN which will be active simultaneously.

FMT. LB is the name of the FORTRAN IV multitask library.

FORT1. LB, FORT2. LB and FORT3. LB are three FORTRAN IV Run Time Libraries which must be loaded. FORT4. LB is one of three tapes depending upon the system configuration, either:

099-000056	hardware multiply/divide (HMPYD. LB) (used with Nova 800's, Nova 1200's and the Supernova)
099-000057	hardware multiply/divide (NMPYD. LB) (used with the Nova)
099-000055	software multiply/divide (SMPYD. LB)

FSYS. LB is a FORTRAN library which is loaded only if one or more of the CALLs listed on page D-2 are to be issued from within the user's program.

The Overlay Loader (OVLDR)

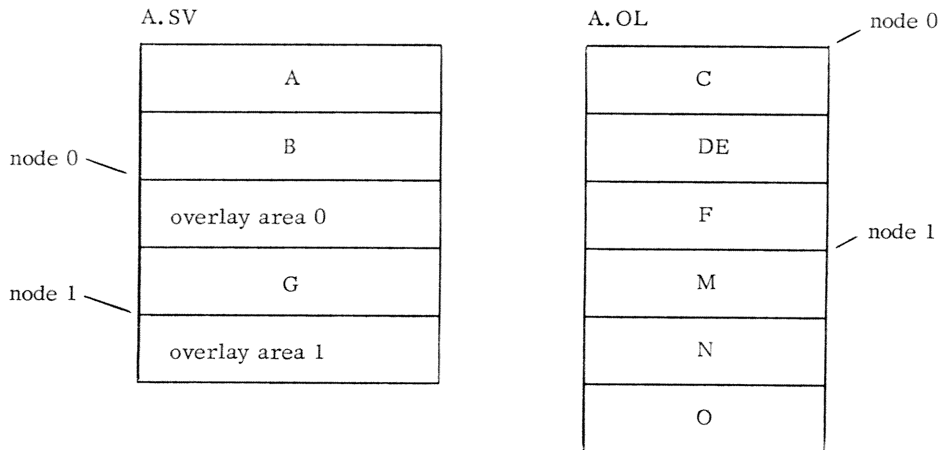
It is possible to replace one or more overlays within an overlay file. To do so, a file of replacement overlays must be loaded using the overlay loader, which is invoked with the command OVLDR. When the replacement file of overlays has been loaded, overlays within the current overlay file may be replaced with overlays in the replacement file, using the command REPLACE. Up to 127 overlays can be replaced.

Use of the overlay loader requires that there exist a save file filename. SV and an overlay file filename. OL, and that the save file contain a symbol table. (The save file will contain a symbol table either if the symbolic debugger is loaded (global/D in the RLDR command line) or if the symbol table only is loaded by declaring it as an external normal, . EXTN .SYM., in the code loader is part of the save file. For example, save file A. SV and overlay file A. OL could be loaded using the following command line:

```
RLDR/D  A  B  [C, D E,  F] G [M,  N,  O] FORT. LB )
```

The diagrams on the following page would represent the save and overlay files created.

The Overlay Loader (OVLDR) (Continued)



Then if one or more overlays of A. OL are to be replaced at a later time, the overlay loader can be used to load a replacement overlay file, A. OR. The format of the OVLDR command line is:

OVLDR filename { overlay symbol/N | overlay number/N } overlay list ... ↑

[{ overlay symbol/N | overlay number/N } overlay list] [devicename/L] ↑

[devicename/E])

where: filename is the name of the save file associated with the overlay file in which overlays are to be replaced.

overlay symbol and overlay number are alternative means of referencing the overlay(s) to be replaced. overlay number is a 1 to 6 digit octal number (see page 5-8). overlay symbol, if used, must have been a symbol declared by . ENTO .

overlay list is a list of one or more overlays which are to replace the overlay specified by overlay symbol or overlay number.

devicename/L is the name of the device to contain the listing file.

devicename/E is the name of the device to contain the error file.

For example, if the user wishes at some time to replace overlay F in A. OL with overlay F1 and to replace overlay O in A. OL with overlay O1, he must first load the overlays into a replacement file using the overlay loader:

```
OVLDR  A  2/N  F1  402/N  O1  $LPT/L  $TTO/E )
```

The Overlay Loader (OVLDR) (Continued)

The resulting overlay replacement file would contain:

A. OR

.OR directory
F1
O1

To substitute F1 for F and O1 for O, the user would then give the command:

REPLACE A)

Loading in a Foreground/Unmapped Environment

All relocatable loads are loads in the background unless indicated as foreground loads. This is indicated by including memory partition address information within the command line. The partition addresses define the starting ZREL (/Z switch) and NREL (/F switch) addresses of the foreground load.

The selected NREL partition address must be equal to $16_8 + \underline{n} * 400_8$ where n is a positive integer. If the given NREL partition address is not of this form (i.e., n is not an integer), the loader will adjust the NREL partition address upwards by rounding n to the next higher integer value.

The format of the RLDR command line is:

```
RLDR main [{taskname ...}] [FRTSK] [number/C] [number/K] ↑)
          [{[overlay-area] ...}]
     number/F number/Z FMT.LB FORT1.LB [FSYS.LB] FORT2.LB ↑)
     FORT3.LB FORT4.LB )
```

where: the command line is identical to previous command line formats except for the inclusion of the /F and /Z switches.

Loading in a Background Unmapped Environment

The RLDR command line format used is identical to that given for the multitask loading environment, except that an optional user-supplied module (called FHMA) may be loaded anywhere before the libraries in the command line.

FHMA specifies the highest memory address a FORTRAN background program may have. This insures that memory will be available for a potential foreground program.

Examples of RLDR Command Lines

RLDR MAIN SUB1 FORT1.LB FORT2.LB FORT3.LB FORT4.LB)

loads one main program unit and one external subprogram unit and the FORTRAN libraries.

Examples of RLDR Command Lines (Continued)

RLDR MAIN TASK1 TASK2 FMT.LB FORT1.LB FORT2.LB FORT3.LB FORT4.LB)

loads one main program unit and two tasks along with the multitask library and the other four FORTRAN libraries.

RLDR MAIN T2 [OV1 OV2, OV3] T3 FRTSK FMT.LB 10/C 4/K FORT1.LB↑)
FORT2.LB FORT3.LB FORT4.LB)

loads a FORTRAN main program unit, two tasks, and an overlay file consisting of two overlays, the user-supplied FRTSK module, the multitask library, and the four FORTRAN libraries. The number of tasks is set to 4 and the number of channels to 10.

Undefined Symbols

At the termination of loading, only the .DSI symbol (which is used in stand-alone) should be undefined. When in a multitasking environment, no symbol should remain undefined in the load map.

To provide dummy definition on BATCH runs, .DSI can be defined as:

.DSI = -1

Debugging

To use the symbolic debugger, DEBUG III, for run time debugging of FORTRAN programs, the global /D switch should be appended to the RLDR command word. The switch causes DEBUG to be loaded. To replace DEBUG III with IDEB, the RLDR command line must also contain the file name IDEB in addition to the /D global switch. The mnemonic IDEB must precede SYS.LB in the command line if SYS.LB is present in the line.

Merging Library Files

The merging of library files can be accomplished by use of the M function within the LFE command line. The library files FORT1.LB, FORT2.LB, FORT3.LB and FORT4.LB can then be merged together as one library file. This file should be named FORT.LB to be recognized by CLG (see The CLG Command on page D-10). The command to merge the FORTRAN libraries is:

LFE M FORT.LB/O FORT1.LB FORT2.LB FORT3.LB FORT4.LB)

where the local /O switch signifies the name of the output library file, FORT.LB.

It is then possible to load a FORTRAN program, with necessary libraries, with the following command line:

RLDR MAIN FORT.LB)

The FORTRAN multitask library, FMT.LB, and the FORTRAN Mapping Library, MFMT.LB, are merged only with the other library files when multitasking in mapped or unmapped facilities is desired.

The FORTRAN IV library FSYS.LB can be merged when it is necessary for its inclusion in the set of libraries, because of a program's issuance of CALLS to one or more of the routines listed on page D-1.

Merging Library Files (Continued)

To load a FORTRAN program, with two subprograms, in a multitask environment, the following command line can be given:

RLDR MAIN TASK1 TASK2 FMT. LB FORT. LB ↩

The CLG Command

The CLG Command is used to perform a FORTRAN IV compilation, load, and execution of one or more FORTRAN IV source files. The CLG (compile, load, and go) will bring in whatever system programs are required to create a save file from the specified input files and then execute the save file just created.

Output includes one or more intermediate source files, one or more relocatable binaries, and an executable save file. The save file is created by the relocatable loader, using the relocatable binary files and four of the FORTRAN IV libraries which must have been merged into a single library called FORT. LB.

CLG is supplied as a file on the FORTRAN system dump tape, which is loaded as part of the FORTRAN compiler. All other needed system programs must have been loaded onto disk. These include, besides the FORTRAN IV compiler, the Assembler and the Relocatable Loader.

The format of the CLG command line is:

CLG filename [filename2 . . . $filename_n$] ↩

By default, all filename arguments are presumed to be file names of FORTRAN IV source files. Optional load switches to the particular file names specified indicate whether the file is an assembly source file or an assembled file that is to be loaded. The CLI will first search for filename. extension (where extension is either . FR, . SR or . RB) and if not found, will search for filename.

Global switches can be appended to the command word CLG. The allowable global switches are:

/B	brief listing (compiler source program input only).
/M	the loader map is suppressed. All compiler and assembler source programs are listed.
/E	error messages from the compiler are suppressed at the $TTO. (Assembler error messages, though, are not suppressed.)
/T	indicates multitask CLG command line. (Note that the multitask library FMT. LB must be available on disk.)

Local switches are appended to the appropriate filename within the CLG command line. Allowable local switches are:

/C	preceding octal number specifies number of channels required.
/L	listing output is directed to the given file name.
/A	assemble and load this file only; do not compile.
/N	load this file only, do not compile or assemble.
/K	preceding octal value specifies number of channels required.

An example of a CLG command line is:

CLG/M PROG1 PROG2/A PROG3/N MT0:1/L ↩

OPERATION UNDER RDOS (Continued)

The CLG Command (Continued)

In the example, CLG will take the following action:

1. Compile PROG1. FR (or PROG1) producing temporary assembler source file, PROG1.SR. Assemble PROG1.SR producing PROG1. RB. Delete PROG1. SR.

2. Assemble PROG2. SR (or PROG2) producing PROG2. RB. Delete PROG2.SR.

3. Listings from each compilation and assembly are appended to file 1 on magnetic tape unit 0.

4. Load PROG1. RB, PROG2. RB, and PROG3. RB together with the FORTRAN IV library file FORT. LB, to a save file named PROG1. SV. The loader map is suppressed.

5. Execute PROG1. SV.

In a single task environment, the FORTRAN merged library file, FORT. LB, must be on disk when the CLG is executed. For multitasking, both the multitasking library, MFMT. LB, and the merged file FORT. LB must be on disk.

OPERATION UNDER RTOS

Since the Real Time Operating System (RTOS) is a compatible subset of the Real Time Disk Operating System (RDOS), RTOS will support a subset of DGC Real Time FORTRAN IV. To write a FORTRAN IV program for use with RTOS, you may use either the RDOS FORTRAN IV compiler or the SOS FORTRAN IV compiler. Operating procedures for each are given within this chapter, beginning on page D-1 for RDOS and page D-14 for SOS.

The only restriction when writing a FORTRAN IV program under RTOS is that only those real time calls may be used which have corresponding system and task calls implemented in RTOS. Therefore, use of the OVERLAY statement is prohibited, but all Real Time FORTRAN IV calls except the following may be issued. The calls which are prohibited in a RTOS environment are:

BACK	CPART	FBACK	FSTAT	ODIS	RDCMN	WRCMN
BOOT	DFILW	FCHAN	FSWAP	OEBL	RDOPR	WRITR
CDIR	DIR	FCLOS	GCIN	OVEXT	RDRW	WROPR
CFILW	DLINK	FGND	GCOUT	OVEXX	READR	WRTR
CHAIN	DULNK	FOVLD	GDIR	OVKIL	RENAM	
CHLAT	EBACK	FOVRL	GSYS	OVKIX	STAT	
CHSAV	EXBG	FQTASK	ICMN	OVLOD	SWAP	
CHSTS	EXFG	FSEEK	MDIR	OVOPN	UPDATE	

After having produced one or more FORTRAN IV relocatable binaries, the relocatable binaries may be loaded using the (1) SOS relocatable loader (RLDR), the (2) RDOS relocatable loader (RLDR), or the (3) stand-alone extended relocatable loader (091-000038).

Using (1) or (2) the format of the command line is:

```
RLDR main [subprograms] RT module [RTOSFMT.LB] FORT1.LB [FSYS.LB] FORT2.LB
     FORT3.LB  FORT4.LB  RTOSGEN.SV  RTOSGEN.RB  GSUBR.RB  PARR.SR  RTOSGEN.AB
     RTOS1.LB  RTOS2.LB  CASDR.LB  MTADR.LB  DSKDR.LB  DKPDR.LB )

RLDR main [subprograms] RT module [RTOSFMT.LB] FORT1.LB [FSYS.LB] FORT2.LB↑)
     FORT3.LB  FORT4.LB  RTOS1.LB  RTOS2.LB )
```

where: main is the name of the main FORTRAN program unit.

subprograms are the names of one or more optional subprogram units called by main.

RT module is the name of the module produced in the RTOS SYSGEN procedure.

RTOSFMT.LB (099-000077) is the RTOS multitasking FORTRAN IV run time library. The library must not be used in a single tasking environment.

FORT1.LB, FORT2.LB, FORT3.LB, and FORT4.LB are the FORTRAN libraries. (These libraries may exist on paper tape, magnetic tape, or cassette tape.) Note that these are the RDOS libraries described on page D-1.

FSYS.LB (099-000083) is a library that is to be loaded if the user's program is to issue one of the following run time calls: DUCLK, GFREQ, INIT, MTDIO, RLSE, and RUCLK. (If this library is to be loaded, it must be loaded between FORT1.LB and FORT2.LB.)

RTOS1.LB (099-000060) is the first RTOS library to be loaded, and RTOS2.LB (099-000061) is the second RTOS library to be loaded.

If using the (3) stand-alone relocatable loader (091-000038) which is loaded via the binary loader, it will self-start and print:

SAFE =

OPERATION UNDER RTOS (Continued)

after which the user responds with a carriage return which will reserve the upper 200 words of memory, preserving both the bootstrap and binary loaders. The loader then prompts:

*

The paper tapes may then be loaded in the same order as they were typed in response to an RLDR command line format. Briefly, the load process is to mount each tape, in turn, in either the teletypewriter reader or the high-speed paper tape reader, then type either 1 or 2.

1 - teletypewriter reader
2 - high speed paper tape reader

After each tape is loaded, the loader prompts with *. After the pertinent tapes have been loaded a loader map can be requested by typing 6. At this time the load process can be terminated by typing 8.

To restart, set the restart address, 376, in the data switches, press RESET, and then press START.

For more detailed instructions of operation in an RTOS environment, refer to the RTOS User's Manual, 093-000056, Appendix B.

OPERATION UNDER SOS

SOS operating procedures are subdivided into three segments (1) those for users whose SOS system does not support a cassette or magnetic tape unit (SOS library tape 099-000010), (2) those for users whose SOS system includes a magnetic tape or cassette unit (SOS library tapes 099-000010 and either 099-000042 (magnetic tape) or 099-000041 (cassette)), and (3) those for users whose SOS system is DOS-compatible (SOS library tape 099-000071). Operation for numbers (1) and (2) is given below and on the following pages, operation for (3) begins on page D-21.

OPERATION UNDER RDOS-COMPATIBLE SOS

This SOS version of the FORTRAN IV compiler is supplied as two absolute binaries, FORT1.AB (091-000039) and FORT2.AB (091-000043) and four libraries, FORT1, FORT2, FORT3, and FORT4, which are the RDOS libraries described on page D-1.

For users whose system supports a cassette or a magnetic tape unit, two relocatable tapes are provided which allow users to configure their own FORTRAN IV compiler to be used with their specialized peripheral devices. These tapes are:

SOSFI. RB	-	089-000041
FORT. RB	-	089-000161

The SOS relocatable loader can be used to configure a specialized version of the FORTRAN compiler. The trigger used may be generated by the SOS SYSGEN program or a separate assembly may be produced to generate external normal symbols which will trigger the loading of SOS device drivers. For example, if the user wants a compiler with a high speed paper tape reader, a high speed paper tape punch, and two cassette units, he could input to the SYSGEN program:

(SYSG) trigger/T $PTP/O .PTRD .PTPD .CTU1 ⏎

The SOS relocatable loader can then be used to load the following files in the order:

TRIG
SOSCT
SOS. LB
FORT. RB
SOSFI. RB

Compilation

When SOS FORTRAN IV is loaded, the prompt:

FORT

is printed on the teletypewriter. The user should respond by typing in a command line giving the file names of the files to be input for compilation, the output file name, and the listing file name if any, along with optional compile-time option switches. The FORT command line will be written in the following format:

FORT filename1 [filename2 ... filenamen] ⏎

where: FORT followed by a space is typed by the system.

each filename can be modified by one or more of the switches described on the following page.

OPERATION UNDER RDOS-COMPATIBLE SOS (Continued)

Compilation (Continued)

The switches which may be appended to a given filename are:

/O	-	this file is to be used for output.
/L	-	this file is to be used as the listing file.
/X	-	compile statements with an X appearing in column 1 of the source line. (This must be used to modify the output file name, the file name which is appended with the /O switch.
/S	-	FORTRAN IV variables and statement numbers are equivalenced to symbols which are acceptable to the assembler. (This switch must modify the output file name).
/n	-	n is a single digit representing the number of files to be input, e.g., $PTR/3.

At a minimum, the command line must contain one file name which is the input file name. If more than one input file is specified, e.g.,

FORT MT0:2 MT0:3 MT1:0/O)

the message:

TO CONTINUE, STRIKE ANY KEY)

is typed on the teletypewriter console whenever one of the intermediate files has passed through the compiler. The next input file must be ready for opening when the user strikes the key. No other prompt messages are output for intermediate input files.

Input files are compiled in the order in which they are specified within the command line. At the completion of each compilation, the prompt:

FORT

is again typed on the teletypewriter. The prompt is reissued if no input file name is found in the command line. If the last specified input file does not have an END statement, the message:

END OF FILE)

is typed at the console. The compiler must then be restarted (the restart location is 377). If any unexpected system error occurs, the message:

FATAL I/O ERROR xx)

is typed at the console. xx is one of the two-digit error codes defined in the SOS User Parameter Tape, PARU.SR (a copy of which can be found in the Stand-alone Operating System User's Manual, 093-000062).

The command line may be deleted, continued, or modified in the following manner:

1. Pressing SHIFT and L keys will delete the line.
2. An up arrow immediately preceding a carriage return (line feed) allows the command line to be continued onto the next console line.
3. Pressing RUBOUT erases the last character typed in the command line. Repeated RUBOUTs delete characters from right to left.

Compilation (Continued)

An example of a FORT command line is:

```
FORT $PTR/2 $PTP/O $LPT/L )
```

The SOS SYSGEN procedures allow the user to tailor the compiler I/O configuration. See the SOS User's Manual.

Assembly

FORTRAN IV output is assembled with the DGC Extended Assembler, 091-000017. The assembler can be loaded from paper tape, at which point it will print the prompt ASM. Or, when a cassette or magnetic tape unit is configured in the system, the CLI command ASM may be issued. In either case, the format of the ASM command line is:

```
    ⎧ 0 ⎫
ASM ⎨ 1 ⎬  filename1 [...filenamen] )
    ⎩ 2 ⎭
```

The ASM command line is used to assemble one or more ASCII source files. Output may be an absolute binary file or a relocatable binary file. Files are assembled in the order specified in the command line, left to right. The same cassette or magnetic tape unit cannot be used for more than one output file but may be used for more than one input file. Further, the same cassette or magnetic tape unit cannot be used for both input and output.

Action taken by the assembler is determined by the key specified in the ASM command line (0, 1, or 2).

0 - Perform pass one on the specified source file, then halt with the highest symbol table address (SST) in AC0.

1 - Perform pass one and pass two on the specified FORTRAN input files, producing the specified binary and listing files. At the completion of pass two, the assembler outputs a new prompt, ASM, and awaits a new command line.

2 - Perform pass two only on the specified input files producing the specified binary and listing files. At the completion of this pass, the Assembler outputs a new prompt, then ASM, and awaits a new command line.

The global switches which may be appended to the key number are:

/E - suppress assembly error messages normally output to the $TTO.
/T - suppress the listing of the symbol table.
/U - include local (user) symbols in the binary output file.

The local switches which may be appended to individual file names are:

/B - relocatable or absolute binary file is output on the given device.

/L - any output device to which the listing is directed.

/N - any input file which is not to be listed in pass 2.

/P - pause before accepting a file from a device. The message:

Assembly (Continued)

PAUSE - NEXT FILE, devicename

is output by the assembler which waits until any key is struck on the teletypewriter console.

/S - skip this source file during pass two.

/n - n is a digit from 2 to 9.

ASM 1/E CT0:16 CT0:17 CT1:0 CT1:1 $LPT/L)

causes a two-pass assembly to be executed on FORTRAN input source file CT0:16, CT0:17, CT1:0, and CT1:1 with a listing produced on the line printer. Error messages normally output to the $TTO are suppressed, and no binary file is produced.

Loading

Having produced one or more FORTRAN IV relocatable binaries, the relocatable binaries for systems using magnetic tape or cassette may be loaded using the SOS relocatable loader (089-000120). Systems using paper tape may be loaded using the stand-alone relocatable loader (091-000038). The SOS relocatable loader prints the prompt RLDR, and the user responds with the command line:

(RLDR) main [subprograms] FORT1. LB FORT2. LB ↑)

FORT3. LB FORT4. LB trigger [{cassette library / mag tape library}] SOS main library)

where: main is the name of the FORTRAN main program unit.

subprograms are the names of one or more optional subprograms to be called by main.

FORT1. LB, FORT2. LB, FORT3. LB are FORTRAN libraries. (These libraries may reside on paper tape, magnetic tape, or cassette tape.)

FORT4. LB is a FORTRAN library tape selected to correspond to the user's system configuration, either:

099-000056	hardware multiply/divide (Nova 1200;s, Nova 800's Supernova)
099-000057	hardware multiply/divide (Nova)
099-000055	software multiply/divide

trigger is the SOS trigger which is created during the SOS SYSGEN procedures. It is a tape containing external symbols for those devices that are to be a part of the system. (trigger is outlined on the following page.)

cassette library is tape number 099-000041 and must be loaded only when cassette units are to be a part of the system.

mag tape library is tape number 099-000042 and must be loaded only when magnetic tape units are to be a part of the system.

SOS main library is tape number 099-000010 and it contains the main library and all driver routines for SOS I/O devices (except cassette and magnetic tape units).

OPERATION UNDER RDOS-COMPATIBLE SOS (Continued)

Loading (Continued)

Upon completion of a successful load, the message OK is printed at the console and the system will halt with the loaded program in core.

The stand-alone version of the relocatable loader (091-000038) is used as described under RTOS, page D-12. Loading of FORTRAN tapes proceeds in the same order as given for the SOS loader just described.

Execution and Restart Procedures

The loaded program may be executed by pressing CONTINUE or by using the RESTART procedures. When a PAUSE statement is executed, the program will continue when the programmer presses any teletypewriter key. Restart procedures are as follows:

1. Set switches to 377.
2. Press RESET.
3. Press START.

Producing a Trigger

A trigger is produced using the SOS SYSGEN program which is loaded via the binary loader, or loaded using the core image loader/writer. Basically, the SYSGEN program accepts a command line containing device driver entry symbols and outputs a file containing external references to the named devices. When the trigger is loaded in the RLDR command line (preceding other SOS libraries) the external normal references on the trigger will cause the named device drivers to be loaded from the SOS libraries. The format of the SYSGEN command line is:

(SYSG) $\underline{driver}_1$,...$\underline{driver}_n$ .RDSI [.CTB] output-device/O [triggername/T]

driver may be one or more device driver entry symbols selected from the following chart:

Device Name	Device Driver Entry Symbol	Device
$CDR	.CDRD	card reader
CT0	.CTAD	cassette unit 0
CT0,1	.CTU1	cassette units 0 and 1
CT0,1,2	.CTU2	cassette units 0,1, and 2
⋮	⋮	⋮
CT0,1,2,3,4,5,6,7	.CTU7	cassette units 0,1,2,3,4,5,6 and 7
$PTP	.PTPD	high-speed paper tape punch
$PTR	.PTRD	high-speed paper tape reader
$LPT	.LPTD	80-column line printer
	.L132	132-column line printer
MT0	.MTAD	magnetic tape unit 0
MT0,1	.MTU1	magnetic tape units 0 and 1
⋮	⋮	⋮
MT0,1,2,3,4,5,6,7	.MTU7	magnetic tape units 0,1,2,3,4,5,6 and 7
$PLT	.PLTD	incremental plotter
$TTO/$TTI	.STTY	teletype printer and keyboard
TTI1/TTO1	.TTI1	second teletype printer and keyboard
	.RTC1	real time clock, 10HZ
	.RTC2	real time clock, 100HZ
	.RTC3	real time clock, 1000HZ
	.RTC4	real time clock, 60HZ
	.RTC5	real time clock, 50HZ

For more detailed instructions for producing a trigger for SOS systems, refer to the Stand-alone Operating System User's Manual, 093-000062.

OPERATION UNDER RDOS-COMPATIBLE SOS (Continued)

Possible Error Messages

The possible error messages resulting from the ASM or RLDR command lines are:

Error Message	Meaning	ASM	RLDR
NO END	No END statement was specified in any source program.	X	
NO INPUT FILE SPECIFIED	No input file name was specified.		X
SAVE FILE IS READ/ WRITE PROTECTED	The save file device must permit both reading and writing: only cassette and magnetic tape units are permitted as save file devices.		X
I/O ERROR n	Input/output error n where n = 1 Illegal file name. 7 Attempt to read a read-protected file. 10 Write-protected file. 12 Non-existent file.	X X X X X	X X X X X

SOS FORTRAN IV Examples

```
FORT CT0:0 $LPT/L CT1:0/O)

FORT $LPT/L CT0:1 CT1:1/O)

FORT CT1:2/O CT0:2 $LPT/L)
```

FORTRAN IV input files on CT0:0, CT0:1, and CT0:2 are compiled and assembly source files are produced on CT1:0, CT1:1, and CT1:2 (indicated by a /O switch) respectively with all listings produced on the line printer.

```
ASM 1 $LPT/L CT0:0/B CT1:0)

ASM 1 CT0:1/B CT1:1 $LPT/L)

ASM 1 CT1:2 $LPT/L CT0:2/B)
```

Assembly source files on CT1:0, CT1:1, and CT1:2 are assembled and relocatable binary files are produced with a listing to the line printer.

```
(SYSG) TRIG/T CT1:0/O .RDSI.CTU2 .PTRD .PTPD)
```

A trigger file is produced on CT1:0 with external normal references necessary to load drivers for 3 cassette units, $PTR driver, $PTP driver, and the RDOS-to-SOS interface from the SOS libraries.

```
RLDR $LPT/L CT2:0/S CT0:0 CT0:1 CT0:2/P CT1:0/P CT1:0↑)

      CT1:0 CT1:1 CT1:2)
```

SOS FORTRAN IV Examples (Continued)

A save file is produced on CT2:0 and also loaded into core with a load map printed on the $LPT. This command line assumes that the following procedure is executed:

1) The relocatable binaries generated from the FORTRAN compilations are loaded from CT0:0, CT0:1, and CT0:2 with a pause (indicated by the /P switch) following the last one loaded.

2) A merged version of the four FORTRAN libraries on a single cassette reel is mounted on unit 1 and loaded with a pause following this file.

3) The reel on unit one is now replaced with a reel which contains the trigger file, the SOS cassette library file, and the SOS library file. These files are then added to the load module.

OPERATION UNDER DOS-COMPATIBLE SOS (STAND-ALONE FORTRAN)

Using the binary loader, FORTRAN IV program tapes 091-000039 and 091-000043 are loaded in that order. Restart location, if needed, is at location 377.

Compile-Time Options

When FORTRAN IV is loaded, the system queries the user in regard to device assignments and compile-time options as follows:

IN:

The user responds to this query with a single number representing the source code input device as follows:

```
1  -  $TTI
2  -  $TTR }
3  -  $PTR }          ASCII output must have even parity.
4  -  $CDR
```

When the source code input device has been given, the system queries:

OUT:

The user is expected to respond to the query with a number representing the assembler source output device (including error listing). The possible responses are:

```
1  -  $TTO
2  -  $TTP
3  -  $PTP
4  -  $LPT
0  -  no device                     ; used when only a listing is desired
```

After the user's response is complete, the system then queries:

LIST:

The user responds with a number indicating the designated listing device. The possible responses are:

```
0  -  no device (no listing desired)
1  -  $TTO
2  -  $TTP
3  -  $PTP
4  -  $LPT
```

The listing includes the FORTRAN source program complete with error messages. All lines of this output listing are preceded by a semicolon in order that the OUT and LIST devices may be the same. It is important to note that error messages are always output to the teletypewriter regardless of whether a LIST or OUT device was specified or not. The system will then query:

COMPILE X ?

requesting the user to specify whether source lines preceded by an X in column 1 should be compiled. A response of 1 will compile the lines; a response of 0 will cause the assembler to treat the lines as comments. The system then queries:

SYMBOLS ?

Compile Time Options (Continued)

where the user responds with a number indicating whether symbols should be equivalenced or not. Possible responses are:

0 - suppresses the symbol list
1 - all FORTRAN variables and statement numbers will be equivalenced as discussed in Appendix C. The symbols will be output.

If the user issues an illegal response to SYMBOLS, the query will be repeated ignoring the illegal response.

After the user has responded successfully to the queries, the source program can be input for compilation from the designated input device. When compilation is complete (as determined by an END statement in the source program), the FORTRAN compiler will type:

TO CONTINUE, STRIKE ANY KEY

To compile another program, using the same designated input and output devices, press any key on the teletypewriter keyboard. To change the device assignments, restart at location 377.

An option is open to the programmer to input his source program from several separate tapes. To do so, each tape must end with the line:

.EOT

The .EOT line will then allow for separate tapes in parameter definition and COMMON declarations.

Assembly

FORTRAN IV output can be assembled with the DGC Extended Assembler. Each FORTRAN IV program generated is complete with all necessary declarations and pseudo-ops in order to use the assembler. (There are many errors which may be ignored by the compiler but detected by the assembler, particularly, usage of assembler reserved mnemonics; therefore, do not suppress error typeout.) The binary output resulting from the assembled mode of operation 2 or 4 is relocatable.

Loading

To run under DOS-compatible SOS, the binary tapes for the .MAIN FORTRAN program and all subprograms can be loaded using DGC's Extended Relocatable Loader. This loader is described in Chapter 1 of manual number 093-000080.

Loading should proceed as outlined below:

1. Load the FORTRAN main program relocatable binary.
2. Load all FORTRAN subprograms which are called by the main program.
3. Load FORTRAN library tape number 099-000005.
4. Load FORTRAN library tape 099-000006.
5. Load FORTRAN library tape 099-000007.

Loading (Continued)

6. Then load one of the following tapes:

099-000009	if system is configured with multiply/divide hardware option (Nova 1200's, Nova 800's, Supernova)
099-000011	if system is configured with multiply/divide hardware option (Nova)
099-000008	no multiply/divide hardware option

7. Load the DOS-compatible SOS library tape (DOS tape no. 099-000071).

8. A loader map can be obtained at this time with load mode 6.

9. Check undefined symbols (load mode 9). Undefined symbols will be listed at the teletypewriter. The following may reasonably be undefined.

FLSZ	Number stack size; if undefined a default value is used.
FLSP	Real arithmetic package.
CMSP.	Complex arithmetic package.

10. Terminate the load (mode 8). The loaded system may be run by pressing CONTINUE or by using the restart procedure.

Restart Procedure

The loaded system may be restarted by:

1. Set switches to 377
2. Press RESET
3. Press START

Execution

When a PAUSE statement is executed, the program will continue when the programmer presses any teletypewriter key.

OPERATION OF 8K STAND-ALONE FORTRAN IV

The FORTRAN IV program tape 091-000052 is loaded with the binary loader. The library tapes for 8K FORTRAN are the same as for DOS-compatible SOS, (in addition to library FORT0. LB).

Operating Procedures

Operating procedures for the 8K version of FORTRAN IV differ from those of the DOS/SOS FORTRAN IV version at compile time. These differences are:

1. There is no separate LIST output.

2. Responce to the "IN:" query is one of the following:

   2 Teletypewriter reader
   3 Paper tape reader

3. Response to the "OUT:" query is one of the following:

   0 None
   2 Teletypewriter punch
   3 Paper tape punch

4. There is no "SYMBOLS" query and no symbol list output.

5. The . EOT tape option is not available.

6. After the query "COMPILE X?" has been answered by either a 1 or a 0 the compiler will type:

   LD RDR HIT CR

   The programmer should prepare the tape in the appropriate reader and type a carriage return (press the RETURN key).

7. At the end of the compilation, the compiler will reinitialize and type again:

   IN:

8. To restart the compiler after shutdown or to change initial assignments, start at location 377.

9. The object code produced by the 8K compiler may need library FORT0. LB to be loaded before the others.

OPERATION OF 8K STAND-ALONE FORTRAN IV (Continued)

Language Limitations

The following features of DGC FORTRAN IV are not available for an 8K memory configuration:

1. DATA statement
2. Statement functions
3. EXTERNAL statement
4. Mixed mode arithmetic
5. File positioning
6. EQUIVALENCE statement
7. FORMAT syntax checking at compile time
8. Library function argument count and type checking
9. Complex literals
10. PARAMETER statement

Most of the features of the larger FORTRAN IV, though, can be effectively replaced by combinations of other FORTRAN IV statements, for example:

> Statement functions are identical to FUNCTION subprograms in calling sequence and code generated.
>
> The two major functions of an EQUIVALENCE statement are to equate logical and numerical storage and to share temporary storage. DGC FORTRAN IV automatically equates logical and integer variable types. To share temporary storage, labeled COMMON that is to be used to contain EQUIVALENCEd storage can be defined in several program units with a different structure being specified in each.

The absence of library function argument and type checking deserves close attention. Functions which are not typed correctly by the IJKLMN convention must appear in a type declaration statement or insufficient temporary storage will be allocated for the return of the function value. All double precision, complex, and double precision complex functions must have their type declared.

SMALLER OBJECT PROGRAMS

The FORTRAN IV programmer should be aware of two means of saving considerable space:

1. The single and double precision arithmetic packages are totally distinct and each requires about 600 words of storage. If possible, use only single or double precision. To force all real variables and constants to double precision, use the statement:

   ```
   COMPILER DOUBLE PRECISION
   ```

2. Labeled COMMON takes up space at load time, whereas unlabeled COMMON and stack variables and arrays are allocated at the time of execution and thus can use the space previously occupied by the relocatable loader.

DISK BOOTSTRAPPING (HIPBOOT)

Disk bootstrapping can be performed by issuing a call to the BOOT routine. After having issued this call, HIPBOOT queries the user with:

FILENAME ?

after which the user must respond in one of three ways:

1. A carriage return, which indicates the default system SYS.SV and SYS.OL on the bootstrap device.
2. A system save file and overlay file name (the overlay file name must have the .OL extension) followed by a carriage return.
3. A partition name on the bootstrap device, with the partition containing the default system save and overlay file names SYS.SV and SYS.OL.

The system file must be prefixed by a global specifier only when an inter-device bootstrap is being performed. If an inter-device bootstrap is performed, i.e., a bootstrap of the form DPn:ABC where DPn is different from the bootstrap device, DPn will become the master device even though a different master device may have been specified during the generation of system ABC.

Not only must a system overlay file exist for each system file specified, but each overlay file must bear the same name as the system save file, with a .OL extension.

Any unrecognizable characters input to HIPBOOT will not be accepted: the TTY bell is sounded for each such character. Erroneous characters can be deleted by typing the RUBOUT key. On the TTY, a left arrow followed by the deleted character is echoed each time the RUBOUT is pressed; on video display units, the deleted character is simply erased.

If after receiving a response to the FILENAME query, HIPBOOT is unable to locate a system directory, the message:

FILE NOT FOUND, FILE: SYS.DR

will be output. This is a fatal error; full initialization is the only recovery procedure possible.

For further detail of disk bootstrapping, refer to the RDOS User's Manual, Appendix E.

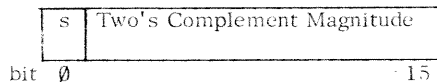
APPENDIX E

DATA STORAGE AND HANDLING

STORAGE OF DATA

Integers

Integers are stored in two's complement form, using one full 16-bit word. The allowable range is -2**15-1 to +2**15-1 (-32, 767_{10} to 32, 767_{10}). The storage format is:



where: s is the sign (Ø = plus, 1 = minus)

Real Numbers

Real numbers are stored in two words with the high order word preceding the low order word in memory. Position Ø contains the sign , bits 1 through 7 represent the exponent, and bits 8 through 31 are the mantissa.

The exponent is represented in excess 64 form, that is, as a seven digit, two's complement integer to which is added an offset of 100_8. Thus,

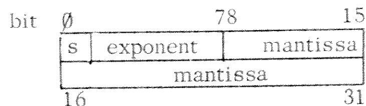
100_8 is an exponent of Ø

177_8 is an exponent of $63_{1Ø}$

077 is an exponent of $-1_{1Ø}$

The mantissa is a normalized hexadecimal fraction between .0625000 and .999999. (All floating point numbers in DGC FORTRAN IV computations are maintained in normalized form.) Real numbers have 6 to 7 decimal digits of significance.

The storage format of real numbers is:



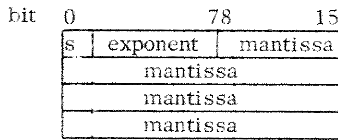
Double Precision Numbers

Double precision numbers are stored in four words. The sign and exponent are stored in the same manner as real numbers. The normalized hexadecimal mantissa is stored in the remaining 56 bits. Double precision numbers have 16 to 17 decimal digits of significance.

The storage format of double precision numbers is:

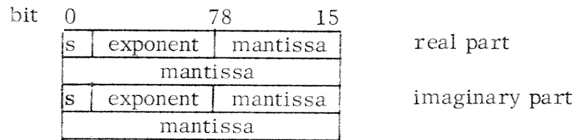
STORAGE OF DATA (Continued)

Double Precision Numbers (Continued)



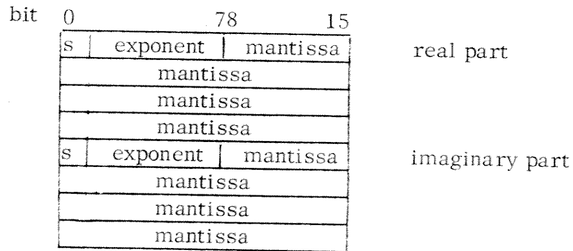
Complex Numbers

Complex numbers are stored as two real data. The real part is stored in the first two words and the imaginary part in the second two words. The storage format of complex numbers is:



Double Precision Complex Numbers

Double precision complex numbers are stored as two double precision data. The real part is stored in the first four words and the imaginary part is stored in the second four words. The storage format of double precision complex numbers is:

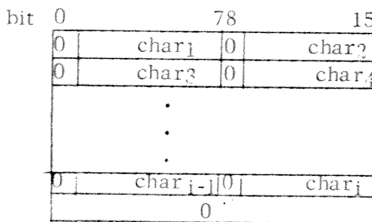


String Data

String data are stored ascending in core, with one character stored per 8-bit byte (two characters per memory word). The leftmost bit of each byte is always Ø.

If the character count of a string is odd, the terminating byte is all zeroes; if the character count is even, the string is terminated by a word of all zeroes. However, when a variable is initialized to a string datum (DATA statement) and the character count is even, no all-zero word is generated.

The storage format of string data is:



STORAGE OF DATA (Continued)

Logical Data

One word of all zeroes is stored for the value .FALSE. and one word containing -1 (177777_8) is stored for .TRUE..

DATA HANDLING

Number Stack

A stack of 630 octal locations is reserved for storage of numeric values, either as input or output or for temporary computational values.

The number stack expands dynamically as numbers are loaded onto it and contracts as they are removed.

In the event that the number stack is not large enough, the user can alter its size by defining a parameter at assembly time by means of the following statements:

```
.ENT        .FLSZ
.FLSZ = xxx
.END
```

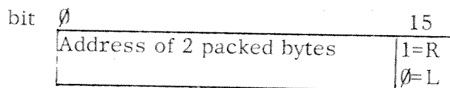
where: xxx is an octal number and the number of locations reserved for the number stack will be:

$$2 * \underline{xxx} + 30_8$$

Byte Manipulation

String data handling is accomplished through the use of byte pointers. Byte pointers are identical to those discussed in "How to Use Nova Computers" except that bit 15 is set to zero if the left byte is pointed to and bit 15 is set to one if the right byte is pointed to.

The format of the byte pointer is:



OVERFLOW CHECKING

Programmable overflow checking is provided by the library routine OVERFLOW. The calling sequence of OVERFLOW is:

```
CALL OVERFLOW ($sl1, $sl2, {"S" | "N"})
```

where: $\underline{sl}_1$ and $\underline{sl}_2$ are statement labels.

Either the literal S or the literal N appears as the third argument.

OVERFLOW CHECKING (Continued)

OVERFLOW checks a system flag to determine whether or not a non-integer arithmetic overflow has occurred since the last call to OVERFLOW. If overflow has occurred, control is returned to the statement numbered $\underline{sl}_1$. If overflow has not occurred, control is returned to the statement numbered $\underline{sl}_2$.

The system overflow flag is reset by any call to OVERFLOW and is only reset by a call to this routine.

If the argument "N" is given, all error messages will be output. If the third argument is "S" or is omitted, messages associated with floating point overflow or underflow will be suppressed.

APPENDIX F

ASSEMBLER/FORTRAN INTERFACE

This Appendix briefly describes the interface between FORTRAN and DGC assembly language programs. It covers:

the FORTRAN extension of assembly language addressing.

the assembly language code generated by a FORTRAN program.

the FORTRAN call-save-return implementation.

the interface between FORTRAN statement numbers and symbols acceptable to the assembler. (conversion to assembler-acceptable symbols can be accomplished by requesting SYMBOLS at compile time.)

For a more detailed discussion concerning the FORTRAN/Assembler Interface refer to the FORTRAN IV Run Time Library User's Manual, DGC Manual Number 093-000068.

ADDRESSING

FORTRAN addressing extends the NOVA family addressing scheme in two ways:

1. Variables on the stack are referenced relative to that stack's FSP. (FORTRAN stack pointer).

2. Full word addressing for all absolute addresses is effected by the subroutines .LD0 and .ST0.

Stack addresses are encoded as being between 0 and 377 (octal) inclusive, or as between 100000 and 100377 (the address of the variable, not the variable itself). FORTRAN addresses greater than 377 (octal) are treated as absolute .NREL addresses.

FORTRAN addresses are transformed into absolute addresses by several library routines (see list below), one of which is normally called immediately upon entry to a subprogram.

```
FRG0/FRG1
MAD/MAD0
FRGLD           ;also loads contents of this address in AC0
CPYARG/CPYLS    ;transfers effective addresses to caller's stack
FARG            ;transfers effective addresses to caller's stack
```

Any of the 377 (octal) locations on a frame can be addressed since the most recent FSP is always placed in AC3 by the FORTRAN linkage routine. Therefore, instructions similar to the following may be written:

```
LDA     0, -167, 3

STA     0, @-167, 3

ISZ     -154, 3
```

ADDRESSING (Continued)

Nova family computers can address 256 (decimal) words in an indexed instruction, using a bias of -200 through -167. Each address on the stack can then be referenced using the centerpoint, FSP, and an offset stack displacement. A FORTRAN program stack frame is laid out as follows:

LOCATION (relative to FSP)	Contents
-200	Stack frame size
-177	Old FSP from calling program
-176	Unused
-175	Entry address to the last routine called by this routine.
-174	State of carry at the time this routine issues a subroutine call.
-173	Contents of AC0 when this routine issues a subroutine call.
-172	Contents of AC1 when this routine issues a subroutine call.
-171	Contents of AC2 when this routine issues a subroutine call.
-170	Address of next sequential address (return address).
-167	Temporary storage available for use by this routine.

Relative locations -200 through -170, FSP, and AC3 are maintained by the library routines .FCALL, .FSAV, .FRET, .FRCAL, and .FQRET .

ASSEMBLER CODE GENERATED BY FORTRAN IV

```
;
;
;  C**ASSEMBLER CODE GENERATED BY FORTRAN IV**
;  C                 **SAMPLE PROGRAM**
;
;         PARAMETER Q1=4
;
;
;
;
;         SUBROUTINE SUB1 (DUM1, DUM2, DUM3, DUM4)
;         COMMON INT1
;         COMMON  UL1, UL2(Q1, Q1), UL3/CL1/ LB1, LB2, LB3(0:3, 0:3)
;         LOGICAL  L3, L4
;         LOGICAL L1, L2
;         INTEGER  DUM4
;         DIMENSION  RARAY (5, -4:0), UL21(6, 6)
;        1   , DUM1 (0, DUM2)
;         EQUIVALENCE (UL2(6), UL21(3)), (UL2(2, 3), UL22)
;        1   , (INT1, L2)
;         DATA LB1, LB2, LB3, LB3(8), LB3(3, 2)/
;        1   2*1, -5, 3H123, 'ABCDEFG'/LB3(4)/4*4/
; 10      FORMAT (I3, -4P4E26. 6/(L2, A3/3H123, T50,
;        1   "AB"), '"TITLE"')
;
;
;         SF1(SFDUM1, SFDUM2)=
;        1   SFDUM1/ABS(SIN(SFDUM2+LB3(LB1, LB2)))
;         TYPE"ERROR #", I1, "EXPECTED VALUE"
;        1   , R1, "ACTUAL", R2
;
;
;         GOTO  3
;         DO  2  ICV=1, 4
;         I1=I2*I3+I4-I5/LB1+LB2**I1. AND. 7777K
;         IF(I10. EQ. "NO")  GO TO 2
;         L1=L2. AND. L3. OR. . NOT. L4
; 2       R1=R2+R3-R4*R5/UL1**I1+ABS
;        1   (SIN(SF1(R3, R4)))
;         IF (R2-R3)3, 4, 4
;
; 3       WRITE(12)"I1=", I1, "R1=", R1
; 4       READ BINARY (13)UL2
;         RETURN DUM4
;         END
;         END
; ***076  ***  CHR  06
;  10
```

ASSEMBLER CODE GENERATED BY FORTRAN IV (Continued)

```
 0001   SUB1

^ 0002   SUB1
               ;
               ;
               ; C**ASSEMBLER CODE GENERATED BY FORTRAN IV**
               ; C          **SAMPLE PROGRAM**
               ;
               ;       PARAMETER Q1=4
               ;
               ;
               ;
               ;
               ;       SUBROUTINE SUB1 (DUM1, DUM2, DUM3, DUM4)
               ;       COMMON INT1
               ;       COMMON UL1,UL2(Q1,Q1),UL3/CL1/LB1,LB2,LB3(0:3,0:3)
               ;       LOGICAL L3,L4
               ;       LOGICAL L1,L2
               ;       INTEGER DUM4
               ;       DIMENSION RARAY(5,-4:0),UL21(6,6)
               ;      1   ,DUM1(0:DUM2)
               ;       EQUIVALENCE (UL2(6),UL21(3)),(UL2(2,3),UL22)
               ;      1   ,(INT1,L2)
               ;       DATA LB1,LB2,LB3,LB3(0),LB3(3,2)/
     000022            .COMM    CL1      22
                       .NREL
                       .TITL    SUB1
                       .ENT     SUB1
                       .NREL
     000001            .TXTM    1
                       .EXTU
                       .EXTN    .I
```

ASSEMBLER CODE GENERATED BY FORTRAN IV (Continued)

```
                 .F1:
  00000'000011 V26.:     11        ;UL21
  00001'000025 V27.:     25        ;UL22
  00002'000000 V30.:     0         ;L2
  00003'000000 V31.:     0         ;INT1
  00004'000001 V32.:     1         ;UL1
  00005'000003 V33.:     3         ;UL2



0003   SUB1
  00006'000043 V34.:     43        ;UL3
  00007'000000 .F2:      0
        000121           .CSIZ     121
                 A26.:                        ;UL21
  00010'000013'          .+3
  00011'100000'          @V26.
  00012'000110           110
  00013'000005           5
  00014'001002           1002
  00015'000001           1
  00016'000006           6
  00017'000001           1
  00020'000044           44
                 A23.:                        ;RARAY
  00021'000005           5
  00022'001002           1002
  00023'000001           1
  00024'000005           5
  00025'177774           177774
  00026'000031           31
                 A16.:                        ;LB3
  00027'000032'          .+3
        000002           .GADD     CL1,2
```

ASSEMBLER CODE GENERATED BY FORTRAN IV (Continued)

```
^ 0004   SUB1

    00031'000020               20
    00032'000005               5
    00033'000401               401
    00034'000000               0
    00035'000004               4
    00036'000000               0
    00037'000020               20
                     A33.:                         ;UL2
    00040'000043'              .+3
    00041'100005'              @V33.
    00042'000040               40
    00043'000005               5
    00044'001002               1002
    00045'000001               1
    00046'000004               4
    00047'000001               1
    00050'000020               20
                     A3.:                          ;DUM1
    00051'000003               3
    00052'001002               1002
    00053'000426'              .C3
    00054'100012               @V.+1                         ;DUM2
    00055'000043               FS.
                     SUB1:
    00056'006017$              JSR      @.CPYL
    00057'006022$              JSR      @.FINI
    00060'000000'              .F1
    00061'000007'              .F2
    00062'006020$              JSR      @.FALO
    00063'000021'              A23.
    00064'000015               V.+4
    00065'000062               62
    00066'006023$              JSR      @.FRED
    00067'000051'              A3.
    00070'100011               @V.+0
    00071'000023               V.+12
    00072'002401               JMP      @.+1
    00073'000155'              L1.
          000001               .TXTN    1
                               .GLOC    CL1
```

ASSEMBLER CODE GENERATED BY FORTRAN IV (Continued)

```
^ 0005   SUB1

        000000            .BLK     0

                  ;       1  2*1,-5,3H123,'ABCDEFG'/LB3(4)/4*4/

    00000X000001          1
                          .GLOC    CL1
          000001          .BLK     1
    00001X000001          1
                          .GLOC    CL1
          000002          .BLK     2
    00002X177773          177773
                          .GLOC    CL1
          000011          .BLK     11
                          .TXT     <1230>
    00011X030462
    00012X031400
                          .GLOC    CL1
          000015          .BLK     15
                          .TXT     <ABCDEFG>
    00015X040502
    00016X041504
    00017X042506
    00020X043400
                          .GLOC    CL1
          000005          .BLK     5
    00005X000004          4
    00006X000004          4
    00007X000004          4
    00010X000004          4
                          .NREL
          000000          .TXTN    0

                  ; 10    FORMAT(I3,-4P4E26.6/(L2,A3/3H123,T50,
                  L2.;
    00074'002401          JMP      @.+1
    00075'000127'         L3.
                          .TXT     <(I3,-4P4E26.6/(L2,A3/3H123,T50,
    00076'024111
    00077'031454
    00100'026464
    00101'050064
    00102'042462
    00103'033056
    00104'033057
    00105'024114
    00106'031054
    00107'040463
    00110'027463
    00111'044061
    00112'031063
    00113'026124
    00114'032460
    00115'026042  "AB"),'"TITLE"')10>
    00116'040502
    00117'021051
    00120'026047
    00121'021124
    00122'044524
```

ASSEMBLER CODE GENERATED BY FORTRAN IV (Continued)

```
0006   SUB1
 00123'046105
 00124'021047
 00125'024441
 00126'000000

                  ;

                  ;

                  ;        SF1(SFDUM1,SFDUM2)=
                  L3.:

 00127'000006              SFS.
                  SF1:
 00130'006017$             JSR      @.CPYL

                  ;        1  SFDUM1/ABS(SIN(SFDUM2+LB3(LB1,LB2)))

 00131'006025$             JSR      @.FSUB
 00132'000004              4
 00133'000027'             A16.                     ;LB3
 00134'000016              FVS.+1
       000000              .GADD    CL1,0           ;LB1
       000001              .GADD    CL1,1           ;LB2
```

ASSEMBLER CODE GENERATED BY FORTRAN IV (Continued)

```
^ 0007   SUB1

    00137'000013$          FXFL1
    00140'100016           @FVS.+1
    00141'000004$          FFLD1
    00142'100013           @V.+2                       ;SFDUM2
    00143'000002$          FAD1
    00144'000014$          SIN.
    00145'000001$          ABS.
    00146'000004$          FFLD1
    00147'100012           @V.+1                       ;SFDUM1
    00150'000007$          FLIP1
    00151'000003$          FDV1
    00152'000005$          FFST1
    00153'100011           @V.+0                       ;SF1
    00154'006024$          JSR      @.FRET

                     ;     TYPE"ERROR #",I1,"EXPECTED VALUE"
                     L1.:

    00155'006026$          JSR      @.FWRI
    00156'000420'          .C11
    00157'000000           0
    00160'000006           6
                           .TXT     (ERROR #)
    00161'042522
    00162'051117
    00163'051040
    00164'021400
    00165'000000           0
    00166'000001           1
    00167'000030           V.+17                       ;I1
    00170'000006           6
                           .TXT     (EXPECTED VALUE)
    00171'042530
    00172'050105
    00173'041524
    00174'042504
    00175'020126
    00176'040514
    00177'052505
    00200'000000

                     ;     1  ,R1,"ACTUAL",R2

    00201'000000           0
    00202'000002           2
    00203'000031           V.+20                       ;R1
    00204'000006           6
                           .TXT     (ACTUAL)
    00205'040503
    00206'052125
    00207'040514
    00210'000000


                     ;


                     ;
    00211'000000           0
    00212'000002           2
```

ASSEMBLER CODE GENERATED BY FORTRAN IV (Continued)

```
0008  SUB1
 00213'000033          V.+22                  ;R2
 00214'000005          5

                ;      GOTO 3
 00215'002401          JMP      @.+1
 00216'000364'         L4.

                ;      DO 2 ICV=1,4
 00217'102520          SUBZL    0,0
 00220'041635          STA      0,T.+24,3      ;ICV


A 0009  SUB1

 00221'000405          JMP      L10.
 00222'000351'         L7.
              L6.:
 00223'021635          LDA      0,T.+24,3      ;ICV
 00224'101400          INC      0,0
 00225'041635          STA      0,T.+24,3      ;ICV
              L10.:
 00226'006031$         JSR@     .LD1
 00227'000430'         .C1
 00230'106423          SUBZ     0,1,SNC
 00231'002771          JMP      @L6.-1

                ;      I1=I2*I3+I4-I5/LB1+LB2**I1.AND.7777K
 00232'021637          LDA      0,T.+26,3      ;I3
 00233'025636          LDA      1,T.+25,3      ;I2
 00234'006033$         JSR      @.SMPY
 00235'021640          LDA      0,T.+27,3      ;I4
 00236'107000          ADD      0,1
 00237'045651          STA      1,TS.+1,3
 00240'006030$         JSR@     .LD0
       000000          .GADD    CL1,0          ;LB1
 00242'025641          LDA      1,T.+30,3      ;I5
 00243'006032$         JSR      @.SDVD
 00244'021651          LDA      0,TS.+1,3
 00245'122400          SUB      1,0
 00246'041651          STA      0,TS.+1,3
 00247'021630          LDA      0,T.+17,3      ;I1
 00250'006031$         JSR@     .LD1
       000001          .GADD    CL1,1          ;LB2
 00252'006027$         JSR      @.IPWR
 00253'021651          LDA      0,TS.+1,3
 00254'107000          ADD      0,1
 00255'006030$         JSR@     .LD0
 00256'000417'         .C12
 00257'107400          AND      0,1
 00260'045630          STA      1,T.+17,3      ;I1

                ;      IF(I10.EQ."NO") GO TO 2
 00261'002401          JMP      @.+1
 00262'000265'         L11.
              L12.:    .TXT     ONO0
 00263'047117
 00264'000000
              L11.:
 00265'031642          LDA      2,T.+31,3      ;I10
 00266'006030$         JSR@     .LD0
```

ASSEMBLER CODE GENERATED BY FORTRAN IV (Continued)

```
▲ 0010   SUB1

    000267'000263'          L12.
    000270'142404           SUB      2,0,SZR
    000271'102401           SUB      0,0,SKP
    000272'102000           ADC      0,0
    000273'101004           MOV      0,0,SZR
    000274'000403           JMP      .+3
    000275'002401           JMP      @.+1
    000276'000301'          L13.
    000277'002401           JMP      @.+1
    000300'000313'          L5.
                   L13.:


                   ;        L1=L2.AND.L3.OR..NOT.L4
    000301'006030$          JSR@     .LD0
    000302'100002'          @V30.                     ;L2
    000303'025622           LDA      1,T.+11,3        ;L3
    000304'123400           AND      1,0
    000305'031621           LDA      2,T.+10,3        ;L4
    000306'150000           COM      2,2
    000307'144000           COM      2,1
    000310'107400           AND      0,1
    000311'133000           ADD      1,2
    000312'051620           STA      2,T.+7,3         ;L1

                   ; 2      R1=R2+R3-R4*R5/UL1**I1+ABS
                   L5.:
    000313'000004$          FFLD1
    000314'000033           V.+22                     ;R2
    000315'000004$          FFLD1
    000316'000043           V.+32                     ;R3
    000317'000002$          FAD1
    000320'000004$          FFLD1
    000321'000045           V.+34                     ;R4
    000322'000004$          FFLD1
    000323'000047           V.+36                     ;R5
    000324'000010$          FML1
    000325'000006$          FIPR1
    000326'000030           V.+17                     ;I1
    000327'100004'          @V32.                     ;UL1
    000330'000003$          FDV1
    000331'000011$          FSB1

                   ;        1   (SIN(SF1(R3,R4)))

    000332'006021$          JSR      @.FCAL
```

ASSEMBLER CODE GENERATED BY FORTRAN IV (Continued)

```
A 0011   SUB1

    00333'000130'        SF1
    00334'000003         3
    00335'000051         VS.+1
    00336'000043         V.+32                     ;R3
    00337'000045         V.+34                     ;R4
    00340'000004$        FFLD1
    00341'000051         VS.+1
    00342'000014$        SIN.
    00343'000001$        ABS.
    00344'000002$        FAD1
    00345'000005$        FFST1
    00346'000031         V.+20                     ;R1
    00347'000401         JMP      @.+1
    00350'000223'        L6.
                  L7.:

                  ;      IF(R2-R3)3,4,4
    00351'000004$        FFLD1
    00352'000033         V.+22                     ;R2
    00353'000004$        FFLD1
    00354'000043         V.+32                     ;R3
    00355'000011$        FSB1

                  ;
    00356'000012$        FSGN1

    00357'101133         MOVZL#   0,0,SNC
    00360'000402         JMP      @.+2
    00361'000402         JMP      @.+2
    00362'000404'        L14.
    00363'000364'        L4.

                  ; 3    WRITE(12)"I1=",I1,"R1=",R1
                  L4.:
    00364'006026$        JSR      @.FWRI
    00365'000416'        .C13
    00366'000000         0
    00367'000006         6
                         .TXT     CI1=0
    00370'044461
    00371'036400
    00372'000000         0
    00373'000001         1
    00374'000030         V.+17                     ;I1
    00375'000006         6
                         .TXT     0R1=0
    00376'051061
    00377'036400
    00400'000000         0
```

ASSEMBLER CODE GENERATED BY FORTRAN IV (Continued)

```
^ 0012  SUB1

    00401'000002            2
    00402'000031            V.+20                   ;R1
    00403'000005            5

                  ; 4       READ BINARY (13)UL2
                  L14.:
    00404'006016$           JSR      @.BRD
    00405'000415'           .C14
    00406'000000            0
    00407'000002            2
    00410'000040'           A33.                    ;UL2
    00411'000005            5

                  ;         RETURN DUM4
    00412'006015$           JSR      @.AFRT
    00413'100014            @V.+3                   ;DUM4

                  ;         END
    00414'006024$           JSR      @.FRET
    00415'000015 .C14:      000015
    00416'000014 .C13:      000014
    00417'007777 .C12:      007777
    00420'000012 .C11:      000012
    00421'000002 .C10:      000002
    00422'000006 .C7:       000006
    00423'177774 .C6:       177774
    00424'000005 .C5:       000005
    00425'000003 .C4:       000003
    00426'000000 .C3:       000000
    00427'000001 .C2:       000001
    00430'000004 .C1:       000004
           000043           FS.=43
           000006           SFS.=6
           177611           T.=-167
           000011           V.=200+T.
           177650           TS.=T.+37
           177615           FTS.=T.+4
           000250           VS.=V.+37
           000015           FVS.=V.+4

                  ;         END
                  ; *** 076 *** CHR 06
                  ;  10
           000047 R5=       V.+36
           000045 R4=       V.+34
           000043 R3=       V.+32
           000042 I10=      V.+31
           000041 I5=       V.+30
           000040 I4=       V.+27
```

ASSEMBLER CODE GENERATED BY FORTRAN IV (Continued)

```
^ 0013   SUB1

          000037 I3=       V.+26
          000036 I2=       V.+25
          000035 ICV=      V.+24
          000033 R2=       V.+22
          000031 R1=       V.+20
          000030 I1=       V.+17
          100001'UL22=     @V27.
          000010'UL21=     A26.
          000015 RARAY=    V.+4
          100202'L2=       @V30.
          000020 L1=       V.+7
          000021 L4=       V.+10
          000022 L3=       V.+11
          000027'LB3=      A16.
                 ;  LB2=  .GADD    CL1,1
                 ;  LB1=  .GADD    CL1,0
          100006'UL3=      @V34.
          000040'UL2=      A33.
          100004'UL1=      @V32.
          100003'INT1=     @V31.
          100014 DUM4=     @V.+3
          100013 DUM3=     @V.+2
          100012 DUM2=     @V.+1
          000023 DUM1=     V.+12
          000404'N4=       L14.
          000313'N2=       L5.
          000364'N3=       L4.
          000074'N10=      L2.
                           .END
```

ASSEMBLER CODE GENERATED BY FORTRAN IV (Continued)

```
  ØØ14   SUB1
A15.     ØØØØ27'
A23.     ØØØØ21'
A26.     ØØØØ1Ø'
A33.     ØØØØ4Ø'
A3.      ØØØØ51'
ABS.     ØØØØØ1$X
DUM1     ØØØØ23
DUM2     1ØØØ12
DUM3     1ØØØ13
DUM4     1ØØØ14
FAD1     ØØØØØ2$X
FDV1     ØØØØØ3$X
FFLD1    ØØØØØ4$X
FFST1    ØØØØØ5$X
FIPR1    ØØØØØ6$X
FLIP1    ØØØØØ7$X
FML1     ØØØØ1Ø$X
FSB1     ØØØØ11$X
FSGN1    ØØØØ12$X
FS.      ØØØØ43
FTS.     177615
FVS.     ØØØØ15
FXFL1    ØØØØ13$X
I1       ØØØØ3Ø
I1Ø      ØØØØ42
I2       ØØØØ36
I3       ØØØØ37
I4       ØØØØ4Ø
I5       ØØØØ41
ICV      ØØØØ35
INT1     1ØØØØ3'
L1       ØØØØ2Ø
L1Ø.     ØØØ226'
L11.     ØØØ265'
L12.     ØØØ263'
L13.     ØØØ3Ø1'
L14.     ØØØ4Ø4'
L1.      ØØØ155'
L2       1ØØØØ2'
L2.      ØØØØ74'
L3       ØØØØ22
L3.      ØØØ127'
L4       ØØØØ21
L4.      ØØØ364'
L5.      ØØØ313'
L6.      ØØØ223'
L7.      ØØØ351'
L93      ØØØØ27'
M1Ø      ØØØØ74'
M2       ØØØ313'
M3       ØØØ364'
M4       ØØØ4Ø4'
R1       ØØØØ31
R2       ØØØØ33
R3       ØØØØ43
R4       ØØØØ45
R5       ØØØØ47
RARAY    ØØØØ15
SF1      ØØØ13Ø'
```

ASSEMBLER CODE GENERATED BY FORTRAN IV (Continued)

```
   0015    SUB1
 SFS.      000006
 SIN.      000014$X
 SUB1      000256'
 TS.       177650
 T.        177611
 UL1       100004'
 UL2       000040'
 UL21      000010'
 UL22      100021'
 UL3       100006'
 V26.      000000'
 V27.      000001'
 V30.    • 000002'
 V31.      000003'
 V32.      000004'
 V33.      000005'
 V34.      000006'
 VS.       000050
 V.        000011
 .AFRT     000015$X
 .RRD      000016$X
 .C1       000430'
 .C10      000421'
 .C11      000420'
 .C12      000417'
 .C13      000416'
 .C14      000415'
 .C2       000427'
 .C3       000426'
 .C4       000425'
 .C5       000424'
 .C6       000423'
 .C7       000422'
 .CPYL     000017$X
 .F1       000000'
 .F2       000007'
 .FALO     000020$X
 .FCAL     000021$X
 .FINI     000022$X
 .FRED     000023$X
 .FRFT     000024$X
 .FSUB     000025$X
 .FWRI     000026$X
 .I        177777 X
 .IPWR     000027$X
 .LD0      000030$X
 .LD1      000031$X
 .SDVD     000032$X
 .SMPY     000033$X
```

Description of Generated Code

Following is a description of the assembly language code generated by the FORTRAN compilation.

Declarations

Label	Code	Operand	Description
	.COMM	name n	At load time, reserve a labeled COMMON block of n words, by name if name is new to the loader. If the name is in the loader symbol table, check that n is equal to the previous n.
	.TITL	.MAIN or subprogram name	Program title for loader and debugger.
	.ENT	.MAIN or subprogram name	Declare the main program (subprogram) name external to the loader.
	.NREL		Normally relocatable code. No page zero code is generated by the compiler.
	.TXTM	1	ASCII code is stored left to right in a word.
	.EXTU		Treat all undefined symbols as if they had appeared in an .EXTD statement.
	.EXTN	.I	Force loading of FORTRAN initialization routine from the library.
.F1:			
Vn. :	n		Pointers to unlabeled COMMON variables. These are displacements relative to the beginning of unlabeled COMMON.
.			
.			
.			
.F2:	0		Made non-zero after pointers are initialized.
	.CSIZ	n	Size of unlabeled COMMON in words.

Array Specifier

Label	Code	Operand	Description
.Axxx:			
	.+3		Pointer to subscript bound specifier. (COMMON arrays only).
	@Vn.	or	Address of first data element of array.
	.GADD	name n	(Address of first data element of array — see above)
	n		Array size in computer words.
	k		2 * (number of subscripts) + 1 .
	400 * len + type		Element length and type. Types are: 1 = integer 2 = real 3 = double precision 4 = complex 5 = double precision complex
	subscript lower bounds, alternated with partial products		

Description of Generated Code (Continued)

Executable Code

	FS.	Program unit's frame size.
name:		Entry to program
	JSR @.CPYL	Copy argument addresses onto this program's stack.
	JSR @.FINI .F1 .F2	Add address of beginning of unlabeled COMMON to the displacements.
	JSR @.FALO array specifier 3-word stack specifier array size	Allocate array on the run-time stack.
	JSR @.FRED array specifier 3-word stack specifier array size	Redimension an array passed as a dummy argument.
	JMP @.+1 L1.	Jump around any statement functions or any internal subprograms.
	.TXTN 1	Forces any text string of even number of bytes to terminate with a word containing the last two characters of the string
	.GLOC name	Temporarily change the loader's program counter to the value of name.
	.BLK n . . .	Allocates a block of storage equal to n number of words.
	.TXTN 0	All text strings containing an even number of bytes will terminate with a full word zero.

Body of Program Unit

	SFS.	Stack frame size.
name:		
	JSR @.CPYL	
	JSR @.FINI	Unlabeled COMMON pointers are initialized.
	. . .	Coding required for program.
	JSR @.FRET	

Description of Generated Code (Continued)

Definition of Stack Parameters

FS. = m	Program unit's frame size for all stack variables (excluding arrays) and all compiler generated temporary variables.
SFS.=6	Program unit's stack frame size.
T.= -167	In instructions such as LDA 0, T.±n, 3 , register 3 points to the middle of the user's stack. The words at locations -200 through -170 relative to register 3 are used for saving accumulators, Carry, etc. The word -167 relative to register 3 is the first available for variable or temporary storage.
V.=200+T.	V. is used in full-word addresses to refer to variables on the user stack. Displacements involving V. are relative to the beginning of a user's stack frame rather than the middle.
TS.=T.±n	TS.+1 is the displacement of the first word of stack storage available for compiler temporary variables.
FTS.=T.±n	Same information as TS., applied to statement functions.
VS.=V.±n	Used in full-word addressing of compiler generated temporaries.
FVS.=V.±n	Same information as VS., applied to statement functions.
symbols	See pages following.

CALLING AND RECEIVING SEQUENCES

The form of the calling sequence generated from the proposed FORTRAN statement CALL NAME (x, y, z) is as follows:

```
.EXTN NAME
JSR @.FCAL
NAME
3                              ;where n = number of arguments
FORTRAN ADDRESS of x
FORTRAN ADDRESS of y
FORTRAN ADDRESS of z
```

The .FCAL routine calls a subroutine which has no page zero entry, or calls a routine (which has a page zero entry) without using its page zero entry. .FCAL creates a new stack for the called routine (if needed) and allocates temporary storage space on the new stack if this is required, determined by a stack length word. The accumulators (except AC3) and the original state of Carry are restored. And AC3 contains the current FSP.

The converse of the calling sequence generated by a FORTRAN call statement is the receiving sequence. This is the means by which the calling parameters are fetched by the called subroutine. The form of the receiving sequence generated by FORTRAN is:

```
        FS.
NAME:   JSR    @.CPYL
        .
        .
        .
```

The routine .CPYL converts the n argument addresses to effective addresses and places these addresses in relative locations -167 through -167+n on the called program's stack frame. Even if no arguments are to be passed, .CPYL is still called so that program control will return to the next sequential FORTRAN statement.

The assembly language code generated by a FORTRAN RETURN statement is:

```
JSR    @.FRET
```

.FRET restores accumulators, carry, contents of FSP, and places FSP in AC3.

There are several points to bear in mind when coding an assembly language subroutine, they are:

The programmer must provide linkage with other programs.

The program name must be declared as an entry in an initial statement (i.e., .ENT name).

Library routines must be mentioned in either .EXTD or .EXTN statements. (FORTRAN Run Time Library User's Manual details which routines are to be declared as .EXTN and which routines are to be declared .EXTD.)

If any .TXT statement is passed in a FORTRAN routine, the first must be preceded by a statement to force the storing of text as left-to-right. (.TXTM 1)

Precede the first statement which generates binary code with .NREL to make the assembly language program relocatable.

CALLING AND RECEIVING SEQUENCES

To compute FS., the frame size for a program, count one word for each dummy argument, plus the number of words for working storage.

For example, if the assembly language program receives three arguments and uses four integer variables and three real variables, set FS.=15_8, where:

15 = 3 (dummy args) + 4 (integers)
   + 3 (real variables @ 2 words each)

To keep track of variables mnemonically, you could define your variables as follows:

```
DUM1    =   -167
DUM2    =   DUM1+1
DUM3    =   DUM2+1
INT1    =   DUM3+1
INT2    =   INT1+1
INT3    =   INT2+1
INT4    =   INT3+1
REAL1   =   INT4+1
REAL2   =   REAL1+2
REAL3   =   REAL2+2
FS.     =   REAL3-DUM1+2
```

and define: A. = 200 Then reference as follows:

```
DSZ     @DUM1,3         ;MEMORY
LDA     =,@DUM2,3       ;REFERENCE
STA     1,INT3,3        ;INSTRUCTIONS

JSR     @.FCAL
.EXTN   USER3
USER3
1
A.+REAL2        ;ARGUMENT IN CALLING SEQUENCE
```

The indirect reference (@sign) is used for dummy variables, since the address, not the variable, is on the stack.

The subroutine ISHIFT (IN, N, OUT) to shift IN righ by N bits might be coded as follows:

```
.TITLE      ISHIFT
.ENT        ISHIFT
.EXTD       .FRET,.CPYL,.FCAL
.EXTN       ERMES

.TXTM       1

.NREL

IN=         -167
N=          IN+1
OUT=        N+1
FS.=        OUT-IN+1
```

CALLING AND RECEIVING SEQUENCES (Continued)

```
          FS.
ISHIFT:   JSR       @.CPYL
          LDA       0,@N,3
          MOVL#     0,0,SZC
          JMP       ISHER  ;SHIFT COUNT NEG
          LDA       1,C20
          SUBZ      0,1,SNC
          JMP       ISHER ;SHIFT COUNT 16
          NEG       0,0,SNR
          JMP       ISHER
          LDA       1,@IN,3
          MOVZR     1,1
          INC       0,0,SZR
          JMP       .-2
ISH2:     STA       1,@OUT,3
          JSR       @.FRET
ISHER:    JSR       @.FCAL
          ERMES
          1
          ISHMS
          SUB       1,1  ;RETURN VALUE OF 0.
          JMP       ISH2

ISHMS:    .TXT      "IMPROPER SHIFT COUNT"

C20:      20
          .END
```

USER SYMBOLS

FORTRAN variable names and statement numbers are not acceptable in assembler source code. Variable names may be too long (more than 5 characters). Statement numbers would be treated as quantities rather than as labels. Therefore, variable names and statement numbers are replaced by generated variable numbers and label numbers respectively in the assembler source code.

The compiler can make FORTRAN variable names and statement numbers with their associated FORTRAN address values available to the assembler for information purposes and for use with the debugger. If SYMBOLS are requested at compile time, the variable names and statement numbers will appear in assembler equivalence statements such as those that follow:

```
                  .
                  .
                  .
K        = @V.+0              (dummy variable)
R        = V.+1               (stack variable)
J        = V.+3               (stack variable)
CV1      = @.V7               (unlabeled common variable)
ARA1     = V.+4               (array specifier)
ARA2     = .A5                (array specifier)
.LC1     = .GADD LC1,0        (labeled common variable)
.100     = .L3                (statement number)
.1       = .L2                (statement number)
.999     = .L7                (statement number)
.END
```

The = statement gives the assembler a binary value for the symbol. This has no effect on the assembly of the body of the program. Symbols which are identical in the first five characters will be considered multiply-defined by the assembler, but this will not impair the assembly or execution

USER SYMBOLS (Continued)

of the program. Its sole effect is that only the first of each set of symbols considered multiply-defined will be passed from the assembler to the loader to the debugger.

The example above shows that only statement numbers and some array specifiers are referenced by relocatable addresses directly usable by the debugger. The other addresses must be interpreted, as is done by the FORTRAN run-time library programs.

The simplest to interpret are the unlabeled COMMON addresses of the form "@.Vn". These are indirect relocatable addresses. Thus, the effective address for this variable is to be found at address ".Vn". This address will not be correct until the program or subprogram containing the address has called library program .FINI at least once. .FINI takes the relocatable addresses stored in the .Vn words and adds the address of the beginning of unlabeled COMMON to them to make effective addresses.

Labeled COMMON variable names are preceded by a semicolon to make the line a comment to the assembler.

The assembler cannot accept the syntax of .GADD name, n within an = statement. The programmer can use the information in these comments to locate his variables by finding the value of "name" (the name of the COMMON area) at execution time and adding the integer n to this.

Address values containing "V.+" refer to the runtime stack. "V.+" quantities evaluate to between 11_8 and 377_8. This value is a displacement from an origin which can be found at run-time in page zero location FSP (absolute memory location 16_8). To determine the effective address of a quantity on the stack, calculate:

$$(FSP) + (V.+n) - 200_8$$

where: (FSP) is the contents of FSP

(V.+n) is a number between 11_8 and 377_8.

Address values specified as @V.+n mean that the quantity itself is not on the stack but rather the address of the quantity is on the stack. This is the means of addressing arguments passed to a subprogram.

Statement numbers are turned into symbols acceptable to the assembler by prefixing a "." to the number.

APPENDIX G

FORTRAN IV RUN TIME REENTRANCE AT INTERRUPT TIME

FORTRAN run time routines are reentrant at interrupt time. To reenter a FORTRAN run time routine from interrupt level, i.e., to make an FCALL, the user must save certain page zero stack pointers for later restoration. At the same time, the user must alter the value of certain page zero pointers. In effect, the user is making use of stack space allotted to the interrupted program for use by the routines to be called at interrupt time.

The coding required to save and restore page zero values, to change values for the time time call, and to make a return is the same whether the user is in a real time, multitasking environment or whether he is in a non-real time, single-tasking environment. In either case, he is borrowing stack space from the interrupted routine, whose state is temporarily frozen.

The steps the user must take before calling a FORTRAN run time routine at interrupt time are given below. The page zero variables involved are briefly defined. However, for a more complete meaning of each of the pointers and the stacks to which they point, see the FORTRAN IV Run Time Library User's Manual, 093-000068.

Before calling a FORTRAN run time routine at interrupt time, the user should:

1. Save the contents of .SV0.

2. Save the contents of .OVFL.

3. Save the contents of FSP. FSP is the FORTRAN Linkage Stack Pointer.

4. Increment SP.

5. Create a temporary NSP as follows:

   C(NSP) + 6 → NSP

   NSP is the FORTRAN Number Stack pointer. The value will later be restored to its original value upon return.

6. Create a new FSP as follows:

   C(FSP) + FLGT + 2*FFEL → FSP'

   FLGT is the length of the variable portion of the FORTRAN Linkage Stack, and FFEL is the 11_8 fixed header of the FORTRAN Linkage Stack. The original value of FSP is later restored upon return.

7. Create a temporary QSP as follows:

   C(FSP') + FAC2 → QSP'

   QSP is later restored upon return.

8. Allocate two temporary words in the new (FSP') frame. (These can be used to save .SV0 and .OVFL.)

FORTRAN IV RUN TIME REENTRANCE AT INTERRUPT TIME (Continued)

The following shows how the new FSP and QSP can be created before the run time call:

```
          LDA     3, FSP
          MOV     3,2
          LDA     0, FLGT, 3      ;COMPUTE FSP'
          LDA     1, MAGIC        ;COMPUTE FSP'
          ADD     0,1             ;COMPUTE FSP'
          ADD     1,3             ;COMPUTE FSP'
          STA     3, FSP          ;STORE FSP'
          LDA     0, TWO          ;TEMPORARY STORAGE
                                  ;FOR .SV0, .OVFL
          STA     0, FLGT, 3      ;SIZE OF VARIABLE
                                  ;FRAME IN FLGT
          STA     2, FOSP, 3      ;SAVE OLD FSP
          LDA     0, .SV0
          STA     0, SAV0, 3      ;STORE .SV0
          LDA     0, .OVFL, 3
          STA     0, OVFL, 3      ;STORE .OVFL
          LDA     0, ABC
          ADD     3, 0
          STA     0, QSP          ;STORE QSP
ABC:      FAC2
MAGIC:    2*FFEL
TWO:      2
SAV0=     FTSTR
OVFL=     SAV0+1
```

On return from the FORTRAN run time routine, the user must restore the values and conditions for the interrupted task:

1. Restore C(.SV0)

2. Restore C(.OVFL)

3. Decrement SP

4. Restore NSP:

   C(NSP) - 6 → NSP

5. Restore C(FSP) from the saved value.

6. Restore C(QSP), again using saved FSP (not FSP'):

   C(FSP) + FAC2 → QSP

The following code will restore the saved value of FSP:

```
          LDA     3, FSP
          LDA     3, FOSP, 3
          STA     3, FSP
```

DataGeneral

PROGRAMMING DOCUMENTATION REMARKS FORM

Document Title	Document No.	Tape No.

SPECIFIC COMMENTS: List specific comments. Reference page numbers when applicable. Label each comment as an addition, deletion, change or error if applicable.

GENERAL COMMENTS: Also, suggestions for improvement of the Publication.

FROM:

Name	Title		Date
Company Name			
Address (No. & Street)	City	State	Zip Code

Form No. 10-24-004

FOLD DOWN FIRST FOLD DOWN

FIRST
CLASS
PERMIT
No. 26
Southboro
Mass. 01772

BUSINESS REPLY MAIL

No Postage Necessary If Mailed In The United States

Postage will be paid by:

Data General Corporation

Southboro, Massachusetts 01772

ATTENTION: Programming Documentation

FOLD UP SECOND FOLD UP

STAPLE