Stardent

FORTRAN REFERENCE MANUAL

Stardent Computer Inc.

Part Number: 340-0111-01

Change History

340-0009-02	Original
340-0009-03	February, 1989 – Software Release 2.0
340-0111-01	January, 1990 – Software Release 3.0

CONTENTS

2 Fortran Statements

3 Fortran I/O Statements

B Data Layout and Calling Conventions

Index

List of Figures

List of Tables

PREFACE

This manual is a reference for the Stardent 1500/3000 Fortran programming language. It is not a tutorial on the Fortran language. However, it describes the elements of the language as well as other elements that specifically address Stardent 1500/3000 Fortran. These include programming statements, input/output statements, file handling instructions, procedures and subprograms, math and intrinsic functions, compiler options, user-controlled parallelism, program transformations, inline functions, vector reporting facility, compiler directives, asynchronous I/O, statements on profiling programs, environment setting variables, library functions, error messages, data layout, and finally, calling conventions.

Structure of This Manual

The manual has six sections and two appendices. Here is a short description of each section and the appendices.

- Section 1 includes chapters 1 to 5 — *Fortran Language Elements*. It discusses everything concerning the language.
  - Chapter 1 – *Language Elements*

    This chapter describes the basics of the Stardent 1500/3000 Fortran language. The character set is identified, keywords and symbolic names are defined, and data types are described.
  - Chapter 2 – *Fortran Statements*

    This chapter provides information on all the statements in the Stardent 1500/3000 Fortran language. The required order of statements is defined, followed by an alphabetical listing of the statements.

- Chapter 3 – *Fortran I/O Statements*

  This chapter describes in detail the input/output statements used in the Stardent 1500/3000 Fortran. All format and edit descriptors are defined with examples showing their application.

- Chapter 4 – *File Handling*

  This chapter describes the file handling statements and procedures available in the Stardent 1500/3000 Fortran. These include the **READ**, **WRITE**, **OPEN**, **CLOSE**, **INQUIRE**, and file-positioning statements. In addition, it also briefly describes the Stardent 1500/3000 environment setting variables which allow you to read or write files imported from VMS and most BSD UNIX systems.

- Chapter 5 – *Procedures and Subprograms*

  This chapter describes procedures and block data subprograms. The procedures include function subprograms, subroutine subprograms, statement functions, and intrinsic functions.

- Section 2 consists of Chapter 6 — *Intrinsic Functions*. This chapter provides a table of intrinsic functions. The table includes the definition of each function, the number of arguments, the generic name of each group of functions, the specific name for each function, the types of arguments allowed, and the type of result of intrinsic functions.

- Section 3 contains Chapter 7 — *Compiler Options*. It concentrates on compiler options which the Stardent 1500/3000 supports. This chapter categorizes specific options together, so that it is easy for you to refer directly to what you need. Those categories are: standard compilation options, input/output options, debugging options, optimization options, porting aid options, and miscellaneous options.

- Section 4 is a stand-alone chapter, Chapter 8, *Stardent 1500/3000 Fortran Optimization Facilities*. This chapter specifically describes the Stardent 1500/3000 optimization facilities such as vector reporting facility, program transformations, compiler directives, inline functions, user-controlled parallelism, and asynchronous I/O.

- Section 5 consists of Chapter 9, *Profiling Programs*. This chapter covers profiling programs on the Stardent 1500/3000.

- Section 6 covers user commands and library functions in two separate chapters. The following provides a short description of each chapter.

  - Chapter 10 – *User Commands*

    This chapter describes several important user commands in the Fortran programming environment. Those include Fortran compiler and its options, environment setting variables which allow you to read or write VMS or BSD unformatted files on the Stardent 1500/3000 system, a filter to print Fortran files, a multi-routine Fortran file splitter, and a conversion program which can be used to convert a rational dialect of Fortran into ordinary irrational Fortran.

  - Chapter 11 – *Library Functions*

    This chapter contains the UNIX manual pages that describe the functions built into the Fortran support library. These library functions include I/O functions, file access functions, asynchronous I/O functions, system functions, timing functions, UNIX utility functions, and VMS compatibility functions. These functions may be directly called from the source program. The loader links the library functions into the executable file.

- Appendix A contains a list of error messages that are provided to you during the compilation or execution of Fortran programs.

- Appendix B consists of two related topics, data layout and language calling conventions in the Stardent 1500/3000 system. Each topic is fully described to serve its useful purpose.

Related Documentation

More information on Fortran and the UNIX Operating System can be found in the following manuals:

- *Programmer's Guide*
- *Programmer's Reference Manual, Vol. 1*
- *Programmer's Reference Manual, Vol. 2*
- *Commands Reference Manual*
- *System Administrator's Guide*

Intended Audience

This manual is written for users who have previous knowledge of the Fortran programming language or for someone who at least has been exposed to some similar high-level language. If you are not familiar with the Fortran programming language or any similar languages, you should acquire some basics through other available commercial materials.

LANGUAGE ELEMENTS

CHAPTER ONE

A Fortran program is a sequence of statements which, when executed in a specified order, process data to produce desired results. Because each program has different data needs, Fortran provides nine data types for constants, variables, functions, and expressions. Stardent 1500/3000 Fortran also provides three additional constant formats, which are extensions to Fortran 77. All are described in *Data Types and Constants* later in this chapter. Keywords, special characters, special symbols, symbolic names, and data make up the statements of a Fortran program. This chapter describes the elements of statements.

1.1 The Stardent 1500/3000 Fortran Character Set

Each language element is written using the letters **A** through **Z**, the digits **0** through **9**, and the following special characters:

	Blank	=	Equals
+	Plus	-	Minus
*	Asterisk	/	Slash
!	Exclamation point	_	Underscore (break)
(	Left parenthesis	)	Right parenthesis
%	Percent sign	&	Ampersand
,	Comma	.	Decimal point
$	Dollar sign	:	Colon
<	Left angle bracket	>	Right angle bracket
'	Single quotation mark (apostrophe)		
"	Double quotation mark		

The backslash escape, within character strings or Hollerith constants, is used to permit normal file system communication (that is "\n" to force a vertical tab).

As an extension to Fortran 77, the 26 lowercase letters (a through z) are allowed. The compiler considers them identical to their uppercase equivalents (unless **-case_sensitive** is used; refer to Chapter 7, *Compiler Options* for detailed information on this option), except in character or Hollerith constants. Note that this differs from the C language, in which lowercase letters are distinct from uppercase letters in identifiers. Lowercase letters improve program readability. In addition, any printable ASCII character can be used in a character or Hollerith constant or a comment.

Blanks can be used anywhere within a statement. However, the first characters of a compiler directive must be exactly **C$DOIT** or **CDIR$**.

A tab character in column 1 through 6 is treated as a tab to column 7, and a tab character in any other column is treated as a single blank. However, tab formatting for the entry of Fortran statements is allowed as follows:

- A tab character following a statement number in columns 1 through 5 is treated as though you have spaced over to at least column 7, the normal starting position of statement text.
- If you begin a line with a tab character:
  - if the first character following the tab is a numeral, then it is treated as a continuation character (that is, as though the numeral had been placed in column 6, the normal continuation column). No normal Fortran keyword begins with a numeral, so this is the indication of a continuation.
  - if the first character following a leading tab character is other than a numeral, this character is treated as the first character of a keyword or variable.

1.2 Special Symbols

The special symbols are groups of characters that define specific operators and values. The special symbols of Fortran are shown in Table 1-1 that follows.

Table 1-1. Special Symbols

//	Concatenation	**	Exponentiation
.TRUE.	Logical true	.FALSE.	Logical false
.NOT.	Logical negation	.AND.	Logical AND
.OR.	Logical OR	.EQV.	Logical equivalence
.XOR.	Synonym for .NEQV.	.EOR.	Synonym for .NEQV.
.EQ.	Equal	.NE.	Not equal
.LT.	Less than	.LE.	Less than or equal
.GT.	Greater than	.GE.	Greater than or equal
.NEQV.	Logical nonequivalence (exclusive OR)		

1.3 Keywords

Keywords are predefined Fortran entities that identify a statement or compiler option. Symbolic names can be identical to keywords because the interpretation of a sequence of characters is implied by the context in which it appears. The keywords for Fortran are listed in Table 1-2.

Table 1-2. Statement Keywords

ACCEPT[1]	ENCODE[1]	OPEN
ASSIGN	END	OPTIONS[1]
BACKSPACE	END DO[1]	PARAMETER
BLOCK DATA	END IF	PAUSE
CALL	ENDFILE	PRINT
CHARACTER	ENTRY	PROGRAM
CLOSE	EQUIVALENCE	READ
COMMON	EXTERNAL	REAL
COMPLEX	FORMAT	RETURN
CONTINUE	FUNCTION	REWIND
DATA	GOTO	SAVE
DATE	IF	STOP
DECODE[1]	IMPLICIT	SUBROUTINE
DIMENSION	INCLUDE[1]	TYPE[1]
DO	INQUIRE	WRITE
DO WHILE[1]	INTEGER	
DOUBLE COMPLEX[1]	INTRINSIC	
DOUBLE PRECISION	LOGICAL	
ELSE	NAMELIST[1]	
ELSE IF	NONE	

[1] Extension to Fortran 77.

1.4 Symbolic Names

Symbolic names are entities that define main program, procedure, block data subprogram, common block, named constant, namelist, or variable names. Each symbolic name consists of a sequence of characters, the first of which must be a letter. The remainder can be letters, digits, or, as an extension to Fortran 77, the underscore character (_) or the dollar sign ($). The letters can be upper case or, as an extension to Fortran 77, lower case. The name can be up to 31 characters in length and all characters are significant.

NOTE
Case is significant when characters appear in Hollerith or character constants, but not otherwise.

External names are those names used by the linker. In Fortran, external names are generated for subroutines, functions, entries, and common blocks. The external name is the internal name with lower case letters converted into their upper case equivalents.

Examples of Symbolic Names

```
INITIALIZATION_SUBROUTINE     REAL_VALUE
char_string                   sum_of_real_values
NumBer_of_ERRors              error_flag
```

Because upper- and lowercase letters are not distinguished within symbolic names, the following are equivalent:

```
result3
RESULT3
ResulT3
```

The name that identifies a variable, named constant or a function also identifies its default data type. A first letter of I, J, K, L, M, or N implies type integer. Any other letter implies type real. This default implied typing can be changed with an **IMPLICIT** statement or by a type statement. A symbolic name that identifies a main program, subroutine, block data subprogram, namelist, or common block has no data type.

A keyword may be used as a symbolic name, but that is usually bad practice. The symbolic name of a named constant or variable can be the same as the symbolic name of a common block without conflict.

EXAMPLES

```
READ = IF + DO * REAL
```

READ, IF, DO, and **REAL** are recognized as variables. They can also be used elsewhere as keywords in statements.

```
IF (IF .EQ. GOTO) GOTO99
```

Within the logical expression, **IF** and **GOTO** are recognized as variables. The **IF** and **GOTO** outside the expression are recognized as statement keywords.

```
DO 10 j = 1.5
```

The symbol **DO 10 j** is recognized as a variable, even though it contains blanks, mixed case, and the characters **DO**.

Although Stardent 1500/3000 Fortran permits these examples, using them is poor programming practice because they inhibit program readability.

1.4.1 Predefined Symbolic Names

Intrinsic functions have symbolic names that are predefined by Fortran. Intrinsic functions are discussed in detail in Chapter 5, *Procedures and Block Data Subprograms* and in Chapter 6, *Intrinsic Functions*. A list of the intrinsic functions of Fortran is in the table that follows. An asterisk marks those names which are extensions to Fortran 77.

1.4.1.1 Predefined Symbolic Names (Intrinsic Functions)

A user-defined function can not have the same name as an intrinsic function when they are located in the same program unit. If the user-defined function is located in a different program unit than the intrinsic function, or if the name appears in an **EXTERNAL** statement, identical names may be used. Predefined intrinsic functions names are listed in table Table 1-3. (Also refer to Chapter 2, *Fortran Statements* in the section *EXTERNAL Statement*).

Table 1-3. Predefined Intrinsic Function Names

ABS	CLOG	DNINT	IINT*	JMIN0*
ACOS	CMPLX	DPROD	IIOR*	JMIN1*
ACOSD*	CONJG	DREAL*	IISHFTC*	JMOD*
AIMAG	COS	DSIGN	IISHFT*	JNINT*
AIMAX0*	COSD*	DSIN	IISIGN*	JNOT*
AIMIN0*	COSH	DSIND*	IMAX0*	JZEXT*
AINT	CSIN	DSINH	IMAX1*	LEN
AJMAX0*	CSQRT	DSQRT	IMIN0*	LGE
AJMIN0*	DABS	DTAN	IMIN1*	LGT
ALOG	DACOS	DTAND*	IMOD*	LLE
ALOG10	DACOSD*	DTANH	INDEX	LLT
AMAX0	DASIN	EXP	ININT*	LOG
AMAX1	DASIND*	FLOAT	INOT*	LOG10
AMIN0	DATAN	FLOATI*	INT	MAX
AMIN1	DATAN2	FLOATJ*	IOR*	MAX0
AMOD	DATAN2D*	IABS	ISHFTC*	MAX1
ANINT	DATAND*	IAND*	ISHFT*	MIN
ASIN	DBLE	IBCLR*	ISIGN	MIN0
ASIND*	DCMPLX*	IBITS*	IZEXT*	MIN1
ATAN	DCONJG*	IBSET*	JIABS*	MOD
ATAN2	DCOS	ICHAR	JIAND*	NINT
ATAN2D*	DCOSD*	IDIM	JIBCLR*	NOT*
ATAND*	DCOSH	IDINT	JIBITS*	RAN*
BITEST*	DDIM	IDNINT	JIBSET*	REAL
BJTEST*	DEXP	IEOR*	JIDIM*	SIGN
BTEST*	DFLOAT*	IFIX	JIDINT*	SIN
CABS	DFLOTI*	IIABS*	JIDNNT*	SIND*
CCOS	DFLOTJ*	IIAND*	JIEOR*	SINH
CDABS*	DIM	IIBCLR*	JIFIX*	SNGL
CDCOS*	DIMAG*	IIBITS*	JINT*	SQRT
CDEXP*	DINT	IIBSET*	JIOR*	TAN
CDLOG*	DLOG	IIDIM*	JISHFTC*	TAND*
CDSIN*	DLOG10	IIDINT*	JISHFT*	TANH
CDSQRT*	DMAX1	IIDNNT*	JISIGN*	ZEXT*
CEXP	DMIN1	IIEOR*	JMAX0*	
CHAR	DMOD	IIFIX*	JMAX1*	

* Extension to Fortran 77.

1.5 Data Types

Except for Hollerith, octal, and hexadecimal constants, every constant, variable, function, and expression is of one type only. Hollerith constants are of no type and conform to the context in which they appear, and octal and hexadecimal constants assume a numeric data type based on the way they are used. The type defines:

- The set of values an entity of that type may assume.
- The amount of storage variables of that type require.
- The set of permissible operations that which can be performed on an entity of that type.

The eleven data types provided by Stardent 1500/3000 Fortran are shown in Table 1-4, together with the range and storage of each.

Table 1-4. Data Types

TYPE	RANGE	STORAGE
INTEGER*4	-2147483648 to +2137483647	1 word/32 bits
INTEGER*2	-32768 to +32767	half word/16 bits
BYTE	-128 to 127	1 byte/8 bits
REAL*4	-3.402823E+38 to -1.175495E-38 0 1.175495E-38 to 3.402823E+38	1 word/32 bits
REAL*8 or DOUBLE PRECISION	-1.79769313486231D+308 to -2.22507385850721D-308 0 2.22507385850721D-308 to 1.79769313486231D+308	2 words/64 bits
COMPLEX*8	Each component same as real	2 words/64 bits
COMPLEX*16 or DOUBLE COMPLEX	Each component same as double precision	4 words/128 bits
LOGICAL*4	.TRUE. or .FALSE.	1 word/32 bits
LOGICAL*2	.TRUE. or .FALSE.	half word/16 bits
LOGICAL*1	-128 to +127	1 byte/8 bits
character	Entire 8-bit ASCII character set	one byte/8 bits (for each character)

A constant is a data element that represents one specific value, such as **-3, .TRUE., 'character constant', 47.21E-8,** and so on. With the **PARAMETER** statement (described in *PARAMETER Statement* in Chapter 2, *Fortran Statements*) constants can be given symbolic names.

Stardent 1500/3000 Fortran also provides three additional constant formats, octal, hexadecimal, and Hollerith. These formats are extensions to Fortran 77. They differ from the data types in that they cannot be associated with variables, functions, or expressions. You can define up to 16 bytes for octal and hexadecimal constants. Hollerith constants may specify up to 2,000 bytes. The operations which can be performed on each of the types is discussed in the section *Expressions* later in this chapter.

1.5.1 Integer

NOTE
The IMPLICIT statement can change the type associated with the initial letter of a variable name.

The integer data types include the types BYTE, INTEGER*2, and INTEGER (also written INTEGER*4). All except INTEGER are extensions to Fortran 77. A symbolic name can be given an integer type by explicit appearance in BYTE, INTEGER*2, INTEGER, or INTEGER*4 type statement. Symbolic names beginning with the letters I, J, K, L, M, or N are implicitly given the type INTEGER unless otherwise defined; this implicit typing can be modified by the IMPLICIT statement or by the **-i4, -noi4,** and **-implicit** options.

Integer constants consist of an optional plus (+) or minus (-) sign followed by one or more digits (0 through 9). Whole numbers within the range −2 147 483 648 to +2 147 483 647 are represented as integer constants. Constants outside this range generate compile-time errors. If a value outside this range is generated during execution, an overflow may occur. Computed results may be in error. Refer to Table 1-5.

Table 1-5. Properties of Integer-Types

TYPE	RANGE	STORAGE	ALIGNMENT
BYTE	-128 - 127	8 bits/1 byte	Byte boundary
INTEGER*2	-32768 - +32767	16 bits/2 bytes	Half-word boundary
INTEGER INTEGER*4	-2147483648 - +2147483647	32 bits/4 bytes/1 word	Word boundary

There is an alternative octal notation for integer constants that may also be used. Be aware that this syntax has been included **only** for compatibility with older Fortran programs and its use is discouraged.

SYNTAX

"nn

where

nn
is a string of digits between the value of 0 through 7.

1.5.2
Real

Real numbers are approximated in Fortran by floating point numbers. These come in single-precision and double-precision forms; the first occupies one word of storage per data item and the second occupies two words. Values representable by floating point numbers are shown in Table 1-4. Double precision values should be aligned on double word boundaries.

Variables can be explicitly typed as floating point by using the following declarations:

REAL	single precision
REAL*4	single precision
DOUBLE	double precision
REAL*8	double precision

Variables starting with the letters A-H or O-Z are implicitly typed REAL*4 unless an **IMPLICIT** declaration is used or either the **-noimplicit** or the **-noi4** option to the compiler is used.

1.5.2.1 Real Constants

Real constants contain a decimal point or an exponent or both. They can have a leading plus (+) or minus (-) sign. Normally, floating point constants derive their precision from the context in which they are used and appropriate conversions are applied automatically. However, the use of **D** instead of **E** in the exponent of a floating point constant forces use of a double precision value.

SYNTAX

sn.n	*sn.n*E*se*
s.n	*s.n*E*se*
sn.	*sn.*E*se*
*sn*E*se*	
*sn.n*D*se*	*sn.*D*se*
*s.n*D*se*	*sn*D*se*

where

n is a string of digits.

s is an optional sign.

e is the exponent, which must be a integer.

The construct E*se* represents a power of 10.

EXAMPLE

`3.4E-4` $= 3.4 \times 10^{-4} = .00034$

`42.E2` $= 42 \times 10^{2} = 4200$

Double precision constants contain an optional decimal point and an exponent. They can have a leading plus (+) or minus (–) sign. The exponent is specified with the letter **D**. The construct D*se* represents a power of 10.

EXAMPLE

`14.D-5` $= 14. \times 10^{-5} = .00014$

`5.834D2` $= 5.834 \times 10^{2} = 583.4$

1.5.3 Complex

Complex numbers are represented in Fortran by one of the complex data types. Single precision complex values are represented by a pair of single precision floating point values with the real component first, and occupy two words of storage. Double precision complex values are represented by a pair of double precision floating point values with the real component first, and occupy four words of storage. The double precision complex data type is a Stardent 1500/3000 extension to Fortran 77.

A variable can be given a complex type by use of a type statement. Following are the complex data types:

COMPLEX	single precision complex
COMPLEX*8	single precision complex
DOUBLE COMPLEX	double precision complex
COMPLEX*16	double precision complex

The declaration of complex values may also be affected by use of the **IMPLICIT** statement and by the **-nor4** and **- implicit** options.

A complex constant is written as a pair of numeric constants enclosed in parentheses and separated by a comma. The precision of the constant is normally inferred from context; however, if either numeric component is a double precision floating point constant, the complex constant is also double precision.

EXAMPLES

```
(3.0, -2.5E3)          (3.5, 5.4)

(0, 0)                 (-187, -160.5)

(1.23D12, 3.4E-2)      This is double
                       precision because one
                       of its components is
                       double precision.
```

1.5.4 Logical

An entity of the **logical** type can assume only the values true or false. Entities of type logical are represented in one 32-bit word. The value false is represented by a value of 0, and the value true is represented by a nonzero value.

The forms and values of a logical constant are:

Form
.TRUE.
.FALSE.

The periods must be included as shown when specifying a logical constant.

A variable can be explicitly typed as logical by specifying it in a **LOGICAL*4** or **LOGICAL** type statement. You can also specify a variable to be of a logical type, but cause it to occupy only a half word or a byte by specifying it in a LOGICAL*2 (for half word storage) or a LOGICAL*1 (for byte storage) statement respectively.

1.5.5 Character

The character type is used to represent a string of characters. The string can consist of any characters in the 8-bit ASCII character set. Nonprintable characters can be included in a string. The blank character is valid and significant in a character entity. Lowercase characters are not identical to their uppercase equivalent in character entities. Characters otherwise difficult to write can be specified using the **CHAR** intrinsic function. (Refer to Chapter 6, *Intrinsic Functions* for additional information).

Each character in the string has a character position that is numbered consecutively: 1, 2, 3, and so on. The number indicates the sequential position of a character in the string, beginning at the left and proceeding to the right. Entities of type character are stored one character per byte (eight bits).

A variable can be explicitly typed as character by specifying it in a **CHARACTER** type statement.

The form of a character constant is a single quotation mark (apostrophe, ') followed by a nonempty string of characters and followed by a single quotation mark. When single quote marks are used to enclose a string, then double-quote marks within the string are treated as ordinary characters.

If a single quotation mark is included in a string delimited by single quotation marks, it must be written twice to distinguish it

from the delimiting characters. The length of a character constant is the number of characters between the delimiting characters (which are not counted). Double apostrophes or double double quotation marks (″ or "") count as one character each. The length of a character constant must be greater than 0.

EXAMPLES

Constant	Comments
`'Input the next item'`	Fully enclosed in apostrophes.
`'/usr/grk/lib/file1.c'`	Fully enclosed in apostrophes.
`'Item #1 =>'`	Fully enclosed in apostrophes.
`'NEEDED A "1 " OR A "2"'`	Fully enclosed in apostrophes.
`'That''s life!'`	Two apostrophes in a row indicate an apostrophe in the resultant string.
`'She replied 'Tell me another''`	The inner apostrophes enclosed by the outer apostrophes become part of the string.

1.5.6 Hollerith Constants

Hollerith constants are an extension to Fortran 77. A Hollerith constant is a typeless bit pattern specified by a character string. Hollerith constants do **not** substitute for character constants.

SYNTAX

*n*H*c*[*c*]...

where

n

is an unsigned integer (cannot be zero) that specifies how many characters are in the string.

c (and subsequent characters)

are a character string. Spaces and tabs can be used as part of the string and are included in the character count.

A maximum of 2000 characters can be specified for any Hollerith constant.

EXAMPLES

Sample	Length
`2HXY`	2
`5HTEST`	5 (a blank space is used after the constant; it is part of the constant)
`17HA LONGER CONSTANT`	17

When used with binary operators, the Hollerith constant assumes a data type that is compatible with the context in which it is used. This includes the assignment operator. For example, if a Hollerith constant is added to or assigned to an INTEGER variable, then this Hollerith constant is treated as an integer, of a length equal to the length of the integer.

EXAMPLES

`IX = JX + 4HEFAB`	If JX is INTEGER*4, then the constant is treated as INTEGER*4.
`DP = 8HXYZ12345`	If DP is REAL*8, then the constant is also treated as REAL*8.

If the length of the Hollerith constant is less than the length of the data type that is implied by its use, blanks (ASCII hex 20) are added to the right side of the constant to fill it out to correct the length to match the implied length. If the Hollerith constant is too long to be used in a particular context, Stardent 1500/3000 Fortran truncates the Hollerith constant at the right side to match the implied length. Only blanks can be truncated without causing an error.

1.5.7 Octal Constants

Octal constants are an extension to Fortran 77. An octal constant can be used where any other constant can appear.

SYNTAX

'*n*[*n*]...'**O**

where

n is any digit between the value of 0 through 7. The digit(s) are specified between apostrophe characters and are followed by an uppercase letter **O**.

An octal constant can specify up to 128 bits of data storage (16 bytes). This amounts to about 43 octal digits.

As with Hollerith constants, octal constants are treated as typeless bit patterns that take on the characteristics of their surroundings. That is, if used with integer calculations, they are treated as if specified as integers and so on (see *Hollerith Constants* for typical uses).

Unlike Hollerith constants, however, when an octal constant is found to be longer than required by the data type with which it is to interact, it is truncated from the **left** rather than from the right. Also, if it is not wide enough, zeros are added to the **left** rather than spaces to the right to pad the length of the constant to match the size. If the octal constant is too long and truncation results in removal of a nonzero digit, an error occurs.

EXAMPLES

`'377'O`	Occupies 3 bytes.
`IX = '377'O`	Extended to '00000000377'O and treated as INTEGER*4.
`CALL TEST('37777777777'O)`	Largest octal constant which can be passed because limit is size of INTEGER*4.

1.5.8 Hexadecimal Constants

Hexadecimal constants are an extension to Fortran 77. A hexadecimal constant can be used where any other constant can appear. The format of a hexadecimal constant is:

SYNTAX

'*n*[*n*]...'**X**

where

n is any digit between the value of 0 through 9 or any letter between A and F. The letters are specified between apostrophe characters, and followed by an uppercase letter **X**.

A hexadecimal constant can specify up to 128 bits of data storage (16 bytes). This amounts to 32 hexadecimal digits.

As with Hollerith constants, hexadecimal constants are treated as typeless bit patterns that take on the characteristics of their surroundings. That is, if used with integer calculations, they are treated as if specified as integers, and so on (see *Hexadecimal Constants* for typical uses).

Unlike Hollerith constants, however, when a hexadecimal constant is found to be longer than required by the data type with which it is to interact, it is truncated from the **left** rather than from the right. Also, if it is not wide enough, zeros are added to the **left** rather than spaces to the right to pad the length of the constant to match the size. If the hexadecimal constant is too long and truncation results in removal of nonzero digit, an error occurs.

EXAMPLES

```
'F4'X

IX = 'F4'X
```

Extended to '000000F4'X and treated as INTEGER*4.

`CALL TEST('FFFFFFFF'X)`	Largest hexadecimal constant which can be passed because limit is size of INTEGER*4.

1.5.9 Simple Variables

A simple variable is used for processing a single data item. It identifies a storage area that can contain only one value at a time. Subscripted variables are treated everywhere within this manual as simple variables unless stated otherwise.

EXAMPLES

```
total          sum_of_values
voltage        ERROR_FLAG1
Final_Score    i
```

1.5.10 Arrays

An array is an ordered collection of data values, all of the same type. An individual element of an array can be named and manipulated by providing subscripts which select the element. In some cases, the entire array may take part in an operation. Arrays are declared in several ways.

1.5.10.1 Array Declarators

Array declarators are used in **DIMENSION**, **COMMON**, and **type** statements to define the number of dimensions, the number of elements per dimension (called bounds), the element type, and (sometimes) the data to be stored in the elements of the array.

SYNTAX

name(*d1, d2, d3,...*)

where

name

is the symbolic name of the array.

d

is a dimension declarator. There must be one dimension declarator for each dimension in the array. The syntax of a dimension declarator is

n
or
m:n

where

m is the lower dimension bound.

n is the upper dimension bound.

If only the upper dimension bound is specified, the value of the lower dimension bound is one. The value of either dimension bound can be positive, negative, or 0; however, the value of the upper dimension bound must be greater than or equal to the value of the lower dimension bound.

NOTE
Using variables or asterisks in a dimension declarator is limited to declarators of formal arguments to subprograms. This is discussed in detail in Chapter 5, *Procedures and Block Data Subprograms.*

The lower and upper dimension bounds are integer arithmetic expressions. These expressions should not contain a user-defined function or array element reference. The upper dimension bound of the last dimension in the array declarator of a formal argument can be an asterisk. The array bounds indicate the number of dimensions of the array and the maximum number of elements in each dimension. The number of elements in each dimension is defined by $n-m+1$, where n is the upper bound and m is the lower bound.

EXAMPLES

`name(4,-5:5,6)`	Specifies a three-dimensional array. The first dimension has 4 elements, the second 11, and the third 6.
`decision_table(2,3,2,2,3,4,2)`	Specifies a seven-dimensional array.
`m(0:0)`	Specifies a one-dimensional array of one element—**m(0)**.
`list(10)`	Specifies a one-dimensional array of 10 elements—**list(1)...list(10)**.

A complete array declarator for a particular array can be used once only in a program unit, although the array name can appear in several specification statements. For example, if the array declarator is used in a **DIMENSION** statement, the array name only (no dimensions or subscripts) can be used in a **COMMON** or type statement. If the complete array declarator is used in a **COMMON** or type statement, the array must not be mentioned in a **DIMENSION** statement.

1.5.10.2 Subscripts

Subscripts designate a specific element of an array. An array element reference (subscripted variable) must contain the array name followed by as many subscripts as there are dimensions in the array. The subscripts are separated by commas and the whole list is enclosed in parentheses. Each subscript value must fall between the declared lower and upper bounds for that dimension. For example, if **A** is dimensioned **(A1:3)**, **A(I)** is a subscripted element and **I** must have a value in the range of 1 to 3. Even though the compiler does not generate an error if a subscript is outside its declared lower and upper bounds (unless the **-fullsubcheck** option is specified; refer to Chapter 7, *Compiler Options* for additional information on the **-fullsubcheck** option), the results of such a reference are unpredictable. A subscript can be any arithmetic expression.

EXAMPLES

`arr(1,2)`	Represents the element (1,2) of the array **arr**. If **arr** was declared by **arr(10,20)**, **arr** would describe a two-dimensional table, and **arr(1,2)** would describe the element in the second column of the first row.
`chess_board(i,j,k)`	Subscripts **i**, **j**, and **k** are variables which, taken together, select an element of array **chess_board**.
`arr(i+4,j-2)`	Subscripts **i+4** and **j-2** are expressions that select specific elements of array **arr** when evaluated.

1.5.10.3 Array Element Storage

The total number of elements in an array is calculated by multiplying the number of elements in each dimension. For example, the array declarator, **i(3,4,-3:5)** indicates that array **i** contains 108 elements: ((3-1)+1)×((4-1)+1)×((5–(–3)+1) = 3×4×9 = 108. The number of words of memory needed to store an array is determined by the number of elements in the array and the type of data that the array contains.

Data Type	Storage Requirement Per Element
INTEGER	4 Bytes
INTEGER*4	4 Bytes
REAL	4 Bytes
REAL*4	4 Bytes
REAL*8	8 Bytes
LOGICAL	4 Bytes
LOGICAL*4	4 Bytes
LOGICAL*2	2 Bytes
LOGICAL*1	1 Byte
COMPLEX	8 Bytes
COMPLEX*8	8 Bytes
COMPLEX*16	16 Bytes
CHARACTER	X-bytes (depends on character count)

A one-dimensional array is stored as a linear list. Arrays of higher dimensions are stored in **column major order**, with the first subscript from the left varying most rapidly; the second, next most rapidly; and so forth, with the last varying least rapidly.

EXAMPLES

The addresses expressed below are examples only.

Array declarator:

```
Integer arr(2,0:1,-5:-4)
```

Array storage:

```
arr(1,0,-5)    lowest address = 10000204
arr(2,0,-5)    address = 10000208
arr(1,1,-5)    address = 10000212
arr(2,1,-5)    address = 10000216
arr(1,0,-4)    and so on...
arr(2,0,-4)
arr(1,1,-4)
arr(2,1,-4)
```

1.5.11 Character Substrings

A character substring is a contiguous portion of a character variable.

SYNTAX

name([*first*]:[*last*])
or
a(*s1*[,*s2*]...) ([*first*]:[*last*])

where

name is a character variable name.

a(*s1*[,*s2*]...) is a character array element name.

first is any arithmetic expression that specifies the leftmost position of the substring; default value is 1.

last is any arithmetic expression that specifies the rightmost position of the substring; default value is the length of **name**.

The values of *first* and *last* must be such that $1 \le first \le last \le len$, where *len* is the length of the character variable, named constant, or array element. Notice that either *first* or *last* or both may be dropped; the colon is always required.

EXAMPLES

`name(2:5)`	If the value of name is **SUSANNA**, then **name(2:4)** specifies **USA**.
`address(:4)`	If the value of the address is **1452 NORTH**, then **address(:4)** specifies **1452**.
`city(6,2)(5:8)`	If the value of **city(6,2)** is **SAN JOSE**, then **city(6,2)(5:8)** specifies **JOSE**.
`title or title(:)`	These specify the value of the complete character variable.

1.6 Expressions

An expression represents a single value. An expression can contain constants, simple or subscripted variables, function references or combinations thereof, with operators acting on and among these elements. The result of evaluating the expression is a single value. There are four types of expressions; these are

Arithmetic
Character
Relational
Logical

Arithmetic expressions return a single value of type **BYTE**, **INTEGER*2**, **INTEGER*4**, **REAL*4**, **REAL*8**, **COMPLEX*8** or **COMPLEX*16**. Character expressions return character values. Relational and logical expressions return logical values.

1.6.1 Arithmetic Expressions

Arithmetic expressions perform arithmetic operations. An arithmetic expression can consist of a single operand, an arithmetic constant, the symbolic name of an arithmetic constant, an array element reference, or a function reference; or it can consist of two or more operands together with arithmetic operators and parentheses. The arithmetic operators are:

+	Addition; unary plus (positive or plus sign).
-	Subtraction; unary minus (negation or minus sign).
*	Multiplication.
/	Division.
**	Exponentiation.

A unary operator is one that affects one operand only. For example, the unary minus (also called a minus sign or sign of negation) negates the expression following it. There is an extension to Fortran 77 that allows a unary arithmetic operator to appear after another arithmetic operator.

EXAMPLES

```
a                 num(1)
-4. + z           a**2
3.145             (c**4)*d
SQRT(r+d)         total + sum_of_values
arr(5,2)*45.5     number_of_successes/number_of_tries*100
```

Multiplication must be specified explicitly. Fortran has no implicit multiplication that can be indicated by **a(b)** or **ab**; **a*b** must be used.

1.6.1.1 Hierarchy of Arithmetic Operators

The order of evaluation of an arithmetic expression is established by a precedence among the operators. This precedence determines the order in which the operands are to be combined. The precedence of the arithmetic operators is

**	Exponentiation	*highest*
+ ,−	Unary plus or minus	
× , /	Multiplication and division	
+ ,−	Addition and subtraction	*lowest*

Evaluation of operations within parentheses occurs first. Exponentiation precedes all arithmetic operations within an expression; multiplication and division precede addition and subtraction.

EXAMPLE

Expression: `-a**b+c*d+6`

Evaluation occurs as follows: **a**b** is evaluated to form the operand *op1*. **c*d** is evaluated to form the operand *op2*. *op1*+*op2*+**6** is evaluated to determine the value of the expression.

If an expression contains two or more operators of the same precedence, the order of evaluation is determined by the following rules:

- Two or more exponentiation operations are evaluated from right to left.
- Multiplication and division or addition and subtraction are evaluated from left to right.

EXAMPLES

Expression: `2**3**a`

Evaluation occurs as follows: **3**a** is evaluated to form *op1*. **2*****op1* is evaluated.

Expression: `a/b*c`

Evaluation occurs as follows: **a/b** is evaluated to form *op1*. *op1****c** is evaluated.

Parentheses can control the order of evaluation of an expression. Each pair of parentheses contains a subexpression that is evaluated according to the rules stated above. When parentheses are nested in an expression, the innermost subexpression is evaluated first.

EXAMPLES

Expression: `((a+b)*c)**d`

Evaluation occurs as follows: **a+b** is evaluated to form *op1*. *op1****c** is evaluated to form *op2*. *op2*****d** is evaluated.

Expression: `((b**2-4*a*c)**.5)/(2*a)`

Evaluation occurs as follows: **b**2-4*a*c** this subexpression is evaluated to form *op1*. *op1*****.5** is evaluated to form *op2*. **2*a** is evaluated to form *op3*. *op2*/*op3* is evaluated.

1.6.1.2 Expressions with Mixed Operands

BYTE, INTEGER, INTEGER*2, REAL, DOUBLE PRECISION, COMPLEX, and DOUBLE COMPLEX operands can be intermixed freely in arithmetic expressions. Before an arithmetic operation is performed, the lower type is converted to the higher type. The type of the expression is that of the highest type operand in the expression. Operand types rank from highest to lowest in the order shown in Table 1-6.

Table 1-6. Operand Rankings

COMPLEX*16	highest
COMPLEX*8	
REAL*8	
REAL*4	
INTEGER*4	
INTEGER*2	
BYTE	lowest

The conversion precedence for mixed type arithmetic expressions is specified in Table 1-7. For example, if **a** and **b** are real variables and **i** and **j** are integer variables, then in the expression **a*b-i/j**, **a** is multiplied by **b** to form *op1*; **i** is divided by **j** with integer division, converted to real, and subtracted from *op1* resulting in an expression of type real.

NOTE
An exception to the operand ranking is that if the two operands have types COMPLEX*8 and REAL*8 the result is COMPLEX*16.

Table 1-7. Conversion of Mixed Type Operands

		op2									
		I2	I4	R4	R8	C8	C16	I1	L1	L2	L4
op1	I2	I2	I4	R4	R8	C8	C16	I2	I2	I2	I4
	I4	I4	I4	R4	R8	C8	C16	I4	I4	I4	I4
	R4	R4	R4	R4	R8	C8	C16	R4	R4	R4	R4
	R8	R8	R8	R8	R8	R8	R8	R8	R8	R8	R8
	C8	C8	C8	C8	C16	C8	C16	C8	C8	C8	C8
	C16	C16	C16	C16	C16	C16	C16	C16	C16	C16	C16
	I1	I2	I4	R4	R8	C8	C16	I1	I1	I2	I4
	L1	I2	I4	R4	R8	C8	C16	I1	I1	I2	I4
	L2	I2	I4	R4	R8	C8	C16	I2	I2	2	I4
	L4	I4	I4	R4	R8	C8	C16	I4	I4	I4	I4

(*op1* operator *op2*; operators +,-,*,/,**)

I1	= BYTE
I2	= INTEGER*2
I4	= INTEGER*4
R4	= REAL*4
R8	= REAL*8
C8	= COMPLEX*8
C16	= COMPLEX*16
L1	= LOGICAL*1
L2	= LOGICAL*2
L4	= LOGICAL*4

1.6.1.3 Arithmetic Constant Expressions

An arithmetic constant expression is an arithmetic expression in which each operand is an arithmetic constant, the symbol name of an arithmetic constant, or an arithmetic constant expression enclosed in parentheses. The exponentiation operator (**) is not allowed in constant expressions unless the exponent is of type integer. Note that variable, array element, and function references are not allowed, with the following exception: as an extension to Fortran 77, the intrinsic function **ICHAR** can reference a character constant expression; this use is limited to **PARAMETER**, statement function, and executable statements. See the explanation of the **PARAMETER** statement for other extensions to Fortran 77.

1.6.2 Character Expressions

A character expression is used to represent a character string. Evaluation of a character expression produces a result of type character.

The simplest form of a character expression is a character constant, symbolic name of a character constant, character variable reference, character array element reference, character substring reference, or character function reference. More complicated character expressions can be formed by using two or more character operands together with the concatenation operator and parentheses. The concatenation operator is //.

When a concatenation operation is performed on two strings, the two strings become a single string composed of all of the characters of the first followed by all of the characters of the second

string. The length of the resulting string is the sum of the lengths of each string. For example, the value of **'FOOT' // 'BALL'** is the string **FOOTBALL**.

Parentheses have no effect on the value of a character expression. For example, the expression **'ab'//('CD'//'ef')** is the same as the expression **'ab//'CD'//'ef'**. The result of either of these expressions is **'abCDef'**.

EXAMPLES

```
char_string (5:9)
'constant string'
string1//string2//'another string'
home//'/'//filename
```

1.6.2.1 Character Constant Expressions

A character constant expression is a character expression in which each operand is a character constant, the symbolic name of a character constant, or a character constant expression enclosed in parentheses. Note that variable, array element, substring, and function references are not allowed, with the following exception: as an extension to Fortran 77, the intrinsic function **CHAR** can reference an integer constant expression; this use is limited to **PARAMETER**, statement function, and executable statements. See the description of the **PARAMETER** statement in Chapter 2, *Fortran Statements* for more information.

1.6.3 Relational Expressions

Relational expressions compare the values of two arithmetic expressions or two character expressions. Evaluation of a relational expression produces a result of type logical.

SYNTAX

op1 relop op2

where

op1 and *op2*
must both be either arithmetic expressions or character expressions.

relop

is a relational operator.

The relational operators are: .EQ. (for equal), .NE. (for not equal), .LT. (for less than), .LE. (for less than or equal to), .GT. (for greater than), and .GE. (for greater than or equal to).

Each relational expression is evaluated and assigned the logical value true or false depending on whether the relation between the two operands is satisfied (true) or not (false).

1.6.4 Arithmetic Relational Expressions

Arithmetic expressions as operands in a relational expression are evaluated according to the previously defined rules governing arithmetic expressions. If the expressions are of different types, the one with the lower rank is converted to the higher ranking type. Once the expressions are evaluated and converted to the same type, they are compared. An arithmetic relational expression is interpreted as having the logical value true if the values of the operands satisfy the relation specified by the operator. If the operands do not satisfy the specified relation, the expression is interpreted as the logical value false. Expressions of type complex or double complex can be used as operands with **.EQ.** and **.NE.** relational operators only. The concept of less than or greater than is not defined for complex numbers.

EXAMPLES

```
a  .GT.  237
b - c  .LT.  num
i - j  .GE.  z - 1
o  .GT.  p
```

1.6.4.1 Character Relational Expressions

Character relational expressions compare two operands, each of which is a character expression. The character expressions are first evaluated; then the two operands are compared character by character, starting from the left. The initial characters of the two operands are first compared. If the initial character is the same in both operands, the comparison proceeds with the second character of each operand. When unequal characters are encountered,

the greater of the two operands is determined by the greater of these two characters. Thus, the ranking of the operands is determined only by the first character position at which the two operands differ. If there is no such position, then the two operands are equal.

For example, when the two expressions **PEOPLE** and **PEPPER** are compared, the first expression is considered less than the second. This is determined by the third character **O**, which is less the **P** in the ASCII collating sequence. If the operands are of unequal length, the comparison is as if the shorter string was padded with blanks on the right to the length of the longer string.

EXAMPLES

```
IMPLICIT CHARACTER*6 (a-n)
    ! Variables beginning with A-N are this type.

'the' .LT. 'there'
'MAY 23' .GT. 'MAY 21'
name .LE. 'PETERSEN'
char_str1 .GE. char_str2
first .EQ. a_string(2:8) // 'COD'
```

1.6.5 Logical Expressions

Logical expressions produce results of type logical with values of true or false. A logical expression can consist of a single operand, a logical constant, the symbolic name of a logical constant, a logical variable, a logical array element reference, a logical function reference, or a relational expression; or it can consist of one or more operands together with the following logical operators:

.NOT.	Logical negation (unary)
.AND.	Logical **AND**
.OR.	Logical **OR**
.EQV.	Logical equivalence
.NEQV.	Logical nonequivalence (exclusive **OR**)
.XOR.	Exclusive **OR** (same as **.NEQV.**).

The unary operator **.NOT.** takes the complement (that is, the opposite) of the logical value of the operand immediately following the **operator.**

The **.AND.** operator returns a value of true only if the logical operands of the **.AND.** operator both evaluate as true.

The **.OR.** operator returns a value of true if either of the logical operands of the **.OR.** operator is true.

The **.NEQV.** operator returns a value of true only if exactly one (but not both) of the logical operands of the **.NEQV.** operator is true. As an extension to Fortran 77 , **.XOR.** and **.EOR.** can be used in place of **.NEQV.**

The **.EQV.** operator returns a value of true if the logical operands of the **.EQV.** operator are both true or both false.

A truth table for the logical operators is shown in Table 1-8.

The order of evaluation of a logical expression is established by the following precedence of the logical operators:

.NOT.	highest
.AND.	
.OR.	
.EQV., .NEQV.	lowest

Thus, **.NOT.** operations are performed before all other operations: **.EQV.**, and **.NEQV.** operations are performed after all other operations. If there is more than one operator of the same precedence, evaluation occurs from left to right.

Table 1-8. Truth Table for Logical Operators

a	b	.NOT. a	a .AND. b	a .OR. b	a .NEQV. b a .XOR. b	a .EQV. b
True	True	False	True	True	False	True
True	False	False	False	True	True	False
False	True	True	False	True	True	False
False	False	True	False	False	False	True

EXAMPLES

Expression: `a .OR. b .AND. c`

Evaluation proceeds as follows: **b .AND. c** is evaluated to form *lop1*. **a .OR.** *lop1* is evaluated.

Expression: `z .LT. b .OR. .NOT. k .GT. z`

Evaluation proceeds as follows: **z .LT. b** is computed to produce a logical value *lop1*. **k .GT. z** is computed to produce a logical value of *lop2*. **.NOT.** *lop2* is computed to produce of logical value of *lop3*. *lop1* **.OR.** *lop3* is evaluated to produce the final result. Either *lop1* or *lop3* may be evaluated first. The operands of the expression (**z, b,** and **k**) may be evaluated in any order. If the evaluation of *lop1* or *lop3* has proceeded far enough to demonstrate that the value is **.TRUE.**, the other logical subexpression need not be evaluated at all because the truth of one operand of **.OR.** is sufficient to determine that the value of **.OR.** is **.TRUE.**.

Expression: `z .AND. d .OR. lsum(q,d) .AND. p .AND. i`

Evaluation proceeds as follows: **z .AND. d** is evaluated to produce *lop1*. **lsum(q,d)** is evaluated to produce *lop2*. *lop2* **.AND. p** is evaluated to produce *lop3*. *lop3* **.AND. i** is evaluated to produce *lop4*. *lop1* **.OR.** *lop4* is evaluated to produce the final result. The values of **z, d, p, i,** and **lsum(q,d)** may be evaluated in any order. The values *lop1, lop2, lop3* and *lop4* may be produced in any order (as long as any expressions or subexpressions are evaluated first). If one operand to an **.AND.** operator is evaluated and found to have the value **.FALSE.**, the other operand need not be evaluated (because if one operand of **.AND.** is **.FALSE.**, its value is **.FALSE.**). In particular, the function call **lsum(q,d)** need never be made if, for example, **p** is **.FALSE.**. A similar discussion applies to **.OR.**.

Expression: `a .AND. (b .OR. c)`

Evaluation occurs as follows: **b .OR. c** is evaluated to form *lop1*. **a .AND.** *lop1* is evaluated.

Parentheses, as shown in the example above, can be used to control the order of evaluation of a logical expression. As with arithmetic expressions, the actual order of evaluation may be different from that stated above, but the result is the same as if these rules were followed.

1.6.6 Bit-Masking Expressions

As an extension to Fortran 77, the logical operators can be used with BYTE, INTEGER*2, and INTEGER*4 operands to perform bit-masking operations. The user must be aware of the internal binary representations of the data to use the masking operators to produce predictable results.

A complete truth table is shown in Table 1-9 . A bit-by-bit comparison is done of the operands and the corresponding bit of the expression result is set according to the truth table. Note that Stardent 1500/3000 Fortran also supplies these bit-masking operations and other bit manipulation operations as intrinsic functions. These are described in Chapter 6, *Intrinsic Functions*. These intrinsic functions comply with the MIL-STD-1753 extensions to Fortran 77.

Table 1-9. Truth Table for Masking Operators

i	j	.NOT. i	.NOT. j	i .AND. j	i .OR. j	i .NEQV. j	i .EQV. j
1	1	0	0	1	1	0	1
1	0	0	1	0	1	1	0
0	1	1	0	0	1	1	0
0	0	1	1	0	0	0	1

EXAMPLES

.AND. returns the logical product of two operands:

```
0111111111111111        op1=32767

0001011001011001        op2=5721

0001011001011001        result=5721
```

.NEQV. returns the symmetric difference of two operands.

```
0000000011111111        op1=255

0001011001011001        op2=5721

0001011010100110        result=5798
```

FORTRAN STATEMENTS

CHAPTER TWO

Statements are the fundamental building blocks of Fortran program units. This chapter describes the general form of a statement and then discusses the different categories of statements. Detailed descriptions of each statement follow in alphabetical order. Each description includes statement syntax, applicable rules, and examples.

2.1 Fortran Statement Format

A Fortran statement has the following general form:

[*label*] *statement*

The *label* identifies a particular statement so that it can be referenced from another portion of the program. A statement label consists of one to five digits placed anywhere in columns 1 through 5. Each label must be unique within a program unit; blanks and leading zeros are ignored and do not create distinct labels. Labels are optional and need not appear in numerical order.

The statement itself is written in columns 7 through 72. If a statement is too long for one line, it may be continued on the next line. This is indicated by placing some character other than a 0 (zero) or a blank in column 6 (for practical purpose, you should select a character that clearly identify your intention, perhaps, a plus sign). Columns 1 through 5 of a continuation line are ignored, except that column 1 cannot contain any one of the following characters: **$**, **C**, **!**, or * (if one of these predefined characters appears in column 1, the compiler will interpret the statement follows as a comment). Each statement can have a number of continuation lines. By using a command line option (refer to *Compiler Options* in Chapter 7), you can specify the maximum number of continuation lines that are to be allowed. The maximum you can have is 99.

NOTE
By specifying a compiler option **-extend_source** on the command line when you compile your source program, you can have Fortran statements written in columns 7 through 132. Refer to Chapter 7, *Compiler Options* for additional information.

2.2 Statement Classification

A Fortran statement can be one of two types: executable or nonexecutable. Executable statements specify action that the program is to take. Nonexecutable statements contain information such as characteristics of operands, types of data, and format specifications for input/output. Each Fortran statement is categorized in the tables below.

2.2.1 Executable Statements

ACCEPT Statement
ASSIGN Statement
Assignment Statements
BACKSPACE Statement
CALL Statement
CLOSE Statement
CONTINUE Statement
DECODE Statement
DO Statement
DO WHILE Statement
ELSE Statement
ELSE IF Statement
ENCODE Statement
END Statement
END DO Statement
ENDFILE Statement
ENDIF Statement
GOTO Statement
GOTO (assigned) Statement
GOTO (computed) Statement
IF (arithmetic) Statement
IF (block) Statement
IF (logical) Statement
INQUIRE Statement
OPEN Statement
PAUSE Statement
PRINT Statement
READ Statement
RETURN Statement
REWIND Statement
STOP Statement
TYPE Statement
WRITE Statement

2.2.2 Nonexecutable Statements

BLOCK DATA Statement
COMMON Statement
COMPLEX Statement
DATA Statement
DIMENSION Statement
ENTRY Statement
EQUIVALENCE Statement
EXTERNAL Statement
FORMAT Statement
FUNCTION Statement
IMPLICIT Statement
INCLUDE Statement
INTRINSIC Statement
NAMELIST Statement
OPTIONS Statement
PARAMETER Statement
PROGRAM Statement
SAVE Statement
Statement Function Statement
SUBROUTINE Statement
Type Statement

2.3 Statement Classification

Statements can also be grouped into six functional categories:

- Program unit statements.
- Specification statements.
- Value assignment statements.
- Control statements.
- Input/output statements.
- Program halt or suspension statements.

The statements belonging to each of these categories are shown below.

2.3.1 PROGRAM UNIT STATEMENTS

BLOCK DATA	Identifies the program unit as a block data subprogram.
END	Specifies the end of a program.
ENTRY	Provides an alternate entry into a function or subroutine.
FUNCTION	Identifies the program unit as a function subprogram.
INCLUDE	Causes the compiler to include source statements from a file.
OPTIONS	Confirms or overrides command qualifiers in a program unit.
PROGRAM	Identifies the program unit as a main program.
Statement Function	Defines a one-statement function.
SUBROUTINE	Identifies the program unit as a subroutine subprogram.

2.3.2 SPECIFICATION STATEMENTS

COMMON	Reserves a block of memory which can be used by more than one program unit.
DIMENSION	Defines the dimensions and bounds of an array.
EQUIVALENCE	Associates variables so that they share locations in memory.

EXTERNAL	Identifies external subprogram names and dummy procedure arguments.
IMPLICIT	Specifies the type associated with the first letter of a symbolic name.
INTRINSIC	Identifies intrinsic function names used as actual arguments.
NAMELIST	Identifies groups of variables to be used in namelist I/O statements.
SAVE	Retains the value of an entity after execution of a **RETURN** or **END** statement in a subprogram.
Type	Assigns an explicit type to a variable.
PARAMETER	Defines named constants.

2.3.3 VALUE ASSIGNMENT STATEMENTS

ASSIGN	Assigns a label value to a variable used in a **GOTO** or as a **FORMAT** statement label.
Assignment	Assigns values to variables at execution time.
DATA	Assigns values to variables before execution.
Type	Optionally assigns initial values to variables in addition to its primary function as a specification statement.

2.3.4 CONTROL STATEMENTS

Arithmetic **IF**	Transfers control based on a condition.
Assigned **GOTO**	Transfers control to an assigned label.
Block **IF**	Executes optional groups of statements based on one or more conditions.
CALL	Transfers control to an external procedure.
Computed **GOTO**	Transfers control based on expression evaluation.
CONTINUE	Causes execution to continue; has no effect of its own.
DO	Causes a group of statements to be executed a specified number of times.

DO WHILE	Causes a group of statements to be executed while a logical expression is true.
END DO	Terminates a DO statement.
END IF	Terminates a block IF statement.
GOTO	Transfers control to a specified statement.
Logical IF	Conditionally executes a statement based on a logical value.
RETURN	Transfers control from a subprogram back to the calling program.

2.3.5 INPUT/OUTPUT STATEMENTS

ACCEPT	Transfers data in.
BACKSPACE	Positions a file at the previous record.
CLOSE	Terminates access to a file.
DECODE	Transfers data between variables.
ENCODE	Transfers data between variables.
ENDFILE	Writes an end-of-file.
FORMAT	Describes how input/output information is arranged.
INQUIRE	Supplies information about files.
OPEN	Allows access to a file.
PRINT	Transfers data out.
READ	Transfers data in.
REWIND	Positions a file at beginning-of-file.
TYPE	Transfers data out.
WRITE	Transfers data out.

2.3.6 PROGRAM HALT STATEMENTS

PAUSE	Causes a program suspension.
STOP	Terminates program execution.

2.4 Order of Statements

The required order of statements is shown in Table 2-1. Vertical lines delineate varieties of statements that can be interspersed. For example, **DATA** statements can be interspersed with other specification statements and **PARAMETER** statements. Horizontal lines delineate varieties of statements that must not be interspersed. For example, statement function statements must not be interspersed with executable statements.

Table 2-1. Required Order Of Statements

<table>
<tr><td rowspan="7">Comment Lines and INCLUDE Statements</td><td colspan="4">OPTIONS Statement</td></tr>
<tr><td colspan="4">PROGRAM,FUNCTION,SUBROUTINE, or BLOCK DATA Statements</td></tr>
<tr><td rowspan="5">NAMELIST FORMAT and ENTRY Statements</td><td colspan="3">IMPLICIT NONE Statement</td></tr>
<tr><td colspan="2">IMPLICIT Statements</td><td rowspan="2">PARAMETER Statements</td></tr>
<tr><td rowspan="3">DATA Statements</td><td>Other Specification Statements</td></tr>
<tr><td colspan="2" rowspan="2">Statement Function Definitions</td></tr>
<tr><td>Executable Statements</td></tr>
<tr><td colspan="4">END Statement</td></tr>
</table>

Each of the following statements can appear only as the first statement in a program unit: **PROGRAM, SUBROUTINE, FUNCTION,** and **BLOCK DATA,** except that exactly one **OPTIONS** statement may precede any of these. Statements within a category are restricted as to where they can appear in a program unit. **FORMAT** and **ENTRY** statements can appear anywhere except first or last. Within the specification statements of a program unit, **IMPLICIT** statements must precede all other specification statements except **PARAMETER** statements. The last line of a program unit must be an **END** statement.

2.5 ACCEPT Statement

The **ACCEPT** statement transfers information from the standard input unit (unit 5). This statement functions identically to the formatted input **READ** statement discussed in Chapter 3, *Fortran I/O Statements*, except that it can only be used on implicitly connected logical units. It can never be used with user- specified logical units.

SYNTAX

ACCEPT *fmt, list*

where

fmt is the format designator. *fmt* must be one of the following:

- The statement label of a **FORMAT** statement.
- A variable name that has been assigned the statement label of a **FORMAT** statement.
- A character expression.
- A character or noncharacter array name that contains the representation of a format specification.
- An asterisk (indicating list-directed formatting).

list is the list of variables that specifies where the data is to be transferred. The list may contain implied **DO** loops. For syntax and detailed information on implied **DO** loops, refer to *Implied DO Loops* under *DO Statement* in Chapter 2, *Fortran Statements*.

2.6 ASSIGN Statement

The ASSIGN statement assigns a statement label to an integer variable.

SYNTAX

ASSIGN *label* **TO** *variable*

where

label is a statement label.

variable is an integer variable .

The statement label can only be that of an executable statement or a **FORMAT** statement. The variable defined as a label can then be used in an assigned **GOTO** statement or as the format specifier in an input/output statement.

A variable must be defined with a statement label value when referenced in an assigned **GOTO** statement or as a format identifier in an input/output statement. While defined with a statement label value, the variable must not be referenced in any other way. An integer variable defined with a statement label value may be redefined with the same or a different statement label value or with an integer value.

EXAMPLES

```
      ASSIGN 200 TO nexttest
      GOTO nexttest
```

The variable **nexttest** is assigned the statement label **200**. The label is that of an executable statement.

```
      ASSIGN 75 TO currentfmt
C ...  (more statements)
75    FORMAT (4I5,F8.2)
C ...  (more statements)
      READ(5, currentfmt) test1, test2, test3, test4, width
```

The variable **currentfmt** is assigned the statement label **75**. The label is that of a **FORMAT** statement.

2.7 Assignment Statement

The assignment statement evaluates an expression and assigns the resulting value to a variable. There are three kinds of assignment statements:

- Arithmetic
- Logical
- Character

2.7.1 Arithmetic Assignment Statement

An arithmetic assignment statement computes the value of its right hand side expression according to the normal rules of expression evaluation. That value is then stored in the variable named on the left hand side of the statement. This causes the variable to become defined. If the types of the variable and the expression differ, Table 2-2 describes the conversion that is applied to the expression's value before it is stored.

SYNTAX

var = expr

where

var is a variable or array element of one of the following types:

BYTE
INTEGER*2
INTEGER or INTEGER*4
REAL or REAL*4
DOUBLE PRECISION or REAL*8
COMPLEX or COMPLEX*8
DOUBLE COMPLEX or COMPLEX*16

expr is an arithmetic expression.

EXAMPLES

```
total = subtotal1 + subtotal2
```

Defines the value of **total** as the value of **subtotal1 + subtotal2.**

```
array(i) = interest_rate(i) * months
```

Defines the i-th element of the array as the value **interest_rate(i)** multiplied by **months.**

Table 2-2. Conversion Rules - Assignment Statements

Variable or Array Element (Y)	Expression (D)				
	Integer or Logical	REAL	REAL*8	COMPLEX	COMPLEX*16
Integer or Logical	Assign D to Y	D = INT(Y)	D = INT(Y)	Y = INT(REAL(D))	Y = INT(REAL(D))
REAL	Y = REAL(D)	Assign D to Y	Y = REAL(D)	Y = REAL(D)	Y = REAL(D)
REAL*8	Y = DBLE(D)	Y = DBLE(D)	Assign D to Y	Y = DBLE(D)	Y = DBLE(D)
COMPLEX	Y = (REAL(D),0)	Y = (D,0)	Y = (REAL(D),0)	Assign D to Y	Y = (REAL(D), AIMAG(D))
COMPLEX*16	Y = (DBLE(D), 0)	Y = (DBLE(D), 0)	Y = (D,0)	Y = (DBLE(REAL(D)), DBLE(AIMAG(D)))	Assign D to Y

2.7.2 Logical Assignment Statement

A logical assignment statement computes the value of its right hand side expression according to the normal rules of expression evaluation. That value is then stored in the variable named on the left hand side of the statement. This causes the variable to become defined. If the types of the variable and the expression differ, Table 2-2 describes the conversion that is applied to the expression's value before it is stored.

SYNTAX

lvar = lexp

where

lvar is a variable element of type logical.

lexp is a logical expression.

EXAMPLES

```
LOGICAL log1
i = 10
Log1 = i .EQ. 10
```

log1 is assigned the value *true* because **i** equals **10**.

```
LOGICAL log_res, flag_set
num = 100
flag_set = .TRUE.
Log_res = num .GT. 200 .AND. flag_set
```

log_res is assigned the value *false* because **num** is not greater than **200**.

2.7.3 Character Assignment Statement

A character assignment statement computes the value of its right hand side expression according to the normal rules of expression evaluation. That value is then stored in the variable named on the left hand side of the statement. This causes the variable to become defined. Both the variable and the expression must be of character type.

SYNTAX

cvar = cexp

where

cvar is a variable, array element, or substring of type character.

cexp is a character expression.

If the length of the variable is greater than the length of the expression, the value of the expression is left-justified in the variable and blanks are placed in the remaining positions. If the length of the variable is less than the length of the expression, the value of the expression is truncated from the right until it is the same length as the variable.

EXAMPLES

```
CHARACTER*6 name
CHARACTER*4 effects(6), index
name = 'CAUSES'
```

The variable **name** is assigned the character string **CAUSES**.

```
k = 'fire'
effects(2) = k
```

The second element of the array **effects** is assigned the character string **fire**.

```
effects(3) = name(3:5)
```

The third element of the array **effects** is assigned the character string **USE** followed by a blank.

```
CHARACTER*22 member
member = 'ED JOINER'
```

The variable **member** is assigned the value **ED JOINER** with 13 blanks (that is, 22 - 9 blanks) on the right.

2.8 BACKSPACE Statement

The **BACKSPACE** statement positions a sequential file or device at the preceding record, and is allowed only for sequential files.

SYNTAX

BACKSPACE *unit*

or

BACKSPACE ([UNIT=] *unit* **[, IOSTAT=***ios***] [, ERR=***label***])**

where

unit is an arithmetic expression (0 or positive) specifying a unit.

ios is an integer variable or array element (must be **INTEGER*4**) for error code return (refer to *Appendix A* for **IOSTAT** error codes).

label is the statement label of an executable statement in the same program unit as the **BACKSPACE** statement, to which control will transfer upon error.

The *unit* is backspaced one record. If the *unit* is already positioned at the beginning of the file, there is no effect. If either **IOSTAT** or **ERR** is specified, control returns to the **BACKSPACE** statement in the event of error; otherwise the program's execution is aborted. The variable named by *ios* will be given the value of zero unless an error occurs; then the value will be non-zero and may be interpreted using *Appendix A*. Control normally passes to the statement following the **BACKSPACE** statement; however, if an error occurs and an error label is specified, control transfers to that label upon completion of the **BACKSPACE** statement.

EXAMPLES

```
BACKSPACE 10
```

The sequential file connected to unit 10 is backspaced one record.

```
BACKSPACE (UNIT=k+3, IOSTAT=j, ERR=100)
```

The file connected to unit **k+3** is backspaced one record. If an error occurs, control transfers to statement **100** and the error code is stored in the variable **j**.

If the file pointer is positioned at the beginning of the file, a **BACKSPACE** statement has no effect upon the file.

2.9 BLOCK DATA Statement

The **BLOCK DATA** statement is the first statement in a block data subprogram.

SYNTAX

BLOCK DATA [*name*]

where

name is an optional subprogram name.

The *name* must not be the same as an external procedure, the main program, a common block, or any other block data subprogram in the same execution program, nor may it be the same as any local name in this block data subprogram.

Block data subprograms provide initial values for variables and array elements in labeled common blocks. (Blank common variables cannot be initialized). The **BLOCK DATA** subprogram *name* must not conflict with other subprogram or common block names. Refer to Chapter 5, *Procedures and Subprograms* for more information on block data subprograms.

2.10 CALL Statement

The **CALL** statement transfers control to a subroutine.

SYNTAX

CALL *name*[([*arg1*[, *arg2*[, *arg3*...]]])]

where

name is the name of the subroutine being referenced, or the name of a dummy procedure.

arg is an actual argument or an asterisk followed by a label, where the label is a statement label of an executable statement in the same program unit as the **CALL** statement.

When a **CALL** statement is executed, any expressions in the actual argument list are evaluated, and then control is passed to the subroutine. For a normal return from the subroutine, execution continues with the statement following the **CALL** statement. When an alternate return is taken, execution continues with the statement label in the actual argument list that corresponds to the return number specified in the subroutine's **RETURN** statement. Subroutine subprograms, referencing subroutines, and alternate returns from a subroutine are all discussed in detail in Chapter 5, *Procedures and Subprograms*.

EXAMPLES

```
      CALL print_forms(top, lh, rh)
```

The subroutine **print_forms** is called. Three arguments are passed.

```
      CALL exit
```

The subroutine **exit** is called. No arguments are passed.

```
      CALL checktaxable(m, n, cost, *35)
C      (more statements)
35    withtax = cost*1.07
C      (more statements)
      END
      SUBROUTINE checktaxable(j, k, w, *)
C      (more statements)
      RETURN 1
C      (more statements)
      END
```

The subroutine **checktaxable** is called. Three arguments are passed. ***35** means that the return point is the statement labeled **35** if the subroutine executes the alternate return (**RETURN 1**).

2.11 CLOSE Statement

The CLOSE statement terminates the connection of a file to a unit.

SYNTAX

CLOSE ([UNIT=] *unit* [, IOSTAT=*ios*][, ERR=*label*][, STATUS=*stat*])

where

unit is an integer expression (0 or positive) specifying a unit number.

ios is an integer variable or array element (must be INTEGER*4) for error code return (refer to *Appendix A* for IOSTAT error codes).

label is the statement label of an executable statement in the same program unit as the CLOSE statement. If an error occurs during the execution of the CLOSE statement, control is transferred to the specified statement rather than aborting the program.

stat is a character expression that determines the disposition of the file; *stat* is one of the following:

'KEEP'	The file continues to exist after the execution of the CLOSE statement. 'KEEP' is the default; that is, specifying STATUS='KEEP' and not specifying the STATUS= parameter have the same effect.
'DELETE'	The file does not exist after the execution of the CLOSE statement. The STATUS specifier has no effect on scratch files, as scratch files are always deleted on CLOSE or normal program termination.

A CLOSE statement must contain a unit number and at most one each of the other options. The CLOSE statement need not be in the same program unit as the OPEN statement that connected a file to the specified unit. If a CLOSE statement specifies a unit that does not exist or has no file connected to it, no action occurs. The CLOSE statement is discussed in detail in Chapter 4, *File Handling*.

If either **IOSTAT** or **ERR** is specified, control returns to the **CLOSE** statement in the event of error. The variable named by *ios* will be given the value of zero unless an error occurs; then the value will be nonzero and may be interpreted using *Appendix A*. Control normally passes to the statement following the **CLOSE** statement; however, if an error occurs and an error label is specified, control transfers to that label upon completion of the **CLOSE** statement.

EXAMPLES

```
CLOSE(10)
```

The file connected to unit **10** is disconnected. The file continues to exist.

```
CLOSE(UNIT=6, STATUS='DELETE')
```

The file connected to unit **6** is disconnected. The file no longer exists.

```
CHARACTER*6 cstat
cstat = 'DELETE'
CLOSE(UNIT=6, STATUS=cstat)
```

This produces the same results as the preceding example.

```
CLOSE(5, IOSTAT=io_error,ERR=100)
```

The file connected to unit **5** is disconnected and kept. If an error occurs, control is transferred to statement **100**, and the error code is stored in the variable **io_error**.

2.12 COMMON Statement

The COMMON statement specifies a block of storage space which may be used by more than one program unit.

SYNTAX

COMMON [/[*blkname*] /]*list1* [[,] / [*blkname2*] / *list2*[,]...]

where

blkname is the name of a labeled common block. Each omitted *blkname* specifies blank common.

list is one or more simple variables, array names, or array declarators.

In each **COMMON** statement, the variables following a block name are declared to be in common block **blkname**. If the first block name is omitted, all variables that appear in the first list are specified to be in blank common. Alternatively, the appearance of two slashes with no block name between them declares the variables that follow to be in blank common.

The following data items must not appear in a **COMMON** statement:

- The names of dummy arguments in a subprogram.
- A function, subroutine, entry, statement function, or intrinsic function name.
- A name used in a **DATA** statement, except in block data subprograms.

A variable cannot be specified more than once in the **COMMON** statements within a program unit and all common block names must be distinct from subprogram names.

EXAMPLES

```
COMMON a,b,c
```

The variables **a**, **b**, and **c** are placed in blank common.

```
COMMON pay, time /color/ red
```

The variables **pay** and **time** are placed in blank common; the variable **red** is placed in common block **color**.

```
COMMON /a/ a1, a2 // x(10), y /c/ d
```

The variables **a1** and **a2** are placed in common block **a**; **x** and **y** are placed in blank common; and **d** is placed in common block **c**.

Any common block name or blank common specification may appear more than once in one or more **COMMON** statements in a program unit. The variable list following each successive appearance of the same common block name is treated as a continuation of the list for that block name. For example, the **COMMON** statements:

```
COMMON a, b, c /x/ y, z, d // w, r
COMMON /cap/ hat,visor // tax /x/ o, t
```

are equivalent to the following **COMMON** statements:

```
COMMON a, b, c, w, r, tax
COMMON /x/ y, z, d, o, t /cap/ hat, visor
```

The length of a common block is determined by the number and type of the variables in the list associated with that block.

```
INTEGER*2 b(4)
COMMON /blk1/ b, arr(3)
```

The common block **blk1** takes 5 words of storage, **b** uses 2 (half a word per element) and **arr** uses 3 (1 word per element).

Common block storage is allocated at link time. It is not local to any one program unit. Data space within the common area for arrays **b** and **arr** shown in the preceding example is allocated as shown in Figure 2-1:

Word	Common Block
1	**b(1)**
	b(2)
2	**b(3)**
	b(4)
3	**arr(1)**
4	**arr(2)**
5	**arr(3)**

Figure 2-1. Sample COMMON block storage

Each program unit which uses the common block must include a **COMMON** statement that contains the block name (if a name was specified). The list assigned to the common block by the program unit need not correspond by name, type, or number of elements with those of any other program unit. The size of an unlabeled (blank) common block can differ between program units, but the size of a labeled common block should be the same in all program units. For example, the following appears in program unit 1:

```
INTEGER*2 i,j
COMMON /blocka/ i(4), j(6), ivory(2), sam
INTEGER*2 ivory
```

and the **COMMON** statement

```
INTEGER*2 india, jack
COMMON /blocka/ geo, m(10), india, jack
```

appears in program unit 2. **blocka** is the same size (7 words) in both program units. Thus, referencing **i(4)** in program unit 1 is equivalent to referencing **m(2)** in program unit 2, because both variables refer to the same half word of the named common block. The correspondence between the variables in common in the two program units is shown in Figure 2-2.

The following example shows an unnamed, or blank, common block in program unit 2 that is a different size from the common block referenced in program unit 1:

Program 1 Reference	Common Block Word Number	Program 2 Reference
i(1)	1	geo
i(2)		
i(3)	2	m(1)
i(4)		m(2)
j(1)	3	m(3)
j(2)		m(4)
j(3)	4	m(5)
j(4)		m(6)
j(5)	5	m(7)
j(6)		m(8)
ivory(1)	6	m(9)
ivory(2)		m(10)
sam	7	india
		jack

Figure 2-2. Typical Data References in Named COMMON

Program unit 1:

```
INTEGER*4 i
COMMON i(12)
```

Program unit 2:

```
INTEGER*4 law
COMMON law(7)
```

The correspondence between the variables in blank common in the two program units is shown in Figure 2-3.

Program 1 Reference	Common Block Word Number	Program 2 Reference
i(1)	1	law(1)
i(2)	2	law(2)
i(3)	3	law(3)
i(4)	4	law(4)
i(5)	5	law(5)
i(6)	6	law(6)
i(7)	7	law(7)
i(8)	8	Unused
i(9)	9	Unused
i(10)	10	Unused
i(11)	11	Unused
i(12)	12	Unused

Figure 2-3. Typical Data References in Blank COMMON

2.13 CONTINUE Statement

The CONTINUE statement creates a reference point in a program unit.

SYNTAX

CONTINUE

The CONTINUE statement should always be written with a label; it marks a point in the program where a label is needed but the programmer does not want to associate the label with any specific action.

In Fortran, the CONTINUE statement is usually the last statement in a labeled DO loop that otherwise would end in a prohibited statement such as a GOTO. If a CONTINUE statement appears elsewhere in a program, or if it is not labeled, it performs no function and control passes to the next statement.

EXAMPLE

```
      DO 45 i = 1, 10
   32 r = r + 5
      z = SQRT(r)
      PRINT *, z
      IF (r .LT. 42.) GOTO 45
      GOTO 32
   45 CONTINUE
```

Because the last useful statement in the loop is a GOTO statement, a CONTINUE statement is used to terminate the loop.

2.14 DATA Statement

The **DATA** statement assigns initial values to variables before execution begins.

SYNTAX

DATA *varlist1/conlist1/*[[,]*varlist2/conlist2* /[,]...]

where

varlist is one or more simple variable names, array names, array element names, substring names, or implied **DO** loops. For syntax and detailed information on implied **DO** loops, refer to *Implied DO Loops* under *DO Statement* later in this chapter.

conlist is the list of constants that are to be assigned to the corresponding items in *varlist*. The form of an item in *conlist* is:

[*num**]*con*

where

num is an integer or integer named constant; the default is 1.

* is the repeat specifier.

con is a constant or named constant that is to be repeated *num* times.

There must be the same number of items in each *varlist* and its associated *conlist*. There is a relationship between the items in *varlist* and the items in *conlist*, and there are rules that apply to conversion of data types. Some of these conversions include extensions to Fortran 77.

If the entities in both lists are numeric data types, the following conversion rules apply:

- The constant value is converted to the data type being initialized.
- The number of digits which may be assigned when using octal or hexadecimal constants depends on the data type of the data item being assigned. The constant is extended on

the left and filled with zeros if it contains fewer digits than the capacity of the variable or array element. The constant is truncated on the left if the constant contains more digits than can be stored. The use of octal or hexadecimal constants is an extension to Fortran 77.

If the entities in both lists are character data types, the following conversion rules apply:

- When the constant contains fewer bytes than the length of the entity, the constant has the rightmost character positions initialized with spaces.
- The constant is truncated on the right when it contains more bytes than the length of the entity.

If the constant has a numeric data type and the entity is character data type, the following rules apply and are an extension to Fortran 77:

- The character entity must have a length of one character.
- The constant must have a value in the range of 0 through 255 and must be either an integer, octal or hexadecimal constant.

If the constant is a Hollerith or character, and the entity is numeric, the following rules apply, and are an extension to Fortran 77:

- The constant is extended on the right with spaces if it contains fewer characters than the entity.
- The constant is truncated on the right if it contains more characters than can be stored.

EXAMPLES

```
DATA a, b, c, d, /3.0, 3.1, 3.2, 3.3/
```

Data loading any part of a variable or part of a **COMMON** block causes the entire thing to be data loaded. The following example of a code fragment illustrates this concept.

```
REAL a(1000,1000)
DATA a(10,20) /1.1/
```

In the above example all one million elements of "**a**" are loaded.

2.15 DECODE Statement

The DECODE statement uses format specifiers to transfer data between variables that are in internal storage, and then translates the data from character to internal form. The syntax of the statement is

SYNTAX

DECODE (*c, i, r,* [, IOSTAT=*ios*] [, ERR=*label*]) [*list*]

where

c is an integer expression that is the number of characters to be translated to internal form.

i is a format identifier.

r is any variable, array element, character substring, constant, expression, or array name reference that contains the characters to be translated to internal form.

ios is an integer variable or array element. If no error has occurred, *ios* = 0; otherwise *ios* is not zero and indicates an error condition.

label is a statement number of an executable statement.

list is an input/output list that receives the data after translation to internal form.

This statement has been included only for compatibility with older Fortran programs and its use is discouraged. Internal read and write statements, with internal units, should be used instead of ENCODE and DECODE.

2.16 DO Statement

DO loops are used to control iteration in Fortran; each **DO** loop causes some operation to be executed repeatedly. **DO** statements control groups of other statements for repeated execution. Implied **DO** loops appear in input/output statements and in **DATA** statements and control transmission of data to and from variables. In all **DO** loops, a standard **DO** loop control is used.

2.16.1 DO Loop Controls

A **DO** loop control controls the execution of a **DO** loop; either an explicit loop or an implied **DO** loop.

SYNTAX

DO [*label*[,]]*index* = *init*, *limit*[, *step*]

where

label is the statement label of an executable statement. In a labeled **DO** loop (refer to *Labeled DO Loops* later in this section), this statement must follow the **DO** statement in the sequence of statements within the same program unit as the **DO** statement.

index is a simple integer or floating point variable.

init, limit, step
are each arithmetic expressions.

EXAMPLE

Let **t1**, **t2**, and **t3** be temporary variables of the same type as the variable *index*. (Note that **t3** must not evaluate to zero). Let **count** be a temporary variable of type integer. Then prior to the first iteration of the loop controlled by this **DO** control, it is as if the following code were executed (notice that the assignments may require conversions as specified in Table 2-2).

```
t1 = init
index = t1
t2 = limit
if step exists
   t3 = step
else
   t3 = 1
```

```
        count= INT((limit - init + step)/step)
        count = max(count, 1)
        GOTO  10

<label>     <body of loop>
        index = index + t3
        count = count - 1
    10  if (count .GT. 0)  go to label
C   next operation
```

Modification of the variable *index* during the execution of the controlled operations is an error. The value of the variable *index* may be referenced and used within the controlled operations it will contain a terminal value upon a normal exit from the bottom of the loop.

NOTE
If the **NOF77** switch of the **OPTIONS** statement is on during compilation, every loop will be executed at least once no matter what the value of *count* is.

When a **DO** loop *index* has a floating point type, the final value of the *index* may so somewhat surprising. Index values are usually computed by the addition shown above; floating point addition is subject to rounding. The number of loop trips, however, is controlled by the computed *count*. Because of the rounding, the *index* may be less than, equal to, or greater than the value of *limit* at termination.

DO loops controls are effected by the use of certain compiler options. When the **-onetrip** option is used, all **DO** loops are guaranteed to execute at least once. For additional information, refer to Chapter 7, *Compiler Options*, or Chapter 10, *User Commands* in this manual.

2.16.2 Labeled and Block DO Statements

The labeled and block **DO** statements control execution of groups of statements by causing the statements to be repeated a specified number of times. The repeated statement or group of statements is known as the *range* of the **DO** loop.

SYNTAX

```
        DO  label  DO-loop-control
            controlled statements
label   statement

          or

        DO  DO-loop-control
            controlled statements
        END DO
```

Strictly, the controlled statements, the terminating statement of the labelled **DO** statement, and the **END DO** statement are not part of the syntax of the **DO** statement, but they must go together and so are shown together. The *index* variable of the **DO** statement must not be the index of any enclosing **DO** statement, because that would be a redefinition of the index, prohibited by the rules for **DO** controls. A block **DO** is an extension of Fortran 77. Each block **DO** must have its own **END DO**. The range of any **DO** statement may be empty.

A labeled **DO** loop begins with a **DO** statement that specifies the label of the terminating statement of the loop. The terminating statement of a labeled **DO** loop must follow the **DO** statement. It must **not** be one of the following:

- An unconditional **GOTO** statement
- An assigned **GOTO** statement
- An arithmetic **IF** statement
- Any of the four statements associated with the block **IF** statement:
    - An **IF THEN** statement
    - An **ELSE** statement
    - An **ELSE IF** statement
    - An **ENDIF** statement
- A **RETURN** statement
- A **STOP** statement
- An **END** statement
- Another **DO** statement
- Any nonexecutable statement

The terminating statement of a labeled **DO** loop may be a logical **IF** statement. A labeled **DO** loop may be terminated with an **END DO** statement. As in all labeled **DO** loops, this terminating **END DO** statement must have a label, which must match the label of the **DO** statement. (A **DO** loop terminated with an unlabeled **END DO** statement is a block **DO** loop.)

2.16.3 Proper Nesting

If a **DO** statement appears within the range of another **DO** statement, within a **THEN** or **ELSE** block, or within the range of a **DO WHILE** statement, the inner **DO** statement must have its termination statement within the same outer **DO** statement range, **THEN** or **ELSE** block, or **DO WHILE** statement range. The terminal statement of the inner **DO** statement may be the same as that of the outer **DO** statement. If any part of a block **IF** statement appears in a **DO** statement range, the entire block **IF** statement must be within that range.

2.16.4 Transfers of Control

Control may transfer from inside the range of a **DO** statement to outside the range. Control may transfer from outside the range of a **DO** statement into the range of a **DO** loop **only if**

- a previous transfer took control out of the loop, and
- the index of the loop was not modified.

Statements executed in this way are in the extended range of the **DO** statement; this is an extension of Fortran 77.

If a statement is the common termination of several **DO** statements, then transfer of control to that statement may only be made from the innermost range; otherwise, a prohibited transfer to an inner range has occurred.

Labeled DO Loop Examples

```
      DO 100 i = 1, 10
C       (more statements)
  100 CONTINUE
```

The group of statements terminating with the one labeled **100** is repeated ten times.

```
      DO 200 j = 1, 10, 2
C       (more statements)
  200 IF (a(j) .EQ. 0) STOP
```

The group of statements terminating with the one labeled **200** is repeated five times. The final value of **j** is 11.

```
      DO 300 r = 1.0, 2.0, .1
C       (more statements)
  300 END DO
```

The group of statements ending with the one labeled **300** is executed 11 times. Although this loop ends with an **END DO** statement, it is not considered a block **DO** loop. Notice that the label in the **DO** statement corresponds with the one in the **END DO** statement.

NOTE
Use of block **DO** loops, instead of the equivalent labeled **DO** loops, can sometimes generate more efficient object code.

Block DO Loop Examples

```
      DO j = 10, 1, -2
C       (more statements)
      END DO
```

The group of statements terminating with the **END DO** statement is executed five times.

```
      DO j = 10, 1, 2
C       (more statements)
      END DO
```

The group of statements terminating with the **END DO** statement is not executed. (The **DO** loop is entirely skipped unless the *-onetrip* option is in effect.)

Loop Index Modification Example

```
      DO 10 i = 1, 10, 2
         WRITE (6,'(''i='',I2)')i
         i = i - 2
   10 CONTINUE
```

The modification of the loop index in the body of the loop is erroneous. The execution of the loop is unpredictable, although many loops will execute as many times as if the modification had not taken place. It is generally impossible for the compiler to warn off this error, and it usually does not attempt to do so.

The value of the control variable, upon normal completion of the **DO** loop, is defined to be the next value assigned as a result of the incrementation. For example, in the loop:

```
      DO 20 i = 1,5
C       (more statements)
   20 CONTINUE
```

the value of **i** after normal completion of the loop is 6. And, in the loop

```
      DO 55 i = 1,10,3
C       (more statements)
   55 CONTINUE
```

the value of **i** after normal completion of the loop is 13.

The value of the control variable from the loop, retains its value at the time of any premature exit.

2.16.5 Implied DO Loops

Implied **DO** loops are found in input/output statements (**READ**, **WRITE**, and **PRINT**) and in **DATA** statements. An implied **DO** loop contains a list of data elements to be read, written, or initialized, and a **DO** control. Inner implied loops can use the indexes of outer loops. An index from an outer implied **DO** loop control may be used in the control expressions of an inner implied **DO** control. For **DATA** statements, only integer index variables and expressions can be used.

2.16.6 Implied DO Loops in Input/Output Statements

The implied **DO** loop acts like a labeled or block **DO** loop. The range of the implied **DO** loop is the list of elements to be input or output. The list may have any elements in it that would otherwise be allowed in the context of the implied **DO** loop. Each iteration of the implied **DO** loop causes the list to be traversed and the data items to be transferred.

SYNTAX

(*list*, DO loop control)

where

list is a input/output list. It may contain other implied **DO** loops.

EXAMPLES

```
PRINT *, (a,i = 1,3)
```

Write the value of **a** three times. If **a** is 35.6, the output consists of one record as follows:

```
35.6 35.6 35.6
```

If the list of an implied **DO** contains several simple variables, each of the variables in the list is read or written for each pass through the loop. For example, the statement

```
READ *, (a,b,c,j = 1,2)
```

is equivalent to

```
READ *,a,b,c,a,b,c
```

An implied **DO** loop can also transmit arrays and array elements. For example:

```
DIMENSION b(10)
PRINT *, (b(i), i = 1,10)
```

causes the array **b** to be written in the following order:

```
b(1) b(2) b(3) b(4) b(5) b(6) b(7) b(8) b(9) b(10)
```

If an unsubscripted array name is used in the list, the entire array is transmitted. For example, the result of the following statements:

```
DIMENSION x(3)
PRINT *, (x,i=1,2)
```

is to write the elements of array **x** twice as follows:

```
x(1) x(2) x(3) x(1) x(2) x(3)
```

The list can contain expressions that use the index value. For example:

```
DIMENSION a(10)
PRINT *, (i*2,a(i*2), i=1,5)
```

The preceding prints out the numbers **2, 4, 6, 8, 10** alternating with array elements **a(2), a(4), a(6), a(8), a(10).**

Implied **DO** loops can be nested. Nested implied **DO** loops follow the same rules as other nested **DO** loops. For example, the statement:

```
WRITE(6,*)((a(i,j),i=1,2),j=1,2)
```

produces the following output:

```
a(1,1) a(2,1) a(1,2) a(2,2)
```

Implied **DO** loops are useful for controlling the order in which arrays are transferred. These two examples print an array **a**, dimensioned as **a(2,3)**, with values:

```
1 3 5
2 4 6
```

The statement:

```
WRITE(6,'(3I2)') a
```

prints the array in column-major order, yielding:

```
1 2 3
4 5 6
```

and the statement:

```
WRITE(6,'(3I2)')((a(i,j),j=1,3),i=1,2)
```

prints the array in row-major order, yielding:

```
1 3 5
2 4 6
```

Notice that the following statements both produce the same result:

```
WRITE(6,'(3I2)')((a(i,j),i=1,2),j=1,3)
WRITE(6,'(3I2)') a
```

Implied DO loops in input/output statements are not just used with arrays. The following statement prints a table of degrees and the sine of each, in steps of 10 degrees.

```
WRITE(6,'(I3,F9.5)')(i,SIN(i*3.14159/180.),i=0,360,10)
```

The statement produces the following output:

```
  0    0.00000
 10    0.17365
 20    0.34202
 30    0.50000
 40    0.64279
 50    0.76604
 60    0.86602
 70    0.93969
 80    0.98481
 90    1.00000
100    0.98481
110    0.93969
120    0.86603
130    0.76605
140    0.64279
150    0.50000
160    0.34202
170    0.17365
180    0.00000
190   -0.17365
200   -0.34202
210   -0.50000
220   -0.64279
230   -0.76604
240   -0.86602
250   -0.93969
260   -0.98481
270   -1.00000
280   -0.98481
290   -0.93969
300   -0.86603
310   -0.76605
320   -0.64279
330   -0.50000
340   -0.34202
350   -0.17365
360   -0.00001
```

2.16.7 Implied DO Loops in Data Statements

The syntax of a **DATA** statement containing an implied **DO** loop is

SYNTAX

DATA (*dlist, DO control*)/ *clist*/

where

dlist is a list of array element names and implied **DO** loops.

clist is the list of constants to be assigned to the corresponding items in *dlist*.

The implied **DO** loop in a **DATA** statement acts like the implied **DO** loop in an input/output statement. It is executed at compilation time to initialize a variable list. The index can be used in expressions for subscript values or position specifiers of character substrings. Inner implied **DO** loops can use the indexes of outer loops. The iteration count found in an implied **DO** loop in a **DATA** statement must be positive.

EXAMPLES

```
DATA a, b,(vector(i), i=1,10), k /2.5, -1.0, 10*0.0, 999/
DATA ((matrix(i,j), i=0,5), j=5,10) /36*-1/
```

This example initializes parts of a one-dimensional character array:

```
PROGRAM char
CHARACTER*5 char_array(5)
DATA (char_array(1),i=1,5)/5*'xxxxx'/
WRITE (6,'(A)') (char_array(1) (1:i), i=1,5)
END
```

The program produces this output:

```
x
xx
xxx
xxxx
xxxxx
```

The following example initializes a square array to the identity matrix (its main diagonal contains all ones, and the rest of the array zeros). It uses a **WRITE** statement with an implied **DO** loop to output the array in row order.

```
PROGRAM MAIN
DIMENSION id_array(10, 10)
DATA ((id_array(i,j), j = i+1,10), i = 1,9) /45*0/     ! upper
DATA (id_array(i,i), i = 1,10)              /10*1/     ! diagonal
DATA ((id_array(i,j), i = j+1,10), j = 1,9) /45*0/     ! lower
WRITE(6,'(10I2)') ((id_array(i,j), j = 1,10), i = 1,10)
END
```

The program produces this output:

```
 1 0 0 0 0 0 0 0 0 0
 0 1 0 0 0 0 0 0 0 0
 0 0 1 0 0 0 0 0 0 0
 0 0 0 1 0 0 0 0 0 0
 0 0 0 0 1 0 0 0 0 0
 0 0 0 0 0 1 0 0 0 0
 0 0 0 0 0 0 1 0 0 0
 0 0 0 0 0 0 0 1 0 0
 0 0 0 0 0 0 0 0 1 0
 0 0 0 0 0 0 0 0 0 1
```

2.16.8 DO WHILE Statement

As a MIL-STD-1753 standard extension to Fortran 77, the **DO WHILE** statement controls execution of a group of statements by causing the statements to be repeated while a logical expression is true. The **DO WHILE** construct is an important element of structured programming.

SYNTAX

DO [*label*[,]] **WHILE** (*logical expression*)
  [*controlled statements*]
[*label*] **END DO**

Each **DO WHILE** loop must be terminated by a separate **END DO** statement, which does not require a label. Note that if the **DO WHILE** statement uses the label option, the **END DO** statement that terminates the **DO** loop must have a label, and the labels must match.

A **DO WHILE** operates like this: the logical expression is evaluated and tested at the beginning of the **DO WHILE** loop. If the expression evaluates to true, the group of statements between

the **DO WHILE** and the corresponding **END DO** statement, referred to as the range of the **DO WHILE** loop, is executed. Control then returns to the logical expression for another test and possible iteration. If the logical expression evaluates to false, the **DO WHILE** loop terminates and execution continues with the statement following the **END DO** statement. The rules for transfers into the range of a **DO WHILE** loop are the same as for other **DO** loops.

EXAMPLES

```
      DO WHILE (i .NOT. 999)
        READ(5,33) i
C      (more statements)
      END DO
```

The example above repeatedly reads input until entry of a terminating flag (**999** in this case).

The next example repeatedly increments **index** while the condition of the **DO WHILE** statement is true.

```
      index = 1
      DO WHILE (array(index) .NE. value .AND. index .LE. limit)
         index = index + 1
      END DO
```

2.16.9 Nesting DO Loops

DO loops can contain other **DO** loops. This is called "nesting." The only restriction is that each level (that is, each nested loop) must be completely contained within the surrounding loop. In a nested **DO** loop, the last statement of an inner (nested) loop must either be the same as, or occur before, the last statement of the outer loop. (For programming clarity, you should always use a separate terminating statement for each loop.)

EXAMPLE

Here is an example in which the terminating statement of the innermost loop occurs before the last statement of the preceding loop. The two outer loops have the same terminating statement.

```
      DO 100 i = 1, 10
        DO 100 j = 1, 10
          sum = 0
          DO 90 k = 1, 10
   90 sum = sum + a(i,k)*b(k,j)
        c(i,j) = sum
  100 CONTINUE
```

Control passes to the statement following statement **100** only after all iterations of all three loops are executed.

In a block **DO** or **DO WHILE** loop, each level must be terminated with an **END DO** statement.

NESTED EXAMPLE

```
      DO WHILE (x .GE. 0)
C      (more statements)
        DO WHILE (y .LT. 10)
C         (more statements)
        END DO
      END DO
```

DO loops may be nested as long as the range of statements in any **DO** loop does not overlap the range of the preceding loop.

The following example shows an illegal construction, one in which the ranges of two loops overlap.

```
      DO 100 i = 1,10
      DO 500 j = 1,10          Range of first loop
  100 x(i) = i**2              Range of second loop
  500 z(j) = j**6
```

2.16.10 Ranges of DO Loops

The range of the **DO** loop is defined as the first statement following the **DO** statement up to and including the terminating statement.

EXAMPLES

This example shows the range of a labeled **DO** loop:

```
      a = 6
      DO 20 i = 1,10
        b = SQRT(a)          -----+
        WRITE (6,200) b           +---- Range of the DO loop
        a = a + 1                 +
   20 CONTINUE               -----+
```

This example shows the range of a block **DO** loop. It produces the same results as the preceding example.

```
      a=6
      DO i = 1, 10
        b = SQRT(a)        ------+
        WRITE(6, 200) b           +----  Range of the DO loop
        a = a + 1                 +
      END DO               ------+
```

This example shows the range of a **DO WHILE** loop:

```
      DO WHILE (i .NOT. 999)
       READ (5,33) i       ------+
C       (more statements)          +----  Range of the DO loop
      END DO               ------+
```

A **DO** loop can be exited at any time. Normal exit is accomplished by completion of the **DO** loop and continuation with the statement following the termination statement of the loop. A **DO** loop may be exited prematurely; for example, a **GOTO** statement may transfer control out of the loop.

This example searches a list for a keyword. If the keyword is found, control passes out of the **DO** loop to the statement labeled **60**. If the keyword is not found, the loop terminates normally with the **STOP** statement.

```
      DO 50 i = 1,n
   50   IF (list(i) .EQ. keyword) GOTO 60
      STOP 'Not found.'
   60 PRINT *, 'Match at', i
```

It is illegal to transfer control into the range of a **DO** loop from outside the range, for example by a **GOTO** statement. In the following example, the statement **GOTO 20** is illegal.

```
      DO 50 i = 1, 10
   20   n = n + 1
   50 CONTINUE
      GOTO 20
```

In the following example, the statement **GOTO 10**, which is inside the logical **IF** statement, is illegal because control is transferred into the inner loop from outside its range. (At first glance, this may seem permissible, because statement **10** terminates both loops. This is because the **CONTINUE** statement is within the range of the inner loop. You can avoid this problem by using a separate terminating statement for each nested loop.)

```
      DO 10 i = 1, n
         IF (i .EQ. k) GOTO 10
         DO 10 j = 1, m
            a(i,j) = b(i,j)
10 CONTINUE
```

2.17 ENCODE Statement

The ENCODE statement uses format specifiers to transfer data between variables that are in internal storage, and then translates the data from internal to character form. The syntax of the statement is

SYNTAX

ENCODE (*c*, *i*, *r*, [, IOSTAT=*ios*] [, ERR=*label*]) [*list*]

where

c is an integer expression that is the number of bytes to be translated to character form.

i is a format identifier.

r is any variable, array element, character substring, constant, expression, or array name reference that receives the characters, after translation to external form.

ios is an integer variable or array element. If no error has occurred, *ios* = 0; otherwise *ios* is not zero and indicates an error condition.

label is a statement number of an executable statement.

list is an input/output list that contains the data to be translated to character form.

The **ENCODE** statement has been included only for compatibility with older Fortran programs and its use is discouraged. Internal read and write statements, with internal units, should be used instead of **ENCODE** and **DECODE**.

2.18 END Statement

The END statement indicates the end of a program unit, that is, the end of a program, subroutine, function, or block data subprogram.

SYNTAX

END

If an **END** statement is executed in a subprogram, it has the same effect as a **RETURN** statement. If an **END** statement is executed in a main program, execution of the program is terminated. An **END** statement can be labeled, and it can be continued if the word **END** is all on one line. It must be the last statement in a program unit.

EXAMPLES

```
PROGRAM xtest
READ (5,*) a,b
IF (a .LT. b) a = b
PRINT *, a, b
END
```

The **END** statement terminates program **xtest**.

2.19 ENDFILE Statement

The **ENDFILE** statement writes an end-of-file record to the specified sequential file or device.

SYNTAX

```
   ENDFILE unit
or
   ENDFILE ( [UNIT=]unit [ ,IOSTAT=ios ][ ,ERR=label] )
```

where

unit is the unit number of a sequential file.

ios is an integer variable or array element for error code return (refer to *Appendix A* for **IOSTAT** error codes).

label is the statement label of an executable statement. It is in the same program unit as the **ENDFILE** statement. If an error occurs during the execution of the **ENDFILE** statement, control is transferred to the specified statement.

An end-of-file record in a disk file must be the last record. After execution of an **ENDFILE** statement, the file is positioned beyond the end-of-file record. Some devices (magnetic tape units, for example) may have multiple end-of-file records, with or without intervening data records. An end-of-file record cannot be written to a direct access file.

EXAMPLES

```
ENDFILE 10
```

An end-of-file record is written to the file connected to unit **10**.

```
ENDFILE (UNIT=12, IOSTAT=j, ERR=100)
```

An end-of-file record is written to the file connected to unit **12**. If an error occurs, control is transferred to statement **100** and the error code is stored in variable **j**.

2.20 ENTRY Statement

The ENTRY statement provides an alternate name, argument list, and starting point for a function or subroutine. It can appear only in a subroutine or function subprogram (not in a main program or block data subprogram).

SYNTAX

ENTRY *name* [([*arg1*, *arg2*, *arg3*,...])]

where

name is the name of the external entry.

arg is a dummy argument. *arg* can be a variable name, dummy procedure name, or an asterisk. An asterisk is permitted only in a subroutine.

The dummy arguments in an ENTRY statement can differ in order, number, type, and name from the dummy arguments in the FUNCTION statement, SUBROUTINE statement, or other ENTRY statements. However, for each call to the subprogram through a given entry point, only the dummy arguments of that entry point can be used. If no dummy arguments are listed after a particular ENTRY statement, no arguments are passed to the subprogram when a call to that ENTRY name is made. The ENTRY statement name cannot appear as a variable in any statement prior to the ENTRY statement, except in a type statement in a function subprogram. In particular, an ENTRY name can not be a dummy argument for any other function, subroutine, or ENTRY statement not can it be the name of or appear in any COMMON block.

An ENTRY statement can appear anywhere in a subprogram after the FUNCTION or SUBROUTINE statement, with the exception that the ENTRY statement must not appear between a block IF statement and its corresponding END IF statement or between a DO statement and the end of its DO loop.

A subprogram can have zero or more ENTRY statements. An ENTRY statement is considered a nonexecutable statement. If control falls into an ENTRY statement, it is treated as an unlabeled CONTINUE statement; that is, control falls through to the next statement.

EXAMPLES

```
      SUBROUTINE linka(d, i, f)
C      (more statements)
      ENTRY search(table, f)
              .
              .
      END
```

search defines an alternate entry point into subroutine **linka**.

```
      CHARACTER*10 FUNCTION compose(word, sent, para)
C       (statements)
      ENTRY search(document)
C      (more statements)
      RETURN
```

The **ENTRY** statement allows an alternate way of entering the function **compose**.

The following example shows a function with entries of different types.

```
      REAL FUNCTION f(x)
      INTEGER k, i
C      (more statements)
      ENTRY k(i)
C      (more statements)
      END
```

Here is an example that creates a stack of integers. It shows the use of **ENTRY** to group the definition of a data structure together with the code that accesses it, a technique known as "encapsulation".

NOTE
This subroutine, **manipulate_stack**, cannot be called directly and control can not flow through the top, randomly into the first entry point.

```
      SUBROUTINE manipulate_stack
      IMPLICIT NONE
      INTEGER size, top, value
      PARAMETER (size = 100)
      INTEGER stack(size)
      SAVE stack, top
      DATA top /0/

      STOP 'Manipulate stack'    ! Prohibits direct calls to
                                 ! manipulate_stack
C     Push value onto the stack

      ENTRY push(value)
      IF (top .EQ. size) STOP 'Stack overflow'
      top = top + 1
      stack(top) = value
      RETURN
```

```
C     Pop the top of the stack and place in value

      ENTRY pop(value)
      IF (top .EQ. 0) STOP 'Stack underflow'
      value = stack(top)
      top = top - 1
      RETURN

      END
```

Here are examples of **CALL** statements associated with the preceding example:

```
CALL push(10)
CALL push(15)
CALL pop(i)
CALL pop(j)
```

2.21 EQUIVALENCE Statement

The **EQUIVALENCE** statement associates variables so that they share the same storage space.

SYNTAX

EQUIVALENCE (*list1*) [,(*list2*), . . .]

where

list is two or more simple variables, array elements, array names, or character substrings. All items in each list entry share the same storage space.

Subprogram names, common block names, block data names, namelist group names, dummy arguments, and program names must not appear in an **EQUIVALENCE** statement. Each array or substring subscript must be an integer constant expression.

The types of equivalenced data items can be different. The **EQUIVALENCE** statement does not cause type conversion or imply mathematical equivalence. If an array and a variable share the same storage space through the **EQUIVALENCE** statement, the array does not have the characteristics of a variable and the variable does not have the characteristics of an array. They only share the same storage space.

Care should be taken when data types of different sizes share the same storage space because the **EQUIVALENCE** statement specifies that each data item in a list has the same first storage unit. For example, if an integer and a double precision real value share the same storage space, the integer value shares the same space as the most significant word of the two-word double precision real value.

EXAMPLE

```
EQUIVALENCE (a,b),(c(2),d,e)
```

The variables **a** and **b** share the same storage space; **c(2)**, **d**, and **e** share the same storage space.

2.21.1 Equivalence of Array Elements

Array elements can share the same storage space with elements of a different array or simple variables. For example:

```
DIMENSION a(3), c(5)
EQUIVALENCE (a(2), c(4))
```

specifies that array element **a(2)** shares the same storage space as array element **c(4)**. This implies that:

- **a(1)** shares storage space with **c(3)** and **a(3)** shares storage space with **c(5)**.
- No equivalence occurs outside the bounds of any of the arrays.

The storage space for the above two arrays is shown in the following example:

Array **a**	Storage Space Word Number	Array **c**
	1	**c(1)**
	2	**c(2)**
a(1)	3	**c(3)**
a(2)	<-----4----->	**c(4)**
a(3)	5	**c(5)**

If the arrays are not of the same type, they may not line up element by element. For example, the statements:

```
REAL*4 a
INTEGER*2 ibar
DIMENSION a(2), ibar(4)
EQUIVALENCE (a(1), ibar(1))
```

produce the following storage space allocation:

Array **a**	Storage Space Word Number	Array **ibar**
a(1)	1	**ibar(1)**
		ibar(2)
a(2)	2	**ibar(3)**
		ibar(4)

If only an array name appears in an **EQUIVALENCE** statement, it has the same effect as using an array element name that specifies the first element of the array. Specifying **EQUIVALENCE (a,ibar)** instead of **EQUIVALENCE (a(1),ibar(1))**, in the above example, would produce the same results.

It is illegal to specify that the same storage space be occupied by more than one element of the same array. The following example is illegal because it specified the same storage space for **a(1)** and **a(2)**:

```
DIMENSION a(2)
EQUIVALENCE (a(1),b),(a(2),b)
```

An **EQUIVALENCE** statement must not specify that consecutive array elements are to be noncontiguous. For example:

```
REAL a(2), r(3)
EQUIVALENCE (a(1), r(1)), (a(2), r(3))
```

is prohibited because the **EQUIVALENCE** statement specifies that **r** is noncontiguous.

2.21.2 Equivalence Between Arrays of Different Dimensions

To determine equivalence between arrays with different dimensions, Fortran contains an internal array successor function that views all elements of an array in linear sequence. Each array is stored as if it were a one-dimensional array. Array elements are stored in ascending sequential column-major order. The leftmost index varies the fastest, then the second leftmost, then the third, and so on.

EXAMPLES

i(-2:4) The elements of **i** are stored in the following order:

```
i(-2) i(-1) i(0) i(1) i(2) i(3) i(4)
```

t(2,3) The elements of **t** are stored in the following order:

```
t(1,1) t(2,1) t(1,2) t(2,2) t(1,3) t(2,3)
```

k(2,2,3) The elements of **k** are stored in the following order (as read left to right, top to bottom by row):

```
k(1,1,1) k(2,1,1) k(1,2,1) k(2,2,1)
k(1,1,2) k(2,1,2) k(1,2,2) k(2,2,2)
k(1,1,3) k(2,1,3) k(1,2,3) k(2,2,3)
```

The number of words each element occupies depends on the type of the array. For example, these statements:

```
INTEGER*2 i
DIMENSION a(2,2), i(4)
EQUIVALENCE (a(2,1), i(2))
```

produce the following storage space allocation:

Array a	Storage Space Word Number	Array i
a(l,l)	1	i(1)
a(2,1)	2	i(2)
a(1,2)	3	i(3)
a(2,2)	4	i(4)

2.21.3 Equivalence of Character Variables

As an extension to Fortran 77, character and noncharacter data items may share the same storage space. For example, these statements:

```
INTEGER i(5)
CHARACTER*16 c
EQUIVALENCE (i, c)
```

produce the following storage space allocation:

Array i	Storage Space Word Number	Variable c
i(1)	**1**	**c(1:4)**
i(2)	**2**	**c(5:8)**
i(3)	**3**	**c(9:12)**
i(4)	**4**	**c(13:16)**
i(5)	**5**	Unused

Another example shows sharing of the same storage space in which variables do not begin and end on word boundaries. These statements:

```
REAL a(3)
CHARACTER*16 c
EQUIVALENCE (a, c(4:4))
```

produce the following storage space allocation:

Array a	**Storage Space** Word Number	Variable c
Unused	**1**	1 Unused byte, **c(1:3)**
a(1)	**2**	**c(4:7)**
a(3)	**3**	**c(8:11)**
a(4)	**4**	**c(12:15)**
Unused	**5**	3 unused bytes, **c(16:16)**

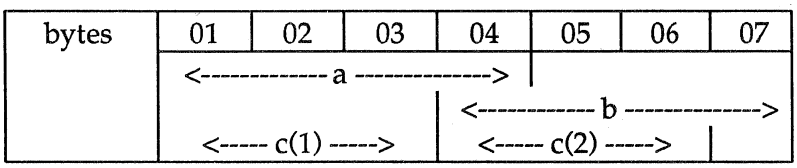
The Stardent 1500/3000 hardware requires data items to be allocated on their "natural" storage boundaries or else hardware operation faults are generated during program execution. For example, the byte address of an INTEGER*4 variable should be a multiple of 4 and the byte address of a double precision floating point quantity should be a multiple of 8. It is possible, through the use of **EQUIVALENCE** statements, to force data items off the natural boundaries normally allocated by the compiler. When this occurs, the compiler issues an error message. Good programming practice suggests that all data be naturally aligned. Stardent 1500/3000 Fortran guarantees that all **COMMON** blocks begin on a double-word boundary and that all local variables, in the absence of **EQUIVALENCE**, are correctly aligned.

The lengths of the data items that share the same storage space are not required to be the same. An **EQUIVALENCE** statement specifies that the storage sequences of the character data items whose names are specified in the list have the same first character

storage unit. This causes the association of the data items in the list and can cause association of other data items. Any adjacent characters in the associated data items can also have the same character storage unit and thus can also be associated. In the example

```
CHARACTER*4 a, b
CHARACTER*3 c(2)
EQUIVALENCE (a,c(1)), (b,c(2))
```

the association of **a**, **b** and **c** can be illustrated this way:



2.21.4 Equivalence in Common Blocks

Data elements can be put into a common block by specifying them as equivalent to data elements mentioned in a **COMMON** statement. If one element of an array shares the same storage space, through the **EQUIVALENCE** statement, with a data element within a common block, the whole array is placed in the common block with equivalence maintained for storage units preceding and following the data element in common. The common block may be extended on the right when it is necessary to fit an array that shares storage space into the common block.

No array can share storage space with a common block, however, if storage elements would have to be prefixed to the common block to contain the entire array. Equivalences cannot insert storage into the middle of the common block or rearrange storage within the block.

Because the elements in a common block are stored contiguously according to the order in which they are mentioned in the **COMMON** statement, two elements in common cannot be made to share the same storage space through the **EQUIVALENCE** statement.

EXAMPLE

In the following example, array **i** is in a common block and array element **j(2)** is equivalent to **i(3)**:

```
INTEGER*4 i(6), j(6)
COMMON i
EQUIVALENCE (i(3), j(2))
```

The common block is extended to accommodate array **j** as follows:

Array i	Common Block Word Number	Array j
i(1)	**1**	
i(2)	**2**	**j(1)**
i(3)	**3**	**j(2)**
i(4)	**4**	**j(3)**
i(5)	**5**	**j(4)**
i(6)	**6**	**j(5)**
	7	**j(6)**

The equivalence set up by the following example is **not** allowed:

```
DIMENSION i(6), j(6)
COMMON i
EQUIVALENCE (i(1), j(2))
```

To set array **j** into the common block, an extra word must be inserted in front of the common block. Element **j(1)** would be stored in front of the common block; thus **EQUIVALENCE (i(1), j(2))** is not allowed.

2.22 EXTERNAL Statement

The EXTERNAL statement identifies a name as a subprogram name and permits the name to be used as an actual argument in subprogram calls.

SYNTAX

EXTERNAL *proc1* [,*proc2*,...]

where

proc is the name of a subprogram. Each name can appear once only in each **EXTERNAL** statement and in at most one **EXTERNAL** statement in each program unit.

The **EXTERNAL** statement provides a means of using the names of subroutine subprograms and function subprograms as actual arguments. The **EXTERNAL** statement is necessary to inform the compiler that these names are subprograms or function names, not variable names. Whenever a subprogram name is passed as an actual argument, it must be placed in an **EXTERNAL** statement in the calling program. A dummy argument that is a procedure argument should also appear in an **EXTERNAL** statement.

If an intrinsic function name appears in an **EXTERNAL** statement, the compiler assumes that a user subprogram by that name exists; the intrinsic function is not available to that program unit. A name cannot appear in both an **EXTERNAL** and **INTRINSIC** statement. A statement function name must not appear in an **EXTERNAL** statement.

EXAMPLES

```
      EXTERNAL b1
      CALL sub(a, b1, c)
C      (more statements)
      END

      SUBROUTINE sub(x, y, z)
      EXTERNAL y
      z = y(z)
      RETURN
      END
```

The **EXTERNAL** statement declares **b1** to be a subprogram name. The call to **sub** passes the values of **a** and **c** and passes the name of the subprogram **(b1)**.

The reference to **y** causes **b1** to be called.

```
PROGRAM my_sin
EXTERNAL sin
REAL sin, x, y
READ(5,*) y
x = sin(y)        ! This call is to a user-written
WRITE(6,*) x      ! function named sin, not to the
END               ! intrinsic function sin.
```

2.23 FORMAT Statement

The FORMAT statement describes how input and output information is to be arranged.

SYNTAX

label **FORMAT** (*item*[,*items*,. . . .])

where

label is a statement label.

item is a format item.

A **FORMAT** statement is a non-executable specification statement. It may appear anywhere in a program unit subject to the rules specified in *Order Of Statements* earlier in this chapter. The label on a **FORMAT** statement may not be the target of any transfer of control.

The allowed format descriptors are summarized in Table 2-3 and the edit descriptors are summarized in the next table. A more detailed explanation of the **FORMAT** statement is presented in Chapter 3, *Fortran I/O Statements*. Each of the format descriptors can be preceded by a repeat specifier (**4I7**, for example).

Table 2-3. Format Descriptors

Descriptor	Data Type
A*w*	character or Hollerith
D*w.d*	real, double precision, complex
E*w.d*[E*e*]	real, double precision, complex
F*w.d*	real, double precision, complex
G*w.d*[E*e*]	real, double precision, complex
I*w*[.*m*]	decimal integer
O*w*[.*m*]	octal
L*w*	logical
Z*w*[.*m*]	hexadecimal

where

w is an integer specifying field width.

d is an integer specifying the number of digits to the right of the decimal point.

e is an integer specifying the number of digits in the exponent.

m is an optional integer specifying the minimum number of digits on output.

Table 2-4. Edit Descriptors

Descriptor	Function
BN	Ignore blanks
BZ	Treat blanks as zeros
*k***P**	Scale factor
*n***H***c*	Hollerith editing
*n***X**	Skip *n* positions
T*c*	Skip to column *c*
TL*c*	Skip *c* positions to the left
TR*c*	Skip *c* positions to the right
S	Processor determines sign output
SP	Output optional plus signs
SS	Inhibit optional plus sign output
/	Begin new record
:	Terminate format if list empty
'...'	Literal editing
$	Suppress newline at end of output

EXAMPLE

```
10 FORMAT(I3, 5F12.3)
```

The specification is for an integer number with a field width of 3, and five real numbers with a field width of 12 and three significant digits to the right of the decimal point.

2.24 FUNCTION Statement

The FUNCTION statement identifies a program unit as a function subprogram.

SYNTAX

[*type*] FUNCTION *name*[*m] ([*arg1*, *arg2*,...])

where

type is the type of the function (see *Type Statement* later in this chapter).

name is the name of the function (if *type* is not specified, the name is typed in the same manner as variables are).

**m* is an unsigned, nonzero integer constant that specifies the length of the data type, and must be a valid length specifier for the data type of *type*.

arg is a dummy argument of the function.

The dummy arguments in a **FUNCTION** statement can be used as

- Variables
- Array names
- Subprogram names

The dummy arguments should be of the same type as the actual arguments that are passed to the function from the calling program unit. If a dummy argument of type character has a length of (*) declared, the dummy argument assumes the length of the associated actual argument for each reference of the function.

If the function is of type **character*(*)**, it assumes the length declared for it by the calling program.

Function subprograms are discussed in detail in Chapter 5, *Procedures and Subprograms*.

EXAMPLES

```
FUNCTION comp()
```

Defines a single precision floating point function, **comp**, with no arguments.

```
INTEGER FUNCTION timex(a,b,k)
```

Defines an integer function, **timex**, with three arguments.

```
CHARACTER*6 FUNCTION namex(1)
```

Defines a character function, **namex**, whose value is six characters long, with one argument.

2.25 GOTO Statement

The GOTO statement transfers control to a labeled statement in the same program unit. There are three kinds of GOTO statements:

- Unconditional GOTO
- Computed GOTO
- Assigned GOTO

The statement can be written GOTO or GO TO. All GOTO statements transfer control to other executable statements within the same program unit. Any labels mentioned in any GOTO statement must be attached to an executable statement within the same program unit. Some transfers of control into or out of other control constructs are illegal; see the sections on the DO statement and the IF statement.

2.25.1 Unconditional GOTO Statement

The unconditional GOTO statement transfers control to the specified statement.

SYNTAX

GOTO *label*

where

label is the label of an executable statement.

An unconditional GOTO statement causes immediate transfer of control to the labeled executable statement.

EXAMPLE

```
GOTO 20
```

Control is passed to the statement labeled 20 when the GOTO statement is executed. Statement 20 can be before or after the GOTO statement, but must be present in the same program unit.

2.25.2 Computed GOTO Statement

The computed **GOTO** statement transfers control to one of several statements depending on the results of the evaluation of an expression.

SYNTAX

GOTO (*label1*, *label2*, . . .)[,] *exp*

where

label is the label of an executable statement. The same label can appear more than once.

exp is an arithmetic expression.

The computed **GOTO** statement passes control to one of several labeled statements depending on the result of an evaluation. The index expression *exp* is evaluated and truncated to an integer value. The index selects the statement label in the label list. For example, if the index is 1, control passes to the statement whose label appears in the first position of the list of labels. If the index value is 2, the second label in the list is used, and so on. If *exp* evaluates to less than 1 or to a value greater than the number of labels in the label list, control is passed to the statement following the computed **GOTO**.

EXAMPLES

```
i = 3.0
GOTO (30,60,50,100),i
```

Because **i** has a value of 3, control passes to statement **50**.

```
b = 1.5
z = 1
GOTO (10,20,40,40) b + z
```

Because **INT(b + z) = 2**, control passes to statement **20**.

2.25.3 Assigned GOTO Statement

The assigned **GOTO** statement transfers control to the statement whose label was most recently assigned to the variable in the **GOTO** statement by the execution of an **ASSIGN** statement.

SYNTAX

GOTO *ivar* [[,] (*label1, label2,. . .*)]

where

ivar is an **INTEGER*4** simple variable.

label is the label of an executable statement. The list of labels is optional.

ivar must be given a label value of an executable statement through an **ASSIGN** statement prior to execution of the **GOTO** statement. When the assigned **GOTO** statement is executed, control is transferred to the statement whose label matches the label value of *ivar*. The list of labels has no effect and there is no requirement that the **GOTO** transfer to one of the labelled statements.

EXAMPLES

```
ASSIGN 10 TO my_age
GOTO my_age
```

Control is transferred to statement **10** when the **GOTO** statement is executed.

```
ASSIGN 100 TO new_time
GOTO new_time, (90,100,150)
```

Control is transferred to statement **100** when the **GOTO** statement is executed.

2.26 IF Statement

The IF statement provides a means for decision making. There are three types of IF statements:

- Arithmetic IF
- Logical IF
- Block IF

An arithmetic IF statement transfers control to one of three labeled statements depending on whether an expression evaluates to a negative, zero, or positive value. A logical IF statement causes execution of a statement if an evaluated expression is true. A block IF causes execution of one of a set of blocks of statements depending on the value of one or more logical expressions.

2.26.1 Arithmetic IF Statement

The arithmetic IF statement transfers control to one of three statements. The form of the arithmetic IF statement is

SYNTAX

IF (*exp*) *labeln*, *labelz*, *labelp*

where

exp	is an arithmetic expression.
labeln *labelz* *labelp*	are the statement labels of executable statements.

When an arithmetic statement is executed, *exp* is evaluated. If the resulting value is negative, control passes to the statement whose label is *labeln*. If the evaluated value is 0, control passes to statement *labelz*. If the value is positive, then control passes to the statement whose label is *labelp*.

If the arithmetic expression is of complex or double complex type, it is converted to a single or double precision floating point value using the rules of Table 2-2 before the branch is taken. The effect is to branch on only the real part of the complex number.

EXAMPLES

```
IF (a + b) 10, 20, 30
```

Control is passed to **10**, **20**, or **30**, depending on whether the value of **a+b** is negative, zero, or positive.

```
testa = 0
IF (testa) 50,100,50
```

Because **testa** equals 0, control passes to statement **100**.

```
i = 10
j = -(15)
IF (i+j) 10, 20, 30
```

Because **i + j** is negative, control passes to statement **10**.

```
IF (timex) 60,60,60
```

Control passes to statement **60** regardless of the value of **timex**.

```
z = 10
a = 60
IF (a+z) 100, 100, 60
```

Because **a + z** is positive, control passes to statement **60**.

As illustrated in the examples, two of the labels in the label list may be the same; control branches to one of two possible statements. In fact, all of the labels in the list may be the same and control branches to the statement bearing this label regardless of the results of the evaluation.

In many cases, logical IF statements can be substituted for arithmetic IF statements and produce more readable code. For example:

```
      IF (a+b) 10, 10, 20
10    x = SQRT(y**2 + z**2)
20 . . .
```

can be written

```
IF (a+b .LE. 0.0)  x = SQRT( y**2 + z**2)
```

with no need for either label, one less statement, and a cleaner structure. Older programs may especially suffer from these problems and profit from the improvement.

2.26.2 Logical IF Statement

The logical **IF** statement evaluates a logical expression and executes one statement if the result of the evaluated expression is true.

SYNTAX

IF (*exp*) *statement*

where

exp is a logical expression.

statement is any executable statement other than a **DO**, **END DO**, **END**, block **IF**, or another logical **IF** statement.

The logical **IF** statement is a two-way decision maker. If the logical expression contained in the **IF** statement is true, then the statement contained in the **IF** statement is executed and control passes to the next statement. If the logical expression is false, then the statement contained in the **IF** statement is not executed and control passes to the next statement in the program.

EXAMPLES

```
a = b
IF (a .EQ. b) GOTO 100
```

Because the expression **a .EQ. b** is true, control passes to statement **100**.

```
IF (p .AND. q) res = 10.5
```

If **p** and **q** are both true, the value of **res** is replaced by **10.5**; otherwise, the value of **res** is unchanged.

2.26.3 Block IF Statement

The block IF statement is an extension of the logical IF statement, allowing one of a collection of blocks of statements to be executed depending on the truth value of several logical expressions.

SYNTAX

```
IF (exp1) THEN
    [statement(s)]
[ELSE     [IF(exp2) THEN
              [statement(s)]
[ELSE
    [statement(s)] ] ]
ENDIF
```

where

exp is a logical expression.

statement is any executable statement or a format statement.

ENDIF terminates the block IF. One ENDIF is required for each nesting level.

One block IF statement can contain any number of ELSE IF sub-blocks, but only one ELSE sub-block.

A block IF statement must always begin with an IF-THEN statement followed by a block of controlled statements. This first section may be followed by zero or more instances of ELSE-IF-THEN statements and their following controlled statements. There may always be an optional ELSE statement at the end followed by its controlled statements. Finally, the entire grouping must end with an ENDIF statement. Any of the groups of controlled statements may be empty.

Execution of the block IF proceeds in the following manner. The logical expression in the IF statement is evaluated. If it is TRUE, the associated controlled statements are executed. If control is not otherwise transferred, then upon completion of the controlled statements, control passes to the statements following the ENDIF. If the expression evaluates to FALSE, then attention is turned to the first ELSE IF (if there is one). The logical expression is evaluated. If its value is TRUE, the controlled statements are executed just as for the IF-THEN statements. If its value is FALSE, control flows to the next ELSE-IF, the ELSE, or the ENDIF. Each ELSE-IF follows the same pattern. If control ever reaches the ELSE statement, its controlled statements are executed. Finally, if control ever reaches the END-IF, control simply flows to the next statement.

EXAMPLES

```
x = y
IF (x .EQ. y) THEN
   x = x+1
ENDIF
```

Because **x = y**, the value of **x** is replaced by the value of **x+1**. Note that this is equivalent to the logical **IF** statement:

```
IF (x .EQ. y) x = x+1
```

```
IF (x .LT. 0) THEN
  y = SQRT(ABS(x))
  z = x + 1 - y
ELSE
  y = SQRT(x)
  z = x - 1
ENDIF
```

If **x < 0**, one block of code is executed; if **x >= 0**, a separate block of code is executed.

```
IF (n(i) .EQ. 0) THEN
  n(i) = n(j)
  j = j + 1
  IF (j .LT. k) THEN
    k = k - 1
  ELSE IF (j .EQ. k) THEN
    k = k + 1
  ENDIF
ELSE
  m = i
  k = n(i)
ENDIF
```

2.27 IMPLICIT Statement

The IMPLICIT statement overrides or confirms the type associated with the first letter of a name.

SYNTAX

IMPLICIT *type*(*range1*,[*range2*, ...]) [,*type*(*range1*,[*range2*, ...])]

where

type is the type to be associated with the corresponding letter or list of letters in range (see *Type Statement* later in this chapter).

range is either a single letter or a range of letters (for example, **A- Z**, or **I-N**) to be associated with the specified type. Writing a range of letters has the same effect as writing a list of single letters.

An **IMPLICIT** statement specifies a type for all variables, arrays, named constants, function subprograms, **ENTRY** names in function subprograms, and statement functions that begin with any letter that appears in the **IMPLICIT** statement, and that are not explicitly given a type; it does not change the type of any intrinsic functions.

The **IMPLICIT** statement itself can be overridden for specific names when these names appear in a type statement. For example, **IMPLICIT INTEGER (A)** specifies that symbolic names starting with the letter **A** are type integer. A type statement such as **REAL ABLE**, however, indicates that the variable **ABLE** is type real, overriding the **IMPLICIT** statement. Uppercase and lowercase letters are equivalent in arguments to the **IMPLICIT** statement. Thus **IMPLICIT INTEGER (Q)** and **IMPLICIT INTEGER (q)** are the same.

An explicit type specification in a **FUNCTION** statement overrides an **IMPLICIT** statement for the function name. Note that the length is also overridden when a particular name appears in a **CHARACTER** or **CHARACTER FUNCTION** statement.

As a MIL-STD-1753 extension to Fortran 77, if **IMPLICIT NONE** is specified, inherent typing is disabled and all variables, arrays, named constants, function subprograms, **ENTRY** names, and statement functions (but not intrinsic functions) must be explicitly typed. The **IMPLICIT NONE** statement must be the only

IMPLICIT statement in the program unit. Types of intrinsic functions are not affected.

Within the specification statements of a program unit, **IMPLICIT** statements must precede all other specification statements, except **PARAMETER** statements. The **IMPLICIT NONE** statement should always be used in a good structured program.

The same letter should not appear as a single letter, or be included in a range of letters more than once in all of the **IMPLICIT** statements in a program unit.

EXAMPLE

```
IMPLICIT COMPLEX(i,j,k), INTEGER(a-c)
```

All variables and function names beginning with **i**, **j**, or **k** are of type **COMPLEX**; names beginning with **a**, **b**, or **c** are of type **INTEGER**.

2.28 INCLUDE Statement

The INCLUDE statement is a MIL-STD-1753 extension to Fortran 77. It causes the compiler to include and process subsequent source statements from a specified file or device file. When EOF is read from this file or device file, the compiler continues processing with the line following the **INCLUDE** statement. **INCLUDE** statements cannot be continued. **INCLUDE** can be nested in a non-recursive manner, that is, an **INCLUDE** cannot mention an active include file.

SYNTAX

INCLUDE *'name'*

where

name is the file name, enclosed in single quotation marks.

INCLUDE is also a compiler directive. The search path for the **INCLUDE** directive starts from the directory where the caller is located and moves to the directory */usr/include*.

EXAMPLE

```
INCLUDE 'specs'
```

Line numbering within the listing of an included file begins with 1. When the included file listing ends, the include level decreases appropriately, and the previous line numbering resumes. The **INCLUDE** statement also allows the definition of a path. When the following format of the **INCLUDE** is found

```
INCLUDE  'any...thing:more'
```

everything up to the rightmost colon is stripped off and used as an environment variable name. The value of that variable name then is substituted in before the file is searched for. This form of the **INCLUDE** statement can be used to structure the layout of your source files. For example, if the following is specified:

```
INCLUDE 'stuff:file.f'
```

and the environment variable **stuff** is set to */my/own*, the file searched for is */my/own/file.f*. This form of the statement allows each include file to have a potentially different path.

2.29 INQUIRE Statement

The **INQUIRE** statement ascertains properties of a particular file or unit.

SYNTAX

INQUIRE ([[**UNIT=**] *unit*] [,**FILE=***name*] [,**IOSTAT=***ios*] [,**ERR=***label*] . . . *others*)

where

unit is the unit number of a file.

name is the name of the file (as a character expression).

ios is an integer variable or array element for error code. Refer to *Appendix A* for **IOSTAT** error codes.

label is the statement label of an executable statement in the same program unit as the **INQUIRE** statement. If an error occurs during the execution of the **INQUIRE** statement, control is transferred to the specified statement rather than aborting the program.

others are other keyword specifiers which can follow. See the listing of *Inquire Statement Specifiers* in the following sub-section.

Either the **UNIT** or **FILE** keyword specifier must be present in the keyword list, but not both. If the prefix **UNIT=** is omitted, and the unit specifier is present, *unit* must be the first item in the list.

Most of the information returned by an **INQUIRE** statement is assigned through the use of the **OPEN** statement, which is described in *OPEN Statement* later in this chapter. Refer to Chapter 3, *Fortran I/O Statements* for examples using the **INQUIRE** statement.

In Stardent 1500/3000 Fortran, when a result is specified as undefined, a variable may or may not be changed from its previous value by the **INQUIRE** statement. In any event, the value is meaningless if the file or unit either does not exist or cannot be accessed at the time the **INQUIRE** is executed.

2.29.1 INQUIRE Statement Specifiers

ACCESS=*acc* (Character variable, array element, or substring) *acc* returns the string 'SEQUENTIAL', 'ASYNC', or 'DIRECT' depending upon whether specified unit or file was connected for sequential, asynchronous, or direct access, respectively. If the file is not connected, *acc* is undefined.

Example: **ACCESS=acc**.

BLANK=*blk* (Character variable, array element, or substring) *blk* returns 'NULL' if null blank control is is in effect, and 'ZERO' is zero blank control is active. If the connection does not exist or is not connected for formatted I/O, the value assigned is 'UNKNOWN'.

Example: **BLANK=blk**.

CARRIAGECONTROL=*cc*
(Character variable, array element, or substring). *cc* returns 'Fortran' if the file uses Fortran carriage control attributes, 'LIST' if it uses the implied carriage control attribute, 'NONE' if there are no attributes, and 'UNKNOWN' if nothing else applies.

Example: **CARRIAGECONTROL=cc**.

DIRECT=*dir* (Character variable, array element, or substring) *dir* returns 'YES' if the file or unit is connected for direct access, 'NO' if not connected for direct access, and is undefined if the processor is not able to determine the access type.

Example: **DIRECT=dir**.

ERR=*label* (Statement number) Control is transferred to specified executable statement if an error condition on the named file or unit exists. This occurs during execution of the **INQUIRE** statement.

Example: **ERR=99**.

EXIST=*ex* (Logical variable or array element; must be **LOGICAL**) If the named file exists, *ex* = **.TRUE.**; otherwise, *ex* = **.FALSE.**

Example: **EXIST=lext**.

FILE=*name* (Character expression) Specifies the file name for inquiry by file name.

Example: **FILE='OUTPUT'**.

FORM=*fm* (Character variable, array element, or substring) *fm* returns 'FORMATTED' if the file or unit is connected for formatted data transfer, 'UNFORMATTED' if connected for unformatted data transfer, and is not defined if the file is not connected.

Example: **FORM=for**.

FORMATTED=*fmt*
(Character variable, array element, or substring) *fmt* returns the string 'YES' if the file or unit is connected for formatted data transfer, 'NO' if connected for formatted data transfer, and is undefined if the processor is unable to determine the form of data transfer.

Example: **FORMATTED=fm**.

IOSTAT=*ios* (Integer variable or array element) *ios* is assigned the value zero if no error occurs; *ios* is assigned a positive value if an error occurs. Be careful of EOF, which can generate a negative value.

Example: **IOSTAT=io_status**.

NAME=*fn* (Character variable, array element, or substring). *fn* is assigned the external name of the specified unit; if the file has no name or is not connected, *fn* is undefined.

Example: **NAME=nam**.

NAMED=*nmd* (Logical variable or array element; must be **LOGICAL**) If the specified unit is not a scratch file, *nmd* = **.TRUE.**; otherwise, *nmd* = **.FALSE.**

Example: **NAMED=lnam**.

NEXTREC=*nr* (Integer variable or array element) *nr* is assigned the next record number to be read or written on the specified unit or file. If no records have been read or written, *nr* is one. If the file is not connected or is a device file, or its status is indeterminate, *nr* is undefined.

Example: **NEXTREC=nrec.**

NUMBER=*num*
(Integer variable or array element) *num* is the Fortran logical unit number of the external named file; if no unit is connected to the named file, *num* is undefined.

Example: **NUMBER=n.**

OPENED=*od* (Logical variable or array element; must be **LOGICAL**) If the named file or unit has been opened then *od* = **.TRUE.** otherwise, *od* = **.FALSE.**

Example: **OPENED=lopn.**

RECL=*rcl* (Integer variable or array element) *rcl* returns the record length of the specified unit or file connected for direct access, measured in characters for formatted files and in 4-byte words for unformatted files. If the file is not connected for direct access, *rcl* is undefined.

Example: **RECL=irec.**

SEQUENTIAL=*seq*
(Character variable, array element, or substring) *seq* returns 'YES' if the file or unit is connected for sequential access, 'NO' if not connected for sequential access, and undefined if the processor is unable to determine the access type.

Example: **SEQUENTIAL=seq.**

UNFORMATTED=*unf*

(Character variable, array element, or substring) *unf* returns 'YES' if the file or unit is connected for unformatted data transfer, 'NO' if connected for formatted data transfer, and undefined if the processor is unable to determine the form of data transfer.

Example: **UNFORMATTED=iunf**.

UNIT=*unit* (Integer expression) Specifies unit number for inquiry by unit.

Example: **UNIT=7**.

2.30 INTRINSIC Statement

The INTRINSIC statement identifies a name as representing an intrinsic function and permits the name to be used as an actual argument.

SYNTAX

INTRINSIC *fun1* [,*fun2*,. . .]

where

fun1 is the name of an intrinsic function. Each name can appear only once in a given **INTRINSIC** statement, and in at most one **INTRINSIC** statement within a program unit.

The **INTRINSIC** statement provides a means of using intrinsics as actual arguments. The **INTRINSIC** statement is necessary to inform the compiler that these names are intrinsic names and not variable names. Whenever an intrinsic name is passed as an actual argument, it must be placed in an **INTRINSIC** statement in the calling program.

The names of intrinsic functions for type conversion (**INT, IFIX, IDINT, IMAG, AIMAG, FLOAT, SNGL, REAL, DBLE, CMPLX, ICHAR,** and **CHAR**), logical relationships (**LGE, LGT, LLE,** and **LLT**), for choosing the largest or smallest value (**MAX0, AMAX1, AMAX0, MAX1, MIN0, AMIN1, AMIN0, MIN1, MAX,** and **MIN**) and the logarithm intrinsics **LOG** and **LOG10** must **not** be used as actual arguments. Further, only specific function names can be used as actual arguments; generic names must not be used.

EXAMPLE

```
INTRINSIC SIN,TAN
CALL MATH(SIN,TAN)
```

The **INTRINSIC** statement informs the compiler that **SIN** and **TAN** are intrinsics.

2.31 NAMELIST Statement

NAMELIST directed I/O transfers data without specifying the exact format or order of the data. Data is labeled with the variable names. Variables eligible for NAMELIST-directed I/O are defined in NAMELIST groups which are declared with a NAMELIST statement. NAMELIST-directed I/O statements use the group name to determine which variables are to be read or written.

SYNTAX

NAMELIST /<*group name*>/<*var list*> [[,] /<*group name*>/ <*var list*>] ...

where

group name is a symbolic name.

var list is a list of simple or array variable names, separated by commas.

Variables appearing in <*var list*> may be of any type, but dummy arguments are not allowed. A variable name may appear in several NAMELISTs.

The <*var list*> explicitly defines which items may be read or written in a NAMELIST-directed I/O statement. It is not necessary for every item in <*var list*> to be defined during NAMELIST-directed input, but every input item must belong to the NAMELIST group. The order of items in <*var list*> determines the order of the values written in NAMELIST-directed output.

2.32 OPEN Statement

The OPEN statement establishes a connection between a unit number and a file; it also establishes or verifies properties of a file.

SYNTAX

OPEN ([**UNIT=**] *unit* [,**FILE=***name*] [,**IOSTAT=***ios*] [,**ERR=***label*] . . . *others*)

where

unit is the unit number for the file.

name is the file name of the file to be connected to the specified unit; a character expression. A file name is any character string not including a slash, and not longer than 14 characters.

ios is an integer variable or array element for error code return (refer to *Appendix A* for **IOSTAT** error codes).

label is the statement label of an executable statement in the same program unit as the **OPEN** statement. If an error occurs during the execution of the **OPEN** statement, control is transferred to the specified statement.

others are other keyword specifiers which can follow. See the listing of *Inquire Statement Specifiers* in the following sub-section.

The **UNIT** specifier is required in the keyword list. If the prefix **UNIT=** is omitted, *unit* must be the first item in the list. At most one each of the other items can appear in the keyword list.

If a unit is already connected to a file that exists, execution of another **OPEN** statement for that unit is permitted. If the **FILE=***name* option is absent or the file name is the same, the current file remains connected. Otherwise, an automatic close is performed before the new file is connected to the unit. A redundant **OPEN** call can be used to change only the value of the **BLANK=** option. However, attempts to change the values of any other specifiers are ignored. A redundant **OPEN** does not affect the current position of the file.

Refer to Chapter 3, *Fortran I/O Statements* for additional details and examples.

2.32.1
OPEN Statement
Specifiers

FILE=*name*
(Character expression) *name* specifies a file name.

UNIT=*unit*
(Integer expression) *unit* specifies a unit number.

IOSTAT=*ios*
(Integer variable or array element) If no error has occurred, then *ios* = 0; otherwise, if an error condition exists, *ios* is not zero and indicates the error condition.

ERR=*label* (Statement number) Control is transferred to the specified *label* statement if an error is encountered during execution of the **OPEN** statement.

STATUS=*sta*
(Character expression) *sta* specifies a file as **'OLD'**, **'NEW'**, **'SCRATCH'**, or **'UNKNOWN'** (if not specified, **UNKNOWN** is the default).

'OLD' If a file is specified **'OLD'**, the file name must already exist.

'NEW' If a file is specified **'NEW'**, a new file is created.

'SCRATCH'
If a file is specified **'SCRATCH'**, a new file is created and it is deleted when the file is closed.

'UNKNOWN'
If a file is specified **'UNKNOWN'**, the processor first tries **'OLD'**; If the file is not found, the processor uses **'NEW'**, thus, creating a new file.

ACCESS=*acc*
(Character expression) *acc* specifies a file accessed to be **'DIRECT'**, **'ASYNC'** or **'SEQUENTIAL'** (if not specified, **SEQUENTIAL** is the default).

'SEQUENTIAL'
a file is opened for sequential access. **RECL** must not be present.

'DIRECT' a file is opened for direct access. **RECL** is required.

'ASYNC' a file is opened for asynchronous access. **UNIT** is required.

BLANK=*blnk*

(Character expression) *blnk* specifies treatment of blanks within numbers on input. If **BLANK='NULL'** (default), blanks are ignored; if **BLANK='ZERO'**, blanks are treated as zeros.

CARRIAGECONTROL=*cc*

(Character expression) *cc* specifies a value equal to **'FORTRAN'**, **'LIST'**, or **'NONE'**.

The **CARRIAGECONTROL** parameter specifies the type of carriage control processing to be used when printing a file. The default for formatted files is **'FORTRAN'**; for unformatted files, the default is **'NONE'**. **'FORTRAN'** specifies normal Fortran interpretation of the first character, **'LIST'** specifies single spacing between records, and **'NONE'** specifies no implied carriage control.

DISPOSE=*dis* or **DISP=***dis*

(Character expression) *dis* has a value equal to **'DELETE'**, **'KEEP'**, **'PRINT'**, **'PRINT/DELETE'**, **'SAVE'**, **'SUBMIT'**, or **'SUBMIT/DELETE'**.

The **DISPOSE** parameter specifies the disposition of the file connected to the unit when the unit is closed. For example, if **'KEEP'** or **'SAVE'** is specified, the file is retained after the unit is closed; this is the default value. If **'DELETE'** is specified, the file is deleted. If **'PRINT'** is specified, the file is submitted to the system line printer spooler and is not deleted; it is printed and then deleted if you specify **'PRINT/DELETE'**. If **'SUBMIT'** is specified, the file is submitted to the batch job queue and is not deleted; it is submitted and then deleted if you specify **'SUBMIT/DELETE'**.

A read-only file cannot be deleted. A scratch file cannot be saved, printed, or submitted.

FORM=*fm* (Character expression) *fm* specifies data format to be **'FORMATTED'** or **'UNFORMATTED'**. If not specified, **FORMATTED** is the default.

ORGANIZATION=*org*

(Character expression) *org* specifies data format to be **'INDEX'**, **'RELATIVE'**, or **'SEQUENTIAL'**.

The **ORGANIZATION** parameter specifies the internal organization of the file. The default file organization is sequential. However, if you omit the **ORGANIZATION** keyword when you open an existing file, the organization already specified in that file is used. If you specify **ORGANIZATION** for an existing file, *org* must have the same value as that of the existing file.

RECL=*rcl* (Integer expression) *rcl* specifies record length for direct access. Length is measured in characters for formatted files and in 4-byte words for unformatted files. However, if environment variables are set at run-time to indicate a BSD UNIX style of unformatted IO, the field length is interpreted as measured in bytes rather than words. Refer to Chapter 4, *File Handling* in the section titled *Environment Setting Variables* for additional information.

RECORDTYPE=*type*

(Character expression) *type* has a character value equal to **'FIXED'**, **'VARIABLE'**, **'SEGMENTED'**, **'STREAM'**, **'STREAM_CR'**, or **'STREAM_LF'**.

The **RECORDTYPE** parameter indicates that the file has fixed-length records, variable-length records, segmented records, or stream-type variable-length records. The default record type depends on the file type. For example, if the file type is relative or indexed, the default is fixed; if the file type is direct access sequential, the default is also fixed; if the file type is formatted sequential access, the default is variable; and if the file type is unformatted sequential access, the default is segmented.

A segment record consists of one of more variable-length records. Use of segmented records allows a Fortran logical record to span several records. Only unformatted sequential access files with sequential organization can use segmented records. **'SEGMENTED'** cannot be specified for any other type.

If the **RECORDTYPE** parameter is not specified when accessing an existing file, the record type of the file is used. However, there is an exception when unformatted sequential files are accessed with sequential organization and variable-length records, these files have a default of **'SEGMENTED'**.

If the **RECORDTYPE** parameter is specified when accessing an existing file, the specified type must match the type of an existing file.

In fixed-length record files, if an output statement does not specify a full record, the record is filled with spaces (for a formatted file) or zeros (for an unformatted file).

You cannot use an unformatted **READ** statement to access an unformatted sequential organization file containing variable-length records, unless you specify the corresponding **RECORDTYPE** value in your **OPEN** statement.

Files containing segmented records can be accessed only by unformatted sequential Fortran I/O statements.

EXAMPLES

```
      OPEN(10, FILE='inv', ACCESS='SEQUENTIAL', ERR=100,
     +     IOSTAT=ios)
```

The file **inv** is connected to unit **10** as a sequential file. If an error occurs, control is transferred to statement **100** and the error code is placed in the variable **ios**.

```
      OPEN(ACCESS='DIRECT', UNIT=4, RECL=50, FORM='FORMATTED',
     +     FILE=NEXT1)
```

The character variable **NEXT1** contains the name of the file to be connected to unit 4 as a formatted, direct access file with a record length of 50 characters.

```
      OPEN(6, FILE='/dev/console')
```

The system device **/dev/console** is connected to Fortran unit **6**.

NOTE
The same file cannot be connected to two different units. An attempt to open a file that is connected to a different unit may cause undefined results.

When a file is opened with the unit parameter specified, but no file parameter, a scratch file is opened. Thus, the following two statements are equivalent:

```
      OPEN (UNIT=19)

      OPEN (UNIT=19, STATUS='SCRATCH')
```

2.33 OPTIONS Statement

Options specified in the **OPTIONS** statement override any of the same options that you might have specified on the command line. However, the options so specified are active only for the program module in which they are located. The original command line options again take effect for any other program module in the compilation you have requested.

SYNTAX

OPTIONS *qualifier* [,*qualifier*...]

where

qualifier
is one of the following:

/I4 or **/NOI4**
/F77 or **/NOF77**

For detailed information on the qualifiers and their meanings, refer to Chapter 7, *Compiler Options* in the section titled *Miscellaneous Options*.

2.34 PARAMETER Statement

The **PARAMETER** statement defines named constants. After the definition of a name in a **PARAMETER** statement, subsequent uses of the name are treated as if the constant were used.

SYNTAX

PARAMETER (*cname1=cexp* [*,cname2=cexp*] . . . .)

where

cname is a symbolic name that represents a constant. This name cannot appear in any statement before **PARAMETER** except a type statement.

cexp is a constant expression.

If the symbolic name *cname* is of type integer, short integer, real, double precision, complex, or double complex, the corresponding expression *cexp* must be an arithmetic constant expression. If the symbolic name *cname* is of type character or logical, the corresponding expression *cexp* must be a character constant expression or a logical constant expression, respectively. As an extension to Fortran 77, character constant expressions can include **CHAR**(*value*), in which case *value* must be an integer constant expression, or **ICHAR**(*cvalue*), in which case *cvalue* must be a character constant expression. As an extension to Fortran 77, the following intrinsic functions can be used in the **PARAMETER** statement:

```
ABS    CHAR   CMPLX  CONJG  DIM    DPROD  IAND
ICHAR  IEOR   IMAG   IOR    ISHFT  LGE    LGT
LLE    LLT    MAX    MIN    MOD    NINT   NOT
```

as can the logical operators (that is, **.EQ.**, **.NE.**, **.LT.**, **.LE.**, **.GT.** and **.GE.**).

Any symbolic name of a constant that appears in an expression *cexp* must have been defined previously in the same or a different **PARAMETER** statement in the same program unit. A symbolic name of a constant must not become defined more than once in a program unit.

If a symbolic name of a constant is not of default implied type, its type must be specified by a type statement or **IMPLICIT** statement prior to its first appearance in a **PARAMETER** statement. If the length specified for the symbolic name of a constant of type

character is not the default length of 1, its length must be specified in a type statement or **IMPLICIT** statement prior to the first appearance of the symbolic name of the constant. Its type and length must not be changed by subsequent statements, including **IMPLICIT** statements. If a symbolic name of type **CHARACTER*(*)** is defined in a **PARAMETER** statement, its length becomes the length of the expression assigned to it.

Once such a symbolic name is defined, that name may appear in any subsequent statement of the defining program unit as a constant in an expression or **DATA** statement. A symbolic name of a constant can not be used as a constant in a format specification.

A symbolic name in a **PARAMETER** statement does not identify the corresponding constant outside its program unit.

EXAMPLES

```
      PARAMETER (minval=-10, maxval=50)

      PARAMETER (debug=.TRUE.)

      CHARACTER*(*) file
      PARAMETER (file='WELCOM')

      PARAMETER (lower=0, upper=7)
      DIMENSION a (lower:upper)
      DO 10 i=lower,upper
        a(i) = 1.0
   10 CONTINUE

      PARAMETER (pi=3.14159)
C       (more statements)
        area = pi *(radius**2)

      CHARACTER bell
      PARAMETER (bell = CHAR(7))

      INTEGER case_shift
      PARAMETER (case_shift=ICHAR('a')-ICHAR('A'))

      COMPLEX complex_two
      PARAMETER (complex_two = 2)

      PARAMETER (limit = 1000)
      PARAMETER (limit_plus_1 = limit+1)
```

2.35 PAUSE Statement

The **PAUSE** statement causes a temporary break in program execution.

SYNTAX

```
PAUSE [n]
```

where

n is an integer or a character constant.

The **PAUSE** statement suspends the execution of the program. When the program suspends, the message printed at the terminal depends on whether digits, characters, or nothing has been specified in the **PAUSE** statement. If digits, the message "**PAUSE** *digits*" is written to standard error. If a character expression, the message "**PAUSE** *character expression*" is written to standard error.

After displaying the appropriate message, the **PAUSE** statement writes one of two messages that give information on resuming the program. If the standard input device is a terminal, the message is:

```
To resume program execution, type go.
Any other input will terminate job.
```

At this point the program is suspended and remains so until the operator types **go** plus a carriage return. The program will terminate if anything other than **go** is entered. If the standard input device is other than a terminal, the message is:

```
To resume execution, execute a kill -15 pid command
```

where *pid* is the unique process identification number of the suspended program. This command can be issued at any terminal.

EXAMPLES

```
PAUSE 7777
```

The message "**PAUSE 7777**" is written to standard error.

```
PAUSE 'Mount tape'
```

The message "**PAUSE Mount tape**" is written to standard error.

```
PAUSE
```

The message "**PAUSE**" is written to standard error.

After execution resumes, either by the operator typing **go** or executing the **kill** command, this message is written to standard error:

```
Execution resumes after PAUSE.
```

2.36 PRINT Statement

The PRINT statement transfers data from memory to the standard output unit. (Fortran unit 6 is preconnected to the UNIX "standard output.")

SYNTAX

```
PRINT fmt[ ,list]
PRINT nml
```

where

fmt is the format designator. *fmt* must be one of the following:

- The statement label of a **FORMAT** statement.
- A variable that has been assigned the statement label of a **FORMAT** statement (this form requires **INTEGER*4** type).
- A character or non-character array name that contains the representation of a format specification.
- An asterisk, which specifies list-directed output (refer to *List-Directed Output* in Chapter 3, *Fortran I/O Statements*).

list is a list that specifies the data to be transferred. Each item in *list* must be one of the following:

- A constant.
- A variable name.
- An array element name.
- A substring.
- An array name.
- An expression.
- An implied **DO** loop.

nml is a namelist group previously defined in a **NAMELIST** statement. Refer to *NAMELIST-Directed Output* in Chapter 3, *Fortran I/O Statements*.

EXAMPLES

```
PRINT 10,num,des
```

Prints **num** and **des** according to **FORMAT** statement **10**.

```
PRINT *,'x=',x
```

Prints the literal string **x=** and the value of the variable **x** according to list-directed formatting.

```
ASSIGN 100 TO fmt
PRINT fmt, rat, cat
```

Prints **rat** and **cat** according to **FORMAT** statement **100**.

```
PRINT '(4I3)', i, j, k*2, 330
```

Prints **i, j, k*2** and the constant **330** according to the format specification in the **PRINT** statement itself.

```
      PRINT 100
  100 FORMAT ('End of report')
```

Prints the character constant in the **FORMAT** statement.

```
      PRINT '( 'x   SIN(x)   COS(x)' (I3,2F7.3))',
     1 (i, SIN(i/57.3), COS(i/57.3), i=0, 360,5)
```

Prints a literal heading and 73 rows of values as indicated by the implied **DO** in the input/output list.

```
PRINT namelist1
```

Prints variables in **NAMELIST** group **namelist1** according to **NAMELIST**-directed formatting.

The **PRINT** statement is used only for transferring data from memory to the standard output unit. Refer to Chapter 3, *Fortran I/O Statements* for a more detailed discussion of the **PRINT** statement.

2.37 PROGRAM Statement

The PROGRAM statement defines the name of the program.

SYNTAX

PROGRAM *name*

where

name is the name of the program (and its entry point).

The *name* in a **PROGRAM** statement is used for documentation purposes only. The program statement must be the first non-comment statement in a module except for certain compiler directives.

EXAMPLE

```
PROGRAM main
```

This example specifies **main** as the name of the program.

2.38 READ Statement

The **READ** statement transfers data from a file to program variables.

2.38.1 Read from the Standard Input Unit Statement

The **READ** from the standard input unit statement transfers data from a unit designated to be the standard input unit to memory. Fortran unit 5 is pre-connected to the UNIX "standard input," usually your terminal.

SYNTAX

READ *fmt* [,*list*]
READ *nml*

where

fmt is the format designator. *fmt* must be as specified in the **PRINT** statement discussed earlier in this chapter.

list is a list of variables that specify where the data is to be transferred. The items in list must be one of the following:

- A variable name.
- An array element name.
- An array name.
- A substring.
- An implied **DO** loop containing the above items only.

nml is a namelist group name previously defined in a **NAMELIST** statement. Refer to *NAMELIST-Directed Input* in Chapter 3, *Fortran I/O Statements*.

EXAMPLES

```
READ 10,num,des
```

The values of **num** and **des** are read according to **FORMAT** statement **10**.

```
READ *,a,b,n
```

The values of **a, b** and **n** are read according to list-directed formatting.

```
READ 5
```

Skips a record on the standard input device.

```
READ namelist1
```

Values of variables in **NAMELIST** group **namelist1** are read according to **NAMELIST**-directed formatting.

2.38.2 Read from File Statement

The **READ** from file statement transfers data from a file to memory. Refer to Chapter 4, *File Handling* for details on file handling.

SYNTAX

READ ([**UNIT=**] *unit* [,[**FMT=**] *fmt*] [,[**NML=**] *nml*] [,**IOSTAT=***ios*]
[,**ERR=***errlabel*] [,**END=***endlabel*] [,**REC=***rn*]) [*list*]

where

unit is the unit number for the input device or file and is a required parameter. If the prefix **UNIT=** is omitted, then *unit* must be the first item in the list. Unit can be an arithmetic expression of type integer, an asterisk, or a character variable, array name, array element, or substring. An asterisk indicates that the standard input device (unit 5).

fmt is the format designator. *fmt* must be as specified in *PRINT Statement* earlier in this chapter.

If the prefix **FMT=** is omitted, then *fmt* must be the second item in the list and *unit* (without a prefix) must be the first item. If neither *fmt* nor *nml* are present, the access is unformatted.

nml is a namelist group name previously defined in a **NAMELIST** statement. *fmt* and *nml* may not appear in the same **READ** statement. If the prefix **NML=** is omitted, then *nml* must be the second item in the list and *unit* (without a prefix) must be the first item. No *list* is permitted in a **NAMELIST**-directed **READ** statement.

ios is an integer variable or integer array element name for error return (refer to *Appendix A* for **IOSTAT** error codes).

errlabel is the statement label of an executable statement. If an error occurs during the execution of the **READ** statement, control is transferred to the specified statement.

endlabel is the statement label of an executable statement. If an end-of-file is encountered in a sequential file during the execution of the **READ** statement, control is transferred to the specified statement. In this case, *ios* has a negative value.

rn specifies the record number in a direct access file. A record specifier must not be present if either: (a) *fmt* is an asterisk, or (b) *nml* is present.

list is a list that specifies the data to be transferred. The items in *list* must be one of the following:

- A variable name.
- An array element name.
- An array name.
- A substring.
- An implied **DO** loop containing the above items only.

A **READ** from file statement must contain a unit number and at most one of each of the other options. Note that **REC=***rn* cannot appear with **END=***endlabel* or **UNIT=***. If **REC=***rn* appears, the unit must be connected for direct access.

EXAMPLES

```
READ (8, 10) a, b, c
```

The values of **a, b** and **c** are read from the file connected to unit **8** according to **FORMAT** statement **10.**

```
ASSIGN 4 TO num
READ (UNIT=3, ERR=50, FMT=num) z
```

The value of **z** is read from the file connected to unit **3** according to **FORMAT** statement **4.** If an error occurs, control is transferred to statement **50.**

```
READ(10) x
```

The value of **x** is read from the file connected to unit **10.** Because *fmt* is omitted, the data is unformatted.

```
READ (10, FMT=*, END=60) b
```

The value of **b** is read from the file connected to unit **10** according to list-directed formatting. If an **EOF** is encountered, control is passed to statement **60.**

```
READ (2, '(13)', REC=10) i
```

The value of **i** is read from the tenth record of the direct file connected to unit **2** according to the format specification in the **READ** statement itself.

```
READ (10)
```

This example skips a record in the file connected to unit 10.

```
CHARACTER*8 a
REAL b
a = '  $27.97'
READ (a(4:8), '(F5.2)') b
```

The value of **b** is read from the character variable **a** according to the format specification in the **READ** statement itself.

```
READ(8, NML=namelist1)
```

Values of variables in **NAMELIST** group **namelist1** are read from unit 8 according to NAMELIST-directed formatting.

2.39 RETURN Statement

The **RETURN** statement transfers control from a subprogram back to the calling program unit.

SYNTAX

RETURN [*rtnnum*]

where

rtnnum is an integer expression specifying the alternate return number.

Normally control is returned from a subroutine to the calling program unit at the statement following the **CALL** statement. The specification of alternate return statements allows return to the calling program unit at any labeled executable statement in the calling program unit.

When the **RETURN** statement occurs in a subroutine, and no alternate return is specified, control returns to the first executable statement following the **CALL** statement that referenced the subroutine. When the **RETURN** statement occurs in a **FUNCTION** (alternate returns not allowed), control returns to the statement containing the function call.

The optional integer expression takes the values *rtnnum*, where *n* is the number of alternate returns specified by the number of asterisks in the **CALL** statement. The value of *rtnnum* identifies the number of the statement label in the actual argument list of the **CALL** statement. The asterisks in the **SUBROUTINE** statement are for documentation purposes. There should be the same number of asterisks as there are statement labels in the **CALL** statement.

If the value of *exp* is in the range 1 to N, the return is to the *exp*-th label in the argument list. If *exp* is outside this range, the return is to the statement following the call.

EXAMPLES

```
      PROGRAM main
C      (more statements)
      CALL matrx(*1, m, *2, n, k, *3)
C      (more)
    1 ...   !first alternate return
C      (more)
    2 ...   !second alternate return
C      (more)
    3 ...   !third alternate return
C      (more)
      END
```

The **CALL** statement specifies three possible return labels, plus the normal return point (statement following the **CALL**). The **SUBROUTINE** statement contains a number of asterisks equal to the number of statement labels in the **CALL** statement.

```
      SUBROUTINE matrx(*, m, *n, k, *)
C      (more)
      k = 2
      RETURN k
      END
```

k evaluates to the value 2, causing control to pass to the second alternate return label specified in the **CALL** statement (2). If **k** evaluates to a value outside the range $1 \leq k \leq 3$, control returns to the statement following the **CALL** statement.

2.40 REWIND Statement

The **REWIND** statement positions a sequential file or device at the beginning-of-file.

SYNTAX

```
   REWIND unit
or
   REWIND ([UNIT=]unit [ ,IOSTAT=ios [ , ]ERR=label])
```

where

unit is the unit number of a sequential file or device.

ios is an integer variable or array element for error code return (refer to *Appendix A* for **IOSTAT** error codes).

label is the statement label of an executable statement. If an error occurs during the execution of the **REWIND** statement, control is transferred to the specified statement.

If the file is already positioned at beginning-of-information, a **REWIND** statement has no effect upon the file. The **REWIND** statement cannot be used on direct access files.

EXAMPLES

```
REWIND 10
```

The file connected to unit **10** is positioned at the beginning- of-information.

```
REWIND (UNIT=5, IOSTAT=j, ERR=100)
```

The file connected to unit **5** is positioned at the beginning-of-information. If an error occurs, control is transferred to statement **100** and the error code is returned in **j**.

2.41 SAVE Statement

The variables in a subroutine that are mentioned in a **SAVE** statement maintain their values after the execution of a **RETURN** or **END** statement.

SYNTAX

SAVE [*var1*[,*var2*[,*var3*,...]]]

where

var is a simple variable name, an array variable name, or a labeled common block name. The common block name must be preceded and followed by a slash.

A **SAVE** statement causes all variables named to have their values retained from one call of the subroutine to the next. Variables in **COMMON** are saved even if all subroutines mentioning the **COMMON** block are exited.

The following items must not be mentioned in a **SAVE** statement: dummy argument names, procedure names, and names of variables in a common block. A **SAVE** statement without a list of variable names or common block names declares that all allowable variables in the subprogram must be saved. When a common block name is specified, all of the variables in that common block are saved. Within an executable program, if a common block name is mentioned in a **SAVE** statement, it must be mentioned in a **SAVE** statement in each subprogram where it appears. A **SAVE** statement is optional in a main program.

All variables are always saved in Stardent 1500/3000 Fortran programs.

EXAMPLES

```
      SUBROUTINE matrix
C      (more statements)
      SAVE a,b,c,/dot/
C      (more)
      RETURN
```

The **SAVE** statement saves the values of **a**, **b**, and **c**, and the values of all of the variables in the common block **dot**.

```
      SUBROUTINE fixit
      SAVE
C      (more statements)
      RETURN
```

The **SAVE** statement saves the values of all of the variables in the subroutine **fixit**.

2.42 Statement Function Statement

The statement function statement defines a one-statement local function.

SYNTAX

name ([*parm1*, *parm2*,...])=*exp*

where

name is the user-specified name of the function.

parm is a dummy argument, which must be a simple variable.

exp is an arithmetic, logical, or character expression.

A statement function is a user-defined, single-statement computation that applies only to the program unit where it is defined. A statement function statement can appear only after the specification statements and before the first executable statement of the program unit. The expression defines the actual computational procedure which derives one value. When the statement function is referenced, this value is assigned to the function name. The expression must be an arithmetic, logical, or character expression.

The type of a statement function is determined by using the statement function name in a type statement or by implicit typing. The type of expression in a statement function statement must be compatible with the defined type of the name of the function. For example, arithmetic expressions must be used in arithmetic statement functions, logical expressions in logical statement functions, and character expressions in character statement functions.

The arithmetic expression in an arithmetic statement function need not be the same type as the function name. (For example, the expression can be type integer and the function name can be defined as type real.) The expression value is converted to the statement function type at the time it is assigned to the function name.

Statement functions may reference other previously defined statement functions. Statement functions cannot contain calls to themselves nor can they contain indirect recursive calls.

The values of any dummy arguments in the expression are supplied at the time the statement function is referenced. All other expression elements are local to the program unit containing the reference and derive their values from statements in the containing program unit.

Refer to Chapter 5, *Procedures and Subprograms* for additional information about actual and dummy arguments and calling a statement function.

EXAMPLES

```
disp(a, b, c) = a + b*c
```

A statement function with three arguments is defined by the statement function statement.

```
tim(t1) = t1/2 + b
```

The statement function has one dummy argument. The value of **b** is the current "program" value of **b** when the function is invoked.

2.43 STOP Statement

The STOP statement terminates program execution.

SYNTAX

```
STOP [n]
```

where

n is an integer or character constant. The **STOP** statement terminates program execution before the end of the program unit.

EXAMPLES

```
STOP 7777
```

The message "**STOP 7777**" is written to standard error.

```
STOP 'Program ended!'
```

The message "**STOP Program ended!**" is written to standard error.

```
STOP
```

Nothing is written to standard error.

```
READ *,a,b
IF (a .LT. b) STOP 56789
```

If **a** is less than **b**, execution terminates.

```
10 b = b-1
   IF (b .EQ. a) STOP 'All done'
   GOTO 10
   END
```

When **b** equals **a**, execution terminates.

2.44 SUBROUTINE Statement

The **SUBROUTINE** statement identifies the program unit as a subroutine suprogram.

SYNTAX

SUBROUTINE *name* [([*arg1*,*arg2*,...] [*,...]....)]

where

name is the name of the subroutine.

arg is a dummy argument of the subroutine.

*... indicates one or more alternate returns.

The dummy arguments in a **SUBROUTINE** statement can be variables, array names, or subprogram names. They must be of the same type and structure as the actual arguments that are passed to the subroutine.

One or more alternate returns can be specified by the presence of asterisks (or ampersands) in the **SUBROUTINE** statement. Alternate returns are described in *RETURN Statement* earlier in this chapter. Refer to Chapter 5, *Procedures and Subprograms* for more information on subroutine subprograms.

EXAMPLES

```
SUBROUTINE add
```

Begins a subroutine named **add**.

```
SUBROUTINE sub(z,i,d,*,*,*)
```

Begins a subroutine named **sub** with three arguments, and three alternate return points.

2.45 TYPE Statement

The TYPE statement transfers data from memory to the standard output unit. (Fortran unit 6 is preconnected to the UNIX "standard output.")

SYNTAX

```
TYPE fmt [ ,list]
TYPE nml
```

where

fmt is the format designator. *fmt* must be one of the following:

- The statement label of a **FORMAT** statement.
- A variable that has been assigned the statement label of a **FORMAT** statement.
- A character or non-character array name that contains the representation of a format specification (use of a non-character array is an extension to Fortran 77).
- A character expression.
- An asterisk, which specifies list-directed output (refer to *List-Directed Output* in Chapter 3, *Fortran I/O Statements*).

list is a list that specifies the data to be transferred. Each item in *list* must be one of the following:

- A constant.
- A variable name.
- An array element name.
- A substring.
- An array name.
- An expression.
- An implied **DO** loop.

nml is a namelist group previously defined in a **NAMELIST** statement. Refer to *NAMELIST-Directed Output* in Chapter 3, *Fortran I/O Statements*.

2.46 Type Statement

The type statement is a specification statement that assigns an explicit type to symbolic names which would otherwise have their type implicitly determined by the first letter of their names.

SYNTAX

type name1 [*,name2,...*]

where

type is the type from the following list to be associated with the specified variables:

```
LOGICAL       BYTE
LOGICAL*1     INTEGER       REAL
LOGICAL*2     INTEGER*2     REAL*4
LOGICAL*4     INTEGER*4     REAL*8

COMPLEX*8     COMPLEX*16    DOUBLE COMPLEX

DOUBLE PRECISION            CHARACTER[*len]
```

The *len* parameter specifies the length of character variables not having their own length specifications; default value is 1. *len* can be one of the following:

- An unsigned nonzero integer constant.
- An integer constant expression with a positive value. The integer expression must be enclosed in parentheses, and cannot contain variable names. Example: **(-3 + 4)**.
- An asterisk enclosed in parentheses, **(*)**.

If name has the ******len* length specification, it overrides the length *n* in the **CHARACTER****n* specification. Thus, **CHARACTER*6 q**, **CHARACTER q*6**, and **CHARACTER*10 q*6** are all equivalent. Each reserves space for a character variable of length 6.

name is a simple variable name, an array name, an array declarator, a named constant, or a function name. When type is **CHARACTER**, name may have ******len* as a suffix to designate its length. Each name can appear in a type statement only once.

If an array declarator is specified in a type statement, the declarator for that array must not appear in any other specification

statement (such as **DIMENSION**). If only the array name is specified, then an array declarator must appear within a **DIMENSION** or **COMMON** statement.

The **CHARACTER*(*)** form can be used only for named constants, dummy arguments, function subprograms or **ENTRY** names.

EXAMPLES

```
      INTEGER run, time
```

The variables **run** and **time** are single-word integers.

```
      CHARACTER*5 name(6)*10, zip(6)
```

The variables **name** and **zip** are character arrays with six elements each. Each element in **name** has a length of 10; in **zip**, 5.

```
      INTEGER*2 rn, hours(4,5)
```

The variable **rn** and each element of the two-dimensional array, **hours**, are short integers.

```
      REAL item
      DIMENSION item(2,3,5)
```

item is a three-dimensional real array.

```
      CHARACTER*6 var
      CALL sub (var)
C      (more statements)
      SUBROUTINE sub (var1)
      CHARACTER*(*) var1
```

The dummy argument **var** is defined as being of type character and six characters long. The dummy argument **var1** is defined as being of type character. At execution of the call to *sub*, the dummy argument **var1** will be assigned the length of variable *var* for the duration of the call.

2.47 WRITE Statement

The **WRITE** statement transfers data from memory to a file or device.

SYNTAX

WRITE ([**UNIT=**]*unit* [,[**FMT=**]*fmt*] [,[**NML=**]*nml*] [,**IOSTAT=***ios*]
[,**ERR=***label*] [,**REC=***rn*]) [*list*]

where

unit is the unit number for the file and is required. If the prefix **UNIT=** is omitted, then *unit* must be the first item in the list. *unit* may be an asterisk in which case the write is done to the standard output device. (In Stardent 1500/3000 Fortran, unit 6 is preconnected to UNIX standard output.)

fmt is the format designator. *fmt* must be as specified in *PRINT Statement* earlier in this chapter. If the prefix **FMT=** is omitted, then *fmt* must be the second item in the list and *unit* (without a prefix) must be the first item.

nml is a namelist group name previously defined in a **NAMELIST** statement. *fmt* and *nml* may not appear in the same **WRITE** statement. If the prefix **NML=** is omitted, then *nml* must be the second item in the list and *unit* (without a prefix) must be the first item. No *list* is permitted in a **NAMELIST**-directed **WRITE** statement, nor may *unit* specify internal I/O.

ios is an integer variable or array element for error code return (refer to *Appendix A* for **IOSTAT** error codes).

rn specifies the number of the record in a direct access file. A record specifier must not be present if either: (a) *fmt* is an asterisk, or (b) *nml* is present.

label is the statement label of an executable statement. If an error occurs during execution of the **WRITE** statement, control is transferred to the specified statement rather than aborting the program.

list is a list that specifies the data to be transferred. Each item in *list* must be from the following:

- A variable name.
- An array element name.
- An array name.
- A substring.
- An expression.

The list may contain implied **DO** loops. For syntax and detailed information on implied **DO** loops, refer to *Implied DO Loops* under *DO Statement* earlier in this chapter.

A **WRITE** statement must contain a unit number and at most one of each of the other options. **REC=***rn* must appear for direct access. The **END=** specifier cannot appear in a **WRITE** statement.

EXAMPLES

```
WRITE (7, 10) a, b, c
```

The values of **a, b** and **c** are written to the file connected to unit **7** according to **FORMAT** statement **10.**

```
ASSIGN 4 TO num
WRITE (UNIT=3, IOSTAT=j, ERR=5, FMT=num) z
```

The value of **z** is written to the file connected to unit **3** according to **FORMAT** statement **4.** If an error occurs, control is transferred to statement **5** and the error code is returned in **j.**

```
WRITE (10) (x + y)
```

The value of the expression **(x + y)** is written to the file connected to unit **10.** Because *fmt* is omitted, the data is unformatted.

```
WRITE (10, FMT=*) b
```

The value of **b** is written to the file connected to unit **10** according to list-directed formatting.

```
WRITE (2, '(13)', REC=10) i
```

The value of **i** is written to the tenth record of the direct file connected to unit **2** according to the statement itself.

```
WRITE (*)
```

One record is skipped on the standard output device.

```
WRITE(7, NML=namelist1)
```

Values of variables in **NAMELIST** group **namelist1** are written to unit **7** according to **NAMELIST**-directed formatting.

Refer to Chapter 3, *Fortran I/O Statements*, and Chapter 4, *File Handling* for more information on the **WRITE** statement.

FORTRAN I/O STATEMENTS

CHAPTER THREE

Input/output (I/O) statements allow you to enter data into a program and to transfer data between a program and a disk file, terminal or other device. There are four types of input/output:

- Formatted input/output.
- Unformatted input/output.
- List-directed input/output.
- Namelist input/output.

For each type of input/output, there are one or more input statements and corresponding output statements.

3.1 Formatted Input/Output

Formatted input/output allows you to control each character of a data record. This control is specified by a format given in a **FORMAT** statement or in a character expression.

3.1.1 Formatted Input

Formatted input is specified by the input statements **ACCEPT** and **READ**.

SYNTAX

```
ACCEPT fmt, list
```

Refer to Chapter 2, *Fortran Statements* for more specific syntax and information specifying the optional keywords.

The **ACCEPT** statement transfers information from the standard input unit (unit 5). This statement can only be used on implicitly connected logical units; it can never be used with user-specified logical units.

SYNTAX

```
READ [ fmt,list ]
READ ( unit,fmt,...optional keywords ) list
```

where

fmt is the format designator.

unit is the unit number of the file (discussed in Chapter 4, *File Handling*).

list is the list of variables that specifies where the data is to be transferred.

Refer to Chapter 2, *Fortran Statements* for more specific syntax and information specifying the optional keywords.

The first **READ** statement syntax shown above transfers information from the standard input unit (unit 5 in Stardent 1500/3000 Fortran). The second **READ** statement transfers data from a file or device. (Files, along with other **READ** statement options, are discussed in Chapter 4, *File Handling*.)

Reading always starts at the beginning of a record. Reading stops when the list is satisfied, provided that the format specification and the record length are in agreement with the list. If the list is longer than the format specification, reading skips to the next record, which is read using part or all of the format specification again. This process continues until the list is satisfied. After the **READ**, the file pointer is positioned at the beginning of the next record.

NOTE
As a Stardent 1500/3000 Fortran extension, if the format item is an A format, then blanks are supplied to fill out the A format.

Each **READ** statement begins reading values from a fresh record of the file; any values left unread in records accessed by previous **READ** statements are ignored. For example, if the record contained six data elements, a **READ** statement such as

```
READ (4,100) i,j
```

would only read the first two elements. The remaining four elements would not be read because any subsequent **READ** statement would read values from the next record, unless the file pointer was repositioned (with **BACKSPACE** or **REWIND**, for example) before the next **READ**.

Array names in the list represent all the elements in the array. Values are transferred to the array elements in accordance with

the standard array storage order (refer to Chapter 1, *Language Elements*).

3.1.2 Formatted Output

Formatted output is specified by the following output statements:

SYNTAX

```
PRINT fmt,list
TYPE fmt,list
WRITE ( unit,fmt,...optional keywords ) list
```

where

fmt is the format designator.

list is a list of variables or expressions that specifies the data to be transferred.

Refer to Chapter 2, *Fortran Statements* for more specific syntax and information specifying the optional keywords.

The **PRINT** and **TYPE** statements transfer information to the standard output unit (unit 6 in Stardent 1500/3000 Fortran). The **WRITE** statement transfers information to disk files or to output devices. (Files, along with other **WRITE** options, are discussed in Chapter 4, *File Handling*.)

Each **WRITE** statement begins writing values into a fresh record of the destination file; any space left unused in records accessed by previous write statements is ignored. The carriage control characters are listed in Table 3-1. After the transfer is completed, the file record pointer is advanced.

Table 3-1. Carriage Control Characters

Character	Vertical Spacing
Blank	One line (single spacing)
0	Two lines (double spacing)
1	To first line of next page (page eject)
+	No advance
Any other character	Device dependent

Array names in the list represent all the elements of the array. Array element values are transferred in accordance with the standard array storage order (refer to Chapter 1, *Language Elements*).

3.2 Format Specifications

A format specification is a list of format descriptors and edit descriptors. The format descriptors describe how the data appears and edit descriptors specify editing information.

Format specifications can be specified in **FORMAT** statements or as character expressions in input/output statements. An empty format specification of the form **()** can be used if no list items are specified. In this case, one input record is skipped or one output record is written.

3.2.1 Format Specifications in Format Statements

A format specification can be placed in a **FORMAT** statement that is referenced by a corresponding **READ**, **WRITE** or **PRINT** statement.

SYNTAX

label **FORMAT** **(***des1* **[,***des2*,...**])**

where

label is a statement label.

des is a format or edit descriptor, or, within parentheses, a list of descriptors.

A **FORMAT** statement can be referenced by several input/output statements.

EXAMPLE

```
   READ (5,10) a, i, d, e
10 FORMAT(A2, I3, D8.2, F12.2)
```

The format specification (**A2, I3, D8.2, F12.2**) corresponds with the variables **a, i, d** and **e** in the **READ** statement. List element **a** corresponds to the format descriptor **A2, i** to **I3, d** to **D8.2** and **e** corresponds to **F12.2**.

3.2.2 Format Specifications in Input/Output Statements

The format specification can be contained in the input/output statement as a character expression. Care must be taken when the format specification in an input/output statement contains a single apostrophe; two apostrophes must be written to represent one. Whenever an apostrophe appears within a single apostrophe edit descriptor, it must be represented by two consecutive apostrophes. Each of these, in turn, must be represented by two apostrophes if the format is a character literal contained in an input/output statement. Notice, therefore, that four consecutive apostrophes are required in the second example below.

EXAMPLES

```
WRITE (6,'(3X,''THIS IS THE END'')')
```

writes the following record:

```
THIS IS THE END
```

```
WRITE (6,'(''Ain''''t it true!'')')
```

writes the following on the display screen:

```
Ain't it true!
```

The variables **a** and **z** are read according to the format specification **A3, 3X, F10.2**.

```
READ (UNIT = 4,'(A3, 3X, F10.2)') a,z
```

The three integers, **i**, **j**, and **k** are printed according to the format specification **3I3**.

```
CHARACTER*6 a
DATA a /'(3I3)'/
PRINT a, i,j , k
```

The variable **d** is written as a fixed-point number according to the format specification **F10.2**.

```
WRITE (6,'(F10.2)') d
```

3.2.3 Repeat Specification

The repeat specification is a positive integer written to the left of the format descriptor it controls.

The repeat specification allows one format descriptor to be used for several list elements. It can also be used for nested format specifications; thus edit descriptors can be repeated by enclosing them in parentheses as shown above.

EXAMPLES

`(3F10.5)`	is equivalent to **(F10.5,F10.5,F10.5)**
`(2I3,2(3X,A5))`	is equivalent to **(I3,I3,3X,A5,3X,A5)**
`(L2,2(F2.0,2PE4.1),I5)`	is equivalent to **(L2,F2.0,2PE4.1,F2.0,2PE4.1,I5)**
`(2P3G10.4)`	is equivalent to **(2PG10.4,G10.4,G10.4)**

3.2.4 Nesting of Format Specifications

The group of format and edit descriptors in a format specification can include one or more other groups enclosed in parentheses (called group(s)) at nested level *n*. Each group at nested level 1 can include one or more other groups at nested level 2; those at level 2 can include groups at nested level 3, and so forth.

EXAMPLES

`(E9.3,I6,(2X,I4))`	One group at nested level 1.
`(L2,A3/(E10.3,4(A2,L4)))`	One group at nested level 1 and one at nested level 2.
`(A,(3X,(I2,(A3)),I3),A)`	One group at nested level 1, one at level 2, and one at level 3.

A formatted input/output statement references each element of a series of list elements and the corresponding format specification is scanned to find a format descriptor for each list element. If a

program does not provide a one-to-one match between list elements and format descriptors, execution continues only until a format descriptor, an outer right parentheses, or a colon is encountered, and there are no list items left. If there are fewer format descriptors than list elements, the following three steps are performed:

(1) The current record is terminated.

(2) A new record is started.

(3) Format control is returned to the repeat specification for the rightmost specification group at nested level 1. In the event there is no group at level 1, control returns to the first descriptor in the format specification.

EXAMPLES

`(I5,2(3X,I2,(I4)))`	Control returns to **2(3X,I2,(I4))**
`(F4.1,I2)`	Control returns to **(F4.1,I2)**
`(A3,(3X,I2),4X,I4)`	Control returns to **(3X,I2),4X,I4**

3.2.5 Variable Format Expressions

Variable format expressions allow you to use an arithmetic expression, enclosed in angle brackets, wherever an integer can be used in a **FORMAT** statement. Each time the format is scanned the variable format expression is evaluated. If a value of a variable used in the expression changes during execution of the I/O statement, the new value is used the next time the expression is processed.

The following rules apply to the use of variable format expressions.

- The expression is converted to integer before being used, if it is not originally an integer data type.
- The expression can be any valid Stardent 1500/3000 Fortran expression.
- The value of the expression must obey magnitude restrictions that apply to its use in the format.

- These expressions are not permitted in run-time formats.

Example

The following is an example of a variable format expression.

```
FORMAT ( / <x+2>I3 / 2F<Y-3> . <G(2) - SIN(1.7)> )
```

3.3 Format Descriptors

A list of format and edit descriptors makes up a format specification. The format descriptors describe how the data appears and edit descriptors specify editing information.

The descriptors in a format specification must be separated by a comma except before and after a slash (/) edit descriptor, a colon (:) edit descriptor, and between a scaling (**P**) edit descriptor, and an immediately following **F**, **E**, **D** or **G** edit descriptor. Format descriptors may be preceded by a repeat specification.

A format may include another set of format or edit descriptors, or both, enclosed in parentheses; this is called nesting. The nested format specification may be preceded by a repeat specification.

When format descriptors are written without specifying a value for field width, default values are supplied and are based on the data type of the input/output list element. The default values are listed in the table titled *Default Field Width Values for Format Descriptors* later in this chapter. The format descriptors are summarized in Table 3-2 and the edit descriptors in the table labeled *Edit Descriptors* (later in this chapter). A detailed explanation of the descriptors follows the tables.

Table 3-2. Format Descriptors

Format	Explanation
I*w*[.*m*]	Integer and short integer
F*w.d*	Fixed-point format descriptor for real, double precision, complex, and double complex
D*w.d*	Floating-point
E*w.d*[E*e*]	Format descriptor for real, double precision, complex, and double complex
G*w.d*[E*e*]	Fixed or floating-point format for real, double precision, complex, and double complex descriptors
O*w*[.*m*]	Octal, any data type
L*w*	Logical and short logical
A[*w*]	Character data is left-justified in memory and external format
Z*w*[.*m*]	Hexadecimal, any data type

Table 3-3. Default Field Width Values for Format Descriptors

Format Descriptor	List Element	*w*	*d*	*e*
I, O, Z	BYTE	7		
I, O, Z	INTEGER*2,LOGICAL*2	7		
I, O, Z	INTEGER*4,LOGICAL*4	12		
O, Z	REAL*4	12		
O, Z	REAL*8	23		
L	LOGICAL	2		
F, E, G, D	REAL,COMPLEX*8	15	7	2
F, E, G, D	REAL*8,COMPLEX*16	26	17	2
A	LOGICAL*1	1		
A	LOGICAL*2,INTEGER*2	2		
A	LOGICAL*4,INTEGER*4	4		
A	REAL*4,COMPLEX*8	4		
A	REAL*8,COMPLEX*16	8		
A	CHARACTER*n	n		

EXAMPLES

In the following format specification

```
(I3, 3X, 3F12.3)
```

the format descriptor **I3** specifies an integer number with a field width of three (the integer takes up a total of three character positions), the edit descriptor **3X** specifies that three character positions are to be skipped, and the format descriptor **3F12.3** specifies three real numbers, each with a field width of 12 and three significant digits to the right of the decimal point. A **PRINT** statement referencing this format specification could be of the form

```
      PRINT 10, item, a, b, c
10    FORMAT (I3, 3X, 3F12.3)
```

The output data, as it might appear in the output record with the field widths indicated, is shown in the example below.

```
345   65376453.324   14321.265 4765321.321
 |  |      |           |           |
-3--3-------12---------12----------12-----
```

If a slash descriptor is used to indicate a new line of output, or a new record on input, the comma that would separate the descriptors is not necessary. These two are equivalent

```
   3I, F4.0, / I5, F12.6
and
   3I, F4.0 / I5, F12.6
```

The information shown on the input record in the table below could be accessed with a **FORMAT** statement like the following

```
   10 FORMAT (I3,F7.4,3(F7.2,I3),F12.4)

2626.4336 342.26 242373.86439 649.79  4   4395.4972
 |    |      |    |    |    |    |    |       |
-3----7------7----3----7----3----7----3------12-----
```

A **READ** statement corresponding with the FORMAT statement could be

```
     READ 10, i, a, b, j, d, k, e, m, f
```

The **READ** statement would read values for **i** and **a**, then repeat the nested format specification **F7.2,I3** three times to read values for **b** and **j**, **d** and **k**, and **e** and **m**, and, finally, read a value for **f**.

3.3.1 Numeric Format Descriptors

The numeric format descriptors specify the input/output fields of integer, short integer, byte, real, double precision, complex, and double complex data. The following rules apply to all numeric format descriptors:

- The field width, *w*, specifies the total number of characters that a data field occupies, including any leading plus or minus sign, and any decimal point or exponent.
- On input, leading blanks are not significant. Trailing and embedded blanks are ignored only if the **BN** edit descriptor is specified or if **BLANK = NULL** is specified. In all other cases, embedded and trailing blanks are treated as zeros. A field of all blanks is considered to be a **0**. Input data can be separated by commas if shorter than the specified field length.
- On output, the data is right-justified in the field. If the data length is less than the field width, leading blanks are inserted in the field. If the data is longer than the field width for certain descriptors, the entire field is filled with asterisks, as specified in the output examples of the particular descriptors.
- A complex list item is treated as two real items and a double complex list item as two double precision items.
- If a numeric list item is used with a numeric descriptor of different type, the value is converted where possible.

3.3.2 Integer Format Descriptors

The I*w* or I*w.m* format descriptor defines a field for an integer or short integer number. The corresponding input/output list item must be a numeric type. The optional *m* value specifies a minimum number of digits to be output. If *m* is not shown, a default value of 1 is assumed. The *m* value is ignored on input.

On input, the I*w* format descriptor causes the interpretation of the next *w* positions of the input record; the number is converted to match the type of the list element currently using the descriptor. A plus sign is optional for positive values. A decimal point or exponent must not appear in the field.

On output, the I*w* or I*w.m* format descriptor causes output of a numeric variable as a right-justified integer value. The field width, *w*, should be one greater than the expected number of digits to allow a position for a minus sign for negative values. If *m*=0, a **0** value is output as all blanks.

Examples of Integer Format Descriptors

Descriptor	Input Field	Value Stored
I4	␣1␣␣	100
I5	␣␣␣␣␣	0
I2	-1	-1
I4	-123	-123
I3	␣12	12
I2	123	12

Descriptor	Internal Value	Output
I2	+6234	**
I5	-52	␣␣-52
I3	-124	***
I6.3	3	␣␣␣003
I3.0	0	␣␣␣
Output is expressed in integer notation.		

3.3.3 Real and Double Precision Format Descriptors

The F*w.d*, E*w.d*[E*e*], D*w.d*, and G*w.d*[E*e*] format descriptors define fields for real, double precision, complex, or double complex numbers. (Note that two descriptors must be specified for a complex or double complex value.) The input/output list item corresponding to a F*w.d*, E*w.d*[E*e*], D*w.d*, or G*w.d*[E*e*] descriptor must be a numeric type.

The input field for these descriptors consists of an optional plus or minus sign followed by a string of digits that may contain a decimal point. If the decimal point is omitted in the input string, the number of digits equal to *d* from the right of the string are interpreted to be to the right of the decimal point. If a decimal point appears in the input string and conflicts with the descriptor, the decimal point in the input string takes precedence.

This basic form can be followed by an exponent in one of the following forms:

- A signed integer constant.
- An **E** followed by an integer constant.
- A **D** followed by an integer constant.

All three exponent forms are processed in the same way.

The appearance of the output field depends on whether the format descriptor specifies a fixed- or floating-point format.

Examples of Real and Double Precision Descriptors and Input Fields

Descriptor	Input Field	Value
F6.5	4.51E4	45100.0
G4.2	51-3	1.00051
D9.4	␣␣␣45E+35	$.0045\times10^{35}$
BZ,F7.1	-54E34␣	-5.4×10^{340}
		Error (overflow)
F2.10	34	34×10^{10}
where **Value** is the expected number expressed in scientific notation.		

The **BZ** edit descriptor is described later in this chapter in the section, *Blank Interpretation Edit Descriptors—BN and BZ*. Note that the value of *d* is used as a scale factor; it can be greater than the number of digits in the field.

3.3.4 Fixed-Point Format Descriptor

The F*w.d* format descriptor defines a fixed-point field on output for real, double precision, complex, and double complex values. The value is rounded to *d* digits to the right of the decimal point.

Examples of Fixed Point Descriptors and Input Fields

Descriptor	Internal Value	Output
F5.2	+10.567	10.57
F3.1	-254.2	***
F6.3	+5.66791432	␣5.668
F8.2	+999.997	␣1000.00
F8.2	-999.998	-1000.00
F7.2	-999.997	*******
F4.1	23	23.0
F2.0	7.9	8.
F2.1	0.3	.3
F2.0	-7	**

3.3.5 Floating-Point Format Descriptors

The E*w.d*[E*e*] and D*w.d* format descriptors define a normalized floating-point field on output for real, double precision, complex, and double complex values. The value is rounded to *d* digits. The exponent part consists of *e* digits. If E*e* is omitted in an E format, then the exponent occupies two positions. The field width, *w*, should follow the general rule *w d*+7 or, if E*e* is used, *w d*+*e*+5 to provide positions for a leading blank, the sign of the value, (not needed if positive), the decimal point, *d* digits, the letter D or E, the sign of the exponent, and the exponent.

Examples of Floating Point Descriptors and Input Fields

Descriptor	Internal Value	Output
D10.3	+12.342	␣␣.123D+02
E10.3E3	-12.3454	-.123E+002
E12.4	+12.340	␣␣␣.1234E+02
D12.4	-.00456532	␣␣-.4565D-02
D10.10	+99.99913	**********
E11.5	+999.997	␣.10000E+04
E10.3E4	$+.624\times10^{-30}$	.624E-0030

3.3.6 Fixed- or Floating-Point Format Descriptor

The G*w.d*[E*e*] format descriptor defines a fixed- or floating-point field, as needed, on output for real, double precision, and complex values. The G*w.d*[E*e*] format descriptor is interpreted as an F*w.d* descriptor for fixed-point form or as an E*w.d*[E*e*] descriptor for floating-point form according to the magnitude of the data. If the magnitude is less than 0.1 or greater than or equal to 10*******d* (after rounding to *d* digits), the E*w.d*[E*e*] format descriptor is used; otherwise the F*w.d* format descriptor is used. When F*w.d* is used, trailing blanks are included in the field where the exponent would have been. The field occupies *w* positions; the fractional part consists of *d* digits, and the exponent part consists of *e* digits. If E*e* is omitted, then the exponent occupies two positions. The field width, *w*, should follow the general rule for floating-point descriptors *wd*+7 or, if E*e* is used, *wd*+*e*+5 to provide for a leading blank, the sign of the value, *d* digits, the decimal point, and, if needed, the letter E, the sign of the exponent, and the exponent.

Examples of Fixed or Floating Point Descriptor

Field Descriptor	Internal Value	Interpreted As	Output
G10.3	+1234	E10.3	␣␣.123E+04
G10.3	-1234	E10.3	␣-.123E+04
G12.4	+12345	E12.4	␣␣␣.1235E+05
G12.4	+9999	F8.0,4X	␣␣␣9999.␣␣␣␣
G12.4	-999	F8.1,4X	␣␣-999.0␣␣␣␣
G7.1	+.09	E7.1	␣.9E-01
G5.1	-.09	E5.1	*****
G11.1	+9999	E11.1	␣␣␣␣␣.1E+05
G8.2	+9999	E8.2	␣.10E+04
G7.2	-999	E7.2	*******

3.3.7
Character Format Descriptor

The **A**[*w*] format descriptor defines fields for character data. The size of the list variable (byte length) determines the maximum effective value for *w*. If *w* is not specified, the size of the field is equal to the size of the input/output variable.

Using the **A**[*w*] format descriptor for input and output, *w* can be equal to, less than, or greater than the specified byte size of the input or output variable. If *w* is equal to the length of the variable, the character data field is the same as the variable.

3.3.7.1 Contents of Character Data Fields

Input Descriptor	Length of Input Variable	Result
A[*w*]	*w<len*	Left-justified in variable, followed by blanks.
	w>len	Taken from right part of field.

Output Descriptor	Length of Output Variable	Result
A[*w*]	*w<len*	Taken from left part of variable.
	w>len	Output as the value, preceded by blanks.

Examples of Character Input

In the following examples, ␣ represents a blank.

Input Descriptor	Input Characters	Value Length	Characters Stored
A3	XYZ	3	XYZ
A5	ABC␣␣	10	ABC␣␣␣␣␣␣␣
A5	CHAIR	5	CHAIR
A4	ABCD	2	CD

On output, if *w* is greater than or equal to the specified byte size (*len*) of the output variable, the data is right-justified within a *w* character field. If *w* is less than *len*, only the leftmost *w* characters appear in the output field.

Examples of Character Output

Descriptor	Internal Characters	Variable Length	Output
A6	ABCDEF	6	ABCDEF
A4	ABCDE	5	ABCD
A8	STATUS	6	STATUS␣␣
A4	NEXT	4	NEXT

3.3.8 Logical Format Descriptor

The L*w* format descriptor defines a field for logical data. The input/output list item corresponding to a L*w* descriptor must be of type logical, LOGICAL*4, LOGICAL*2, or LOGICAL*1.

On input, the field width is scanned for optional blanks followed by an optional decimal point, followed by a T for true or an F for false. The first nonblank character in the input field (excluding the optional decimal point) determines the value to be stored in the declared logical variable. After the period, if this first nonblank character is other than a T or an F, an error is generated.

Examples of Logical Input Descriptors

Descriptor	Input Field	Value Stored
L5	␣␣␣T␣	.TRUE.
L2	F1	.FALSE.
L2	F1	.FALSE.
L4	␣x␣T	Error
L5	␣RT␣	Error
L7	TFALSE	.TRUE.
L7	.FALSE.	.FALSE.

On output, a T or an F is right-justified in the output field depending on whether the value of the list item is true or false. The logical value true or false is determined by the low order bit in the internal data storage:

`non-zero` = *true*, **0** = *false*.

Examples of Logical Output Descriptors

Descriptor	Internal Format	Value Output
L5	.FALSE.	␣␣␣␣F
L4	.TRUE.	␣␣␣T
L1	.TRUE.	T
L2	.FALSE.	␣F

3.3.9 Octal Format Descriptor

As an extension to Fortran 77, the **O***w* descriptor defines a field for octal data. This descriptor provides conversion between an external octal number and its internal representation.

Legal octal digits are **0**, **1**, **2**, **3**, **4**, **5**, **6** and **7**. Plus and minus signs are not permitted. If any nonoctal digit appears, an error occurs.

On output, the octal value of the I/O list element is transferred as right justified, to a field *w* characters long. If the field is not filled, leading spaces are inserted, and if the number is too large to represent in *w* octal digits, *w* asterisks are printed. When *m* is present, at least *m* digits appear in the field, and then the field is zero filled on the left if necessary.

Examples of Octal Descriptors on Input

Descriptor	Input Field	Value Stored
O5	376	254

Examples of Octal Descriptors on Output

Descriptor	Internal Value Decimal	Output
O2	99	143

3.3.10 Hexadecimal Format Descriptors

As an extension to Fortran 77, the **Z***w* descriptor defines a field for hexadecimal data, and can be used with any data type. The **Z***w* descriptor provides a means of converting hexadecimal data to its internal binary representation, and *vice versa.*

On input, legal hexadecimal digits are **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f,** and **F.** An all- blank field is treated as a value of 0. Plus and minus signs, commas, or any other symbols are not permitted and will be flagged as an error.

Examples of Hexadecimal Format Descriptors

Descriptor	Field	Value Stored
Z4	FF3B	65339
Z2	A9	169
Z3	9.8	error: illegal character

On output, the field width required to fully represent the hexadecimal value of an item is twice its size in bytes. For example, a **CHARACTER*12** item would require a field *w* of 24 characters.

If the value does not fill the field specified, the output is right-justified. Negative values do not generate a minus sign because the hexadecimal value contains the internal representation.

Examples of Hexadecimal Format Descriptors on Output

Descriptor	Type	Value	Output
Z5	CHARACTER*2	171	␣␣␣ab
Z8	INTEGER*4	27	␣␣␣␣␣␣␣1b
Z5	INTEGER*2	27	␣␣␣␣1b

3.4 Edit Descriptors

Edit descriptors specify editing between numeric, Hollerith, and logical fields on input and output records. There are 15 edit descriptors; two, **BN** and **BZ**, apply only to input, and six '...', *n*H, **$**, **S**, **SP**, and **SS**, only to output. The other seven descriptors, *k***P**, **/**, **T***n*, **TL***n*, **TR***n*, **:**, and *n***X** apply to both input and output.

The edit descriptors are summarized below in Table 3-4. A detailed explanation of the descriptors follows the table.

Table 3-4. Edit Descriptors

Descriptor	Function
BN	Ignore blanks
BZ	Treat blanks as zeros
*n*H	Hollerith editing
*n***X**	Skip n positions to the right
T*c*	Skip to column *c*
TR*c*	Skip *c* positions to the right
TL*c*	Skip *c* positions to the left
/	Begin new record
:	Terminate format if remaining list empty
'...'	Literal editing
S	Processor determines sign output
SP	Output optional plus signs
SS	Inhibit optional plus sign output
*k***P**	Scale factor
$	Suppresses carriage return

3.4.1 Blank Interpretation Edit Descriptors

The **BN** and **BZ** edit descriptors control interpretation of embedded and trailing blanks in numeric input fields. At the beginning of the execution of an input statement, blank characters within numbers are ignored. (An exception to this rule is when the unit is connected with **BLANK = 'ZERO'** specified in the **OPEN** statement. Note that **BN** and **BZ** override the **BLANK=** parameter for the current **READ** statement. Refer to Chapter 4, *File Handling* for more details.) If a **BZ** edit descriptor is encountered in the format specification, trailing and embedded blanks in succeeding numeric fields are treated as zeros. The **BZ** edit descriptor remains in effect until a **BN** edit descriptor or the end of the format specification is encountered.

The **BN** and **BZ** edit descriptors affect only **I**, **O**, **F**, **E**, and **G** format descriptors during the execution of an input statement.

Examples of Edit Descriptors

Descriptor	Input Characters	BN Editing in Effect	BZ Editing in Effect
I4	1␣2␣	12	102
F6.2	␣4␣.␣2	4.2	40.02
E7.1	5␣.␣E1␣	$5.*10^{1}$	$50.0*10^{10}$
E5.0	3E4␣␣	$3.*10^{4}$	$3.*10^{400}$ (overflow)

3.4.2 Dollar Sign Edit Descriptor

The **$** edit descriptor modifies the carriage control specified by the first character of the record. This descriptor applies to sequential output only.

When the first character of a record is a space, the carriage return is suppressed. When the first character is a plus sign (+), output begins at the end of the previous line. The print position is then left at the end of that line. The **$** descriptor is ignored if the first character of the record is a zero or one.

3.4.3 Q Edit Descriptor

The **Q** edit descriptor obtains the remaining number of characters to be transferred from an input record during a **READ** operation. The element that corresponds to the **Q** edit descriptor in the **READ** statement must be of type integer or logical.

The length of the input record can be determined by placing a **Q** first in the **FORMAT** specification. This edit descriptor has no effect on output statements.

3.4.4 Plus Sign Edit Descriptors

The **S**, **SP**, and **SS** edit descriptors control printing of optional plus sign characters in numeric output. A formatted output statement does not normally print optional plus signs. However, an **SP** edit descriptor in the format descriptor prints an optional plus sign in any subsequent numeric output. **S** and **SS** descriptors inhibit printing of plus signs in subsequent numeric output.

3.4.5 Literal Edit Descriptors

The '...' (single apostrophe) edit descriptor is used to write a character constant to the output record; literal editing must not be used on input. The width of the field is the number of characters, including blanks, contained between the delimiting apostrophes. When used as edit descriptors, the apostrophes must appear in pairs. If an apostrophe is used inside a literal edit descriptor, it must be doubled.

Examples of Edit Descriptors

Descriptor	Field Width	Output
'BEGIN DATA INPUT'	16	BEGIN DATA INPUT
'␣␣SPACES␣␣'	10	␣␣SPACES␣␣
'Ain''t'	5	Ain't
''''''	1	''
''''	1	"

3.4.6 Literal Edit Descriptor

The *n*H edit descriptor causes character information to be written from the *n* characters, including blanks, following the H of the *n*H edit descriptor in the format specification itself. Note that if an H edit descriptor appears within a character constant and includes an apostrophe, the apostrophe must be represented by two consecutive apostrophes, which are counted as one character when specifying *n*.

Examples of Literal Edit Descriptor

Descriptors	Output
12HJob complete	Job complete
8hPurge?␣␣	Purge?␣␣

3.4.7 Position Edit Descriptor

The *n*X edit descriptor skips *n* positions of an input/output record; *n* must be a positive nonzero integer. On input, the *n*X edit descriptor causes the next *n* positions of the input record to

WARNING
Be careful of the distinction between positioning the cursor with *X,Tn,TLn,* and *TRn* and the actual writing/reading of blanks and/or nulls.

be skipped. The *n*X descriptor is identical to TR*n*.

Examples of Position Edit Descriptor on Input

Descriptors	Input Record	Values Stored
F6.2,3X,I2	673␣21END45	673.21, 45
1X,I2,A3	$6␣END	60, "END"

On output, the *n*X edit descriptor causes the cursor to move, and blanks appear only if there is more output. (If the positions were already defined, they are left unchanged. This can happen when T*c* or TL*c* is used.)

Examples of Position Edit Descriptor on Output

Descriptors	Values	Internal Output
F8.2,2X,I3	15.87, 436	␣␣␣␣5.87␣␣3346
F4.2,3X,'TOTAL'	32.4	32.4␣␣␣␣TOTAL

3.4.8 Tab Edit Descriptors

WARNING
Be careful of the distinction between positioning the cursor with *X,Tn,TLn,* and *TRn* and the actual writing/reading of blanks and/or nulls.

The tab edit descriptors position the cursor on the input or output record. The T*n* edit descriptor references absolute column numbers (*n*), while the descriptors TL*n* and TR*n* reference a relative number of column positions to the left (TL*n*) or right (TR*n*) of the current cursor position. Note that the TR*n* descriptor is identical to the *n*X descriptor.

Examples of Tab Edit Descriptors on Output

Descriptors	Values	Output
T5,F3.1	1.0	␣␣␣␣1.0
F3.1,TR4,F3.2	1.0,.11	1.0␣␣␣␣.11
T10,F3.1,TL12,F3.2	1.0,.11	.11␣␣␣␣␣␣␣␣␣1.0

Example of a Tab Edit Descriptor on Input

Descriptors	Record	Stored
A4,T1,F4.0	1234	"1234", 1234.0

3.4.9 Record Terminator Edit Descriptor

The / (slash) edit descriptor terminates the current record and begins processing a new record. The / edit descriptor has the same result for both input and output.

If a series of two or more / edit descriptors are written at the beginning of a format specification, as many records as there are slashes are skipped. If n, where n is greater than 1, / edit descriptors appear other than at the beginning of a format specification, n records are skipped. If a format contains only n slashes (and no other format specifiers), n+1 records are skipped. The / edit descriptor does not need to be separated by commas from other descriptors.

3.4.10 Colon Edit Descriptor

If there are no more items in the input/output list, the colon (:) edit descriptor terminates format control (just as if the final right parenthesis in the format specification had been reached). If more items remain in the list, the colon edit descriptor has no effect.

Examples of Colon and Record Terminator Edit Descriptors

Format	Stored Value	Output
(10(' value=',I2))	1, 2	value= 1 value= 2 value=
(10(:,'value=',I2))	1, 2	value= 1 value= 2

In the first example, the descriptor "**value=**" is repeated a third time because format control is not terminated until the descriptor "**I2**" is reached and not satisfied. In the second example, the "**:**" descriptor terminates format control before the string "**value=**" is output a third time.

3.4.11 Scale Factor Edit Descriptor

The scale factor, *n*P (*n* is the scale value), is a descriptor that modifies the output of the E*w.d*, D*w.d* and G*w.d* (interpreted as E*w.d*) format descriptors and the fixed-point output of the F*w.d* format descriptor. The scale factor also modifies the fixed-point inputs to the F*w.d*, E*w.d*, D*w.d*, and G*w.d* format descriptors. A scale factor has no effect on the output of the G*w.d* (interpreted as F*w.d*) descriptor floating-point input.

When there is an exponent in the field, the scale factor has no effect on input with **F**, **E**, **D**, and **G** editing. When an exponent does not exist, the effect is that the external number equals the internal number multiplied by 10**n.

With **E** and **D** editing on output, the real constant part of the quantity to be produced is multiplied by 10**n and the exponent is reduced by n. With **F** editing on output, the quantity to be produced is multiplied by 10**n. With **G** editing on output, the scale factor has no effect unless the magnitude of the data being edited is outside the range of **F** editing. If **E** editing is required, the scale factor has the same effect as with **E** output editing.

For example, if a number of data items were stored without decimal points, but were supposed to be interpreted as containing an implied decimal point two positions from the right, using a scale factor of -2 would cause the items to be printed that way. Thus, with the format descriptor **F7.2**, the value **123** would be printed **123.00**, and, with the format descriptor **-2PF7.2**, **1.23**.

When a format specification is interpreted, the scale factor is set to 0. Each time a scale factor descriptor is encountered in a format specification, a new value is set. This scale value remains in effect for all subsequent affected format descriptors until the format specification ends or another **P** descriptor is seen.

EXAMPLES

```
(E10.3, F12.4, I9)
```

No scale factor change; previous value, 0, remains in in effect.

```
(E10.3, 2PF12.4, I9)
```

Scale factor for **E10.3** remains at 0, changes to 2 for **F12.4**, has no effect on **I9**.

On input, the scale factor affects fixed-field (no exponent) input to the F*w.d*, E*w.d*, D*w.d* and G*w.d* format descriptors. The external value is multiplied by 10 raised to the (*n*)th power, as illustrated in the examples that follow.

Scale Factor and Format Descriptor	Input Field	Value Stored
E10.4	␣␣123.9678	123.9678
2PD10.4	␣␣123.9678	1.239678
-2PG11.5	␣␣123.96785	12396.785
-2PE12.5	123967.85E02	123967.85E02 (if the input includes an exponent, the scale factor has no effect)

On output, the scale factor affects F*w.d*, E*w.d*, D*w.d* and G*w.d* (interpreted as E*w.d*) format descriptors only. The scale factor has no effect on the G*w.d* (interpreted as F*w.d*) field descriptor.

For E*w.d*, D*w.d* and G*w.d* (interpreted as E*w.d*) format descriptors, the scale factor has the effect of shifting the decimal point of the output number right *n* places while reducing the exponent by *n* (the value of the mantissa remains the same) as illustrated below. The number of significant digits printed is equal to: (*d+scale factor*).

Scale Factor and Format Descriptor	Stored Value	Input Field
E12.4	12.345678	␣␣␣.1235E+02
3PE12.4	12.345678	␣␣␣123.5E-01
-3PD12.4	12.345678	␣␣␣.0001D+05
1PG10.3	1234	␣␣1.234E+03

For the F*w.d* format descriptor, the internal value is multiplied by 10 raised to the (+*n*)th power, as illustrated below.

Scale Factor and Format Descriptor	Input Value	Value Stored
F11.3	1234.500	␣␣␣1234.500
-2PF11.3	1234.500678	␣␣␣␣␣␣12.345
2PF11.3	1234.500678	␣

The scale factor need not immediately precede its format descriptor. For example, the format specification

```
(3P,I2,F4.1,E5.2)
```

is equivalent to

```
(I2,3P,F4.1,E5.2)
```

If the scale factor does not precede a F*w.d*, E*w.d*, D*w.d* or G*w.d* format descriptor, it should be separated from other descriptors by commas or slashes. If the scale factor immediately precedes a F*w.d*, E*w.d*, D*w.d* or G*w.d* format descriptor, the comma or slash descriptor is optional.

For example, the format specification

```
(I2,3PF4.1,E5.2)
```

is equivalent to

```
(I2,3P,F4.1,E5.2)
```

The scale factor affects all **F**, **E**, **D** and **G** specifications until either the end of the **FORMAT** statement or another scale factor is encountered.

3.5 Unformatted Input/Output

Unformatted input/output allows the transfer of data in internal representation (binary). Each unformatted input/output statement transfers exactly one record. Unformatted input/output to devices is done in "binary" mode. (For more information, refer to Appendix A, *Error Messages*.)

3.5.1 Unformatted Input

Unformatted input is specified by the following input statement

SYNTAX

READ (*unit*, . . . *optional keywords*) *list*

where

unit is the unit number of the file (discussed in Chapter 4, *File Handling*).

list is a list of variables that specifies where the data is to be transferred. If *list* is omitted, the file is moved to the next record without data transfer. The list can contain implied **DO** loops, refer to *Implied DO Loops* under *DO Statement* in Chapter 2, *Fortran Statements*.

None of the optional keywords can be **FMT=**. Refer to Chapter 2, *Fortran Statements* for a detailed description of the syntax and meaning of the optional keywords.

Because only one record is read when an unformatted **READ** statement is executed, the number of list elements must be less than or equal to the number of values in the record; a complex item requires two real or double precision values.

The type of each input value should agree with the type of the corresponding list item. A complex or double complex value in the input record, however, can correspond to two real or double precision list items, or two real or double precision values can correspond to one complex list item. The data is transferred exactly as it was written; thus, no precision is lost.

3.5.2 Unformatted Output

Unformatted output is specified by the following statement

SYNTAX

```
WRITE (unit, . . . optional keywords) list
```

where

unit is the unit number of the file (discussed in Chapter 4, *File Handling*).

list is a list of variables or expressions that specifies the data to be transferred. It can contain implied **DO** loops. For syntax and detailed information on implied DO loops, refer to *Implied DO Loops* under *DO Statement* in Chapter 2, *Fortran Statements*. If *list* is omitted, an empty record is written.

None of the optional keywords can be **FMT=**. Refer to Chapter 2, *Fortran Statements* for a detailed description of the syntax and meaning of the optional keywords.

The output list must not specify more values than can fit into one record. If the specified values do not fill the direct access record, the remainder of the record is undefined. (Because sequential records are variable length, they have no "remainder").

3.6 List-Directed Input

List-directed input is specified by the following input statements:

SYNTAX

```
READ  *, list
READ  (unit, *, . . . optional keywords) list
```

where

unit is the unit number of the file (discussed in Chapter 4, *File Handling*).

list is a list of variables that specifies where the data is to be transferred. If omitted, the file is positioned at the next record without data transfer. The list can contain implied **DO** loops. For syntax and detailed information on implied **DO** loops, refer to *Implied DO Loops* under *DO Statement* in Chapter 2, *Fortran Statements*.

Refer to Chapter 2, *Fortran Statements* for detailed information on the syntax and meaning of the optional keywords.

The first **READ** statement syntax shown above transfers information from the standard input device (unit 5). The second **READ** statement transfers data from a disk file or device. (Files, along with other **READ** statement options, are discussed in Chapter 4, *File Handling*). Input data for list-directed input consists of values separated by one or more blanks, or by a slash or comma preceded or followed by any number of blanks. An end-of-record also acts as a separator except within a character constant. Leading blanks in the first record read are not considered to be part of a value separator unless followed by a slash or comma. Input data can also take the forms

*r***c*

or

*r**

where

r is an unsigned, nonzero, integer constant.

c is a constant.

The *r***c* form means *r* repetitions of the constant *c*, and the *r** form means *r* repetitions of null values. Neither form can contain embedded blanks, except where permitted in the constant *c*.

Reading always starts at the beginning of a new record. As many records as required to satisfy the list are read unless a slash in the input record is encountered.

Embedded blanks in input values are not allowed (they are always interpreted as a value separator). The forms of values in the input record are as follows:

Integers Same form as integer constants (see Chapter 1, *Language Elements*).

Real and Double Precision

Any valid form for real and double precision constants (see Chapter 1, *Language Elements*). In addition, the exponent may be indicated by a signed integer constant (the **D** or **E** can be omitted), and the decimal point can be omitted for those values with no fractional part.

Complex and Double Complex

Two integer, short integer, real, or double precision constants, separated by a comma and enclosed in parentheses. The first number is the real part of the complex or double complex number and the second number is the imaginary part. Each of the numbers can be preceded or followed by blanks or the end of a record.

Logical Consists of a field of characters, the first nonblank character of which must be a T for true or an F for false. There can be an optional leading decimal point.

Character Same form as character constants. (This implies delimiting with apostrophes, which are discarded upon input.) Character constants can be continued from one record to the next; the end-of-record does not cause a blank or any other character to become part of the constant. If the length of the character constant is greater than or equal to the length, *len*, of the list item, only the leftmost *len* characters of the constant are transferred. If the length of the constant is less than *len*, the constant is left-justified in the list item with trailing blanks.

The data in the input record is converted to that of the list item, following the same assignment rules as given in *Table* 2-2 in Chapter 2, *Fortran Statements*.

Encountering an end-of-line (end-of-record) in the input record causes the read to be continued on the next record until the input list items are satisfied. If a slash (/) is encountered, the read terminates and the remaining items in the input list are unchanged. An end-of-record is treated like a blank. An end-of-record is not itself data and is not placed in a character item when a character constant is continued on another line. (That is, character constants can be continued.)

Examples of List-Directed Input

The statement:

```
READ *, s, t, x, y, z
```

and the input record:

␣␣'TOTAL'␣␣2 (42␣,␣1),TRUE␣␣ 2 362␣␣␣563.63D6

cause the following assignments to take place, assuming the variable is of the specified type:

Variable	Type	Value Assigned
s	Character	TOTAL
t	Complex	(42.,1.)
x	Logical	.true.
y	Real	362.
z	Double Precision	563.63×10^6

A null value can be specified in place of a constant when you do not want the value of the corresponding list item to change; if the value is defined, it retains its value or, if the value is undefined, it remains undefined. A null value is indicated by two successive value separators (two commas separated by any number of blanks) or by placing a comma before the first input value on a line.

The statement:

```
READ *, x, y, z
```

and the input record:

␣,5.12␣,␣␣,

cause the following assignments to take place:

Variable	Type	Value Assigned
x	Real	Retains previous value
y	Real	5.12
z	Real	Retains previous value

3.7 List-Directed Output

List-directed output is specified by the following output statements

SYNTAX

```
PRINT *,list
TYPE *,list
WRITE ( unit,*, . . .optional keywords) list
```

where

unit is the unit number of the file (discussed in Chapter 4, *File Handling*).

* is the list-directed formatting specifier, and may also be specified as **FMT=***.

list is a list of variables or expressions that specifies the data to be transferred. If *list* contains a function reference, that function must not contain any **READ** or **WRITE** statements. The list can contain implied **DO** loops. For syntax and detailed information on implied **DO** loops, refer to *Implied DO Loops* under *DO Statement* in Chapter 2, *Fortran Statements*.

Refer to Chapter 2, *Fortran Statements* for details on the syntax and meaning of the optional keywords.

The **PRINT** and **TYPE** statements transfer information to the standard output unit (unit 6). The **WRITE** statement transfers information to external files or devices. (Files, along with other **WRITE** statement options, are discussed in Chapter 4, *File Handling*.) The forms of values in list-directed output records are as follows:

Integers Output as integer constants.

Real and Double Precision
Output with or without an exponent depending on the magnitude of the value.

Complex Output as two numeric values separated by commas and enclosed in parentheses.

Logical A T is output for the value true and an F for the value false.

Character A character value is not delimited by apostrophes and each apostrophe within the value is represented by one character.

Every value is preceded by exactly one blank except character values. Trailing zeros after a decimal point are omitted. A blank character is also inserted at the beginning of each record to provide carriage control when the file is printed. If the field is longer than the number of character positions left in the record, the current record is written and a new one started.

Examples of List-Directed Output

Internal Values	Types
a = 11.15	REAL
b = .11145D	DOUBLE PRECISION
c = (10,3.0)	COMPLEX
d = (1.582D-03,4.9851)	COMPLEX*16
e = .TRUE.	LOGICAL
i = 11250	INTEGER
j = -32799	INTEGER*4
n = 'PROGRAM NAME'	CHARACTER*15
p = 'test.out'	CHARACTER*8

Output Statements	Output Record
PRINT*,11.15	␣11.15␣11250 (Output to the standard output unit (unit 6))
WRITE (6, *) c	␣(10.,3.)
WRITE (6, *) j, e	␣-32799bT
PRINT*,b	␣1.1145E-6
WRITE (6, *) d	␣(1.582E-3,4.9851)
WRITE (6, *) n, p	PROGRAM␣NAME␣␣␣␣test.out

3.8 NAMELIST-Directed I/O

NAMELIST-directed I/O allows the transfer of data without specifying the exact format or order of the data. Data is labeled with the variable names. Variables eligible for **NAMELIST**-directed I/O are defined in **NAMELIST** groups. You declare a **NAMELIST** group containing several variables. Subsequent **READ** and **WRITE** statements may do **NAMELIST**-directed I/O by referencing the **NAMELIST** group name.

NAMELIST-directed I/O may occur only to or from sequential external formatted files. I/O to or from direct access, internal, or unformatted files is not supported.

EXAMPLE

```
PROGRAM MAIN
INTEGER I, J(10)
CHARACTER*10 C
NAMELIST /N1/ I, J, C
READ (UNIT=5, NML=N1)
WRITE (6, N1)
END
```

Consider the following input (note that column indicators are **not** part of the input data. All input data must begin in column 2 in your standard input device; refer to syntax rule #1 specified in the *NAMELIST-Directed Input* in the following section):

```
(column   1         2         3...)
(12345678901234567890123456789O...)

 $N1
 J(8) = 6, 7, 8
 I= 5
 C = 'xxxxxxxxxx'
 J = 5*0, -1, 2
 C(2:6) = 'abcde'
 $END
```

The output are (assuming above input):

```
 $N1
 I =  5,
 J =  0, 0, 0, 0, 0, -1, 2, 6, 7, 8,
 C = 'xabcdexxxx'
 $END
```

3.8.1 NAMELIST-Directed Input

NAMELIST-directed input is specified by the following statements:

```
READ nml
or
READ (unit, nml, . . . optional keywords)
```

where

unit is the unit number of the file.

nml is the *group-name* namelist specifier.

The first **READ** statement syntax shown transfers information from the standard input device (unit 5). The second **READ** statement transfers data from a disk file or device.

No variable list is specified in a **NAMELIST**-directed **READ** statement. The **NAMELIST** group name declaration determines which variable values may appear in the input stream.

Input data for **NAMELIST**-directed input has the following form:

```
$group-name variable=value [ ,<variable=value>, . . .] $[END]
```

where

$ is the special symbol used to delimit the beginning and end of **NAMELIST** input. The ampersand symbol (&) may be substituted for the $.

group-name is the **NAMELIST** group name specified in the **READ** statement.

variable is a variable name appearing in the **NAMELIST** declaration for <*group name*>. It may be optionally followed by subscript and/or substring specifiers, that is, *variable*[(*subscripts*)][(*substring*)].

value is a list of zero or more constants separated by commas, blanks, tabs, or newlines. The constants take the same form as those for list-directed input including *count***constant* and *count** repetition factors. The type of the constant must match the type of the variable.

END is an optional ending for the terminating $.

Syntax rules for **NAMELIST**-directed input data:

(1) Reading always starts at the beginning of a new record. Records are read until a "$" or "&" is found in column 2, immediately followed by a **NAMELIST** group name matching the one specified in the **READ** statement.

(2) All **NAMELIST** group name and variable name matching is case-insensitive (mapped to uppercase).

(3) At least one blank, tab, or newline character must separate the $*group-name* from the first variable name.

(4) Variables may appear in the input data in any order. Any variable values not defined by the input will remain unchanged.

(5) No characters are permitted between the variable name and the "(" of the optional subscript. A subscript specifier has the form

(*sub1*,*sub2*,...,*subn*).

Each subscript must be an integer constant. Blanks, tabs, or newline characters may appear before and after each subscript. A single comma must appear between each pair of subscripts. The number of subscripts must match the number of array dimensions of the variable. Each subscript must be within the bounds of its dimension.

(6) No characters are permitted between the variable name and the "(" of the optional substring specifier. If there is a subscript specifier, the substring "(" must immediately follow the ")" closing the subscript list. A substring specifier has the form

(*position1*:*position2*).

Each position specifier must be an unsigned integer constant. Blanks, tabs, or newlines may appear before and after each position specifier. If the first position specifier is omitted, it is assumed to be 1. If the last position specifier is omitted, it is assumed to be the length of the **CHARACTER** variable. The position specifiers must obey the inequality:

1char1char2variable-length

A substring specifier may appear only with a variable of type **CHARACTER**.

(7) Blanks, tabs, or newlines may appear before and after the = separating the variable name from the value list.

(8) Constants in the value list are separated by blanks, tabs, newlines, or a single comma.

(9) Values in the value list are placed in ascending memory locations starting with the one specified by the variable with optional subscript and substring specifiers.

Example: INTEGER*4 I(4)Assume I is in a **NAMELIST** group and **NAMELIST** input is

```
I(2) = 1,2
```

Then **I(2)** receives the value 1 and **I(3)** receives the value 2.

(10) The number of constants in the value list must not exceed the remaining memory locations in the variable.

Example: INTEGER*4 I(4) with **NAMELIST** input:

```
I(2) = 1,2,3,4
```

generates a run-time error.

(11) Repetition counts of the form *count***constant* or *count** must have no characters between the unsigned integer *count* and the *, or between the * and the first character of the *constant*.

(12) Null constants may be specified by a leading comma, a trailing comma, two successive commas, or the *count** null repetition form. Null constants cause memory locations in the target variable to be skipped; the contents are not altered in any way.

(13) Constants take the same form as in list-directed input. Hollerith, binary, octal, and hexadecimal constants are not permitted.

(14) Character constants may be split across input records by immediately following the last character on a record by a newline (don't close the string with ' or "). Leave the first character of the next record blank (for carriage control) and resume the string in column 2. No blank characters are inserted in the string.

(15) Logical constants consist of a string of characters, the first of which is **t**, **T**, **f**, or **F**. A leading period is optional.

(16) **PARAMETER** names may not appear in **NAMELIST**-directed input.

(17) If the type of the constant does not match that of the receiving variable, type conversion takes place. See the *Assignment Statement* section for the valid type conversions.

EXAMPLE

```
PROGRAM MAIN
INTEGER IARY(4)
CHARACTER*3 CARY(3,2)
LOGICAL LOG
REAL R
NAMELIST /L1/ IARY, CARY, LOG, R
READ(*,L1)
WRITE(6,L1)
END
```

Consider the following input data (note that column indicators are **not** part of the input data. All input data must begin in column 2; refer to syntax rule #1 specified in *NAMELIST-Directed Input*):

```
(column   1         2         3...)
(12345678901234567890123456789O...)

 &L1 IARY = 4,3,2,1
 LOG = T,
 CARY = 6*'XXX'
 R = 5.75E25, CARY(3,2)(1:2) = 'ab'
 $END
```

After execution, variables have the following values:

```
IARY(1)    = 4
IARY(2)    = 3
IARY(3)    = 2
IARY(4)    = 1
CARY(1,1) = 'XXX'
CARY(2,1) = 'XXX'
CARY(3,1) = 'XXX'
CARY(1,2) = 'XXX'
CARY(2,2) = 'XXX'
CARY(3,2) = 'abX'
LOG        = T
R          = 5.75E+25
```

3.8.2 NAMELIST-Directed Output

NAMELIST-directed output is specified by the following statements

```
PRINT nml
TYPE nml
or
WRITE (unit,nml,...optional keywords)
```

where

unit is the unit number of the file.

nml is the *group-name* namelist specifier.

The **PRINT** and **TYPE** statements syntax shown transfers information to the standard output device (unit 6). The **WRITE** statement transfers data to a disk file or device.

No variable list is specified in a **NAMELIST**-directed **PRINT** or **WRITE** statement. The **NAMELIST** group name declaration determines which variable values will be printed to the output file. The order of the values is also determined by the **NAMELIST** declaration. **NAMELIST**-directed output is in a form suitable for **NAMELIST**-directed input. The format of each value is the same as for list-directed output.

EXAMPLE

```
PROGRAM MAIN
INTEGER IARY(4)
CHARACTER*3 CARY(3,2)
LOGICAL LOG
REAL R
NAMELIST /L1/ IARY, CARY, LOG, R
READ(*,L1)
```

```
WRITE(6,L1)
END
```

With the following input data (note that the column indicators are **not** part of the input data; all inputs must begin in column 2):

```
(column   1         2         3...)
(12345678901234567890123456789O...)

 &L1 IARY = 4,3,2,1
 LOG = T,
 CARY = 6*'XXX'
 R = 5.75E25, CARY(3,2)(1:2) = 'ab'
 $END
```

The following is printed:

```
(column   1         2         3...)
(12345678901234567890123456789O...)

 $L1
 IARY =  4, 3, 2, 1,
 CARY = 'XXX','XXX','XXX','XXX','XXX','abX',
 LOG =  T,
 R =  5.7500000E+25
 $END
```

3.9 Special Programming Considerations

The items that follow should be noted and taken into account when programming.

3.9.1 Recursion in I/O

It is illegal for an I/O specifier or an item in an I/O list to call a function that does any I/O. This is considered recursive I/O, because a statement that invoked the I/O library is now trying to reinvoke it before finishing the original I/O call. A compiler and run-time error is issued if recursive I/O is attempted.

This restriction requires that if you use explicit parallelism, you must lock on a semaphore when doing I/O, to avoid multiple processors being in the I/O library at once. Refer to Chapter 8, *Stardent 1500/3000 Fortran Optimization Facilities* under the section called *User-Controlled Parallelism* for additional information.

EXAMPLE

```
      ...
C Recursive I/O statement (because FUNCTION 'change' does I/O)
      WRITE (internal_file, '(A)') change()
      ...
      CHARACTER*10 FUNCTION change()
C Following statement causes recursive I/O run-time error
      OPEN(20,file='CHGFILE')
      ...
      RETURN
      END
```

3.9.2 ASA Carriage Control

The program **asa** (also described in Chapter 10, *User Commands*, in the section called *FPR(1)*) interprets the output of a Fortran program that uses **ASA** carriage control characters. It processes either the file name arguments or the standard input if no arguments are supplied. The output appears on standard output. The first character of each line is interpreted as a control character. Lines beginning with any other characters are interpreted as if they begin with a blank and an appropriate diagnostic appears on standard error. The first character of each line is **not** printed. Each input file begins on a new page.

SYNTAX

asa [*filenames*]

where

filenames is a list of file names to be output with carriage control characters interpreted according to ASA rules.

The carriage control characters and their meanings are:

(blank)	single new line before printing
0	double new line before printing
1	new page before printing
+	overprint previous line

To view the output of programs that use **ASA** carriage control characters, **asa** should be used as a filter. The following command directs the output, properly formatted and paginated, to the line printer:

```
a.out | asa | lpr
```

FILE HANDLING

CHAPTER FOUR

The input/output statements, (**READ**, **WRITE** and **PRINT**), described in Chapter 3, *Fortran I/O Statements*, reference a unit number for a file. The unit number refers to a Fortran logical unit number assigned to a disk file or to a peripheral input/output or storage device. Fortran allows access to disk files and nondisk units through two methods, and both assign a Fortran logical unit number to the unit. The two methods are as follows:

- Preconnection uses the unit numbers 5, 6 and 0 which are referred to as, respectively, standard input, standard output, and standard error. This method does not require the use of the **OPEN** statement.

- Assignment of a logical unit number within an **OPEN** statement. The **OPEN** statement also can be used on devices to assign standard Fortran unit numbers or to change certain specifications.

This chapter provides details on the types of files, methods of assigning a unit number, control specifications, and other file status and manipulation procedures. The syntax and a limited discussion of each file handling statement appears in Chapter 2, *Fortran Statements*.

In addition, some run-time controls can be exercised over the I/O facilities by setting specific environment variables. These variables, described at the end of this chapter, allow you to control the file format used to read or write unformatted files.

4.1 File Definition

A file in Fortran is defined to be a collection of related information logically organized into records. A file can be stored on disk or can reference nondisk peripheral devices by device file name. Files on the UNIX operating system have no internal structure (that is, no records); a UNIX file is a collection of bytes. Languages (such as Fortran) impose a structure on the file by dividing the file into records. The information in files can be programs or data. For a detailed discussion of the types of files available, refer to the appropriate system reference manual.

A record is defined as a sequence of data values or characters. A record does not necessarily refer to a physical entity (such as a punched card), but refers to a logical representation of data or characters.

The three types of records are:

- Formatted.
- Unformatted.
- End-of-file.

A formatted record consists of data that is edited during both the input and output processes. The length of a formatted record is measured in characters . Formatted records should be read or written only by formatted input/output statements. There is no maximum limit to the length of a formatted record.

An unformatted record consists of data that can be read or written without incurring the overhead of the editing process. The length of an unformatted record is measured in 4-byte words. An unformatted record should only be read or written by unformatted input/output statements.

An end-of-file record is the last logical record of a sequential file. This record is written by the **ENDFILE** statement and contains no data.

The terms **external file** and **internal file** are defined as

external file

a file located on a storage medium external to the program (such as disk) or an external device.

internal file

an area of storage, internal to the program, such as an array in main storage.

Operations involving the movement of data from one internal storage area to another, with the ability to convert data from one form to another, are facilitated by the use of internal files. (Refer to *Internal Files* later in this chapter.)

This chapter uses these terms to specify positioning within a file:

current record

is the record within which the pointer is currently positioned.

file pointer

is the current position within a file.

initial point

is the position just before the first record of the file.

next record

is the next record to be read or written; if the file pointer is at the terminal point, there is no next record.

preceeding record

is the record just read or written; if the file pointer is at the initial point, there is no preceding record.

terminal point

is the position just after the last record of the file.

4.1.1 Sequential Formatted File Format

Records of sequential formatted external files are separated by a newline character.

4.1.2 Sequential Unformatted File Format

Each record of a sequential unformatted file is preceded and succeeded by four bytes that contain the record length.

4.1.3 Direct Formatted File Format

Records of direct formatted files are blank-filled to the specified length of the record. Records are not physically separated.

4.1.4 Direct Unformatted File Format

Records of direct unformatted files are null-filled to the specified length of the record. Records are not physically separated.

4.2 File Access

External Fortran files are categorized by the method of access; either sequential access or direct access.

Sequential access refers to the access of records in the order in which they were written. A sequential file may be consisted of either formatted records or unformatted records, but not both. It totally depends on how the file is written.

Direct access refers to the access of the records in any order by record number. Reading and writing records is accomplished by direct access input/output statements (that is, **READ** and **WRITE** statements containing a **REC=** specification). Each record of the file is identified by a record number that is a positive integer. Once established, a record number of a specific record cannot be changed nor may the record be deleted, although the record may be rewritten. The records may be read or written in any order. For example, record 3 may be written prior to writing record 1. The records of a direct access file cannot be read or written using list-directed formatting. A direct access file does not contain an end-of-file record as an integral part of the file with a specific record number; therefore, when accessing a file with a direct access read or write statement, the **END=** specification is not allowed.

4.3 File Existence and Connection

A file is said to exist for an executable program if the program can access it. There are also cases where the file may exist and be known to the processor, but not to the executable program. At any one time, there is a specific set of files that exist for a program. All input/output statements can refer to files that exist, while the **INQUIRE**, **OPEN**, and **CLOSE** statements can also refer to files that do not yet exist for the program.

Files (devices) that are preconnected for use by a program can be accessed in **READ** or **WRITE** statements without prior execution of an **OPEN** statement. For more detailed information on the system dependent devices, refer to the appropriate system reference manual.

A unit cannot be connected to more than one file at the same time, and conversely a file cannot be connected to more than one unit at the same time. If a unit is disconnected in a program by a **CLOSE** statement, the unit number is available to be reconnected to the same file or connected to a different file in the program. Similarly, a particular file that is disconnected by a **CLOSE** statement can be reconnected to the same unit or to a different unit.

Note that the only means of referring to a disconnected file in an **OPEN** or **INQUIRE** statement is by name; therefore, if a scratch file is disconnected, it cannot be reconnected and data reclaimed (because scratch files are purged on **CLOSE** or program termination).

The following input/output and file positioning statements must reference a unit that is connected:

ACCEPT inputs data from external records.

BACKSPACE
moves the file pointer of the connected sequential file to the position before the previous record.

ENDFILE writes an end-of-file record as the next record of the sequential file.

PRINT outputs data to the preconnected default output device file.

READ inputs data from a connected unit.

REWIND moves the file pointer of the connected sequential file to the initial point of the file.

TYPE outputs data to external records.

WRITE outputs data to a connected unit.

The following three file control statements can reference a unit that either is connected or not connected:

CLOSE disconnects a unit from a file (A **CLOSE** on an unconnected unit has no effect.)

INQUIRE requests information about the properties of a particular named file or of the connection to a particular unit (inquire either by file name or by unit number).

OPEN connects a file to a unit, (possibly) creates a file and connects it to a unit, or changes certain specifiers of a connection between a file and a unit.

4.4 File Control Specifiers

File control specifiers are used with various file input/output statements or file positioning statements. Some of the specifiers can be used with any file manipulation statement, while others have meaning only in particular statements.

This section describes the file control specifiers allowed in each of the file input/output statements, and describes any restrictions on their position or occurrence. Detailed syntax requirements of the file control statements are described in Chapter 2, *Fortran Statements*.

4.4.1 READ and WRITE Statements

The following control specifiers have meaning in the **READ** and **WRITE** statements:

END	= *endlabel*
ERR	= *label*
FMT	= *fmt*
IOSTAT	= *ios*
NML	= *namelist*
REC	= *rn*
UNIT	= *unit*

The specification list must include exactly one unit specifier and at most one each of the other specifiers. If a **REC=** specifier appears, the statement is a direct access request. On a direct access request, the **END=** specifier must not appear. The **END=** specifier is never allowed in a **WRITE** request. Also, the **FMT=** specifier must not indicate list-directed formatting (*) with a direct access file.

There are no positional requirements for the specifiers if all the keywords are shown in the input/output statement. If the keyword **UNIT=** is omitted, the unit number specifier must appear first in the control list. If the optional keyword **FMT=** is omitted, the format specifier must be the second item in the list, following a unit specifier without the optional keyword **UNIT=**.

EXAMPLES

```
READ(5,33) a, b, c
READ(UNIT=5, FMT=33) a, b, c
```

The first two examples are identical: requesting input from logical unit **5** controlled by the format statement labeled **33**.

```
WRITE(17, 11, REC=irec, IOSTAT=ios, ERR=99) x, y, z
```

The third example specifies an output request to a direct access unit **17**, writing record number **irec** as specified in the format statement labeled **11**. If an error occurs, control transfers to the statement labeled **99** and the error code is stored in the variable **ios**.

4.4.2 OPEN Statement

The following control specifiers have meaning in the **OPEN** statement:

ACCESS	= *acc*
BLANK	= *blnk*
ERR	= *label*
FILE	= *name*
FORM	= *fm*
IOSTAT	= *ios*
RECL	= *rcl*
STATUS	= *sta*
UNIT	= *unit*

The **OPEN** statement connects a unit number to a file or changes certain features of the connection between a file and a unit. When a file is opened, the file pointer is positioned at the beginning of the file. A redundant **OPEN** does not affect the current position of the file. For a detailed description of the syntax and meaning of control specifiers, refer to Chapter 2, *Fortran Statements*.

EXAMPLES

The following statement connects Fortran logical unit number **9** to a file called **dat**. If an error is encountered in the **OPEN**, save the error code in the variable **ios** and transfer to statement **99**.

```
OPEN(9, IOSTAT=ios, ERR=99, FILE='dat')
```

(If file **dat** exists, it is connected to unit number **9**. If **dat** does not exist, it is created as a sequential access file in the user's directory).

The following statement creates a file called **fil1**. Connect the new file to Fortran unit number **2**. Handle any **OPEN** errors the same way as in the previous example.

```
OPEN(2, FILE='fil1', STATUS='NEW', IOSTAT=ios, ERR=99)
```

The specification list must contain exactly one unit specifier and at most one each of the other specifiers. There is no positional hierarchy among the control specifiers, with one exception: when the control specifier **UNIT=** does not appear, the unit number must appear as the first specifier in the control list.

The **FILE=** name specification must be present for all named disk files to be created or opened. (Refer to the appropriate system reference manual for more detailed information on the file name.)

The **STATUS=***sta* specifier determines whether or not the file must already exist. If **'OLD'** is specified, the file must already exist, while if **'NEW'** is specified, an error is generated if the file exists. The **'NEW'** specification directs that the file is to be created. If **'SCRATCH'** is specified, a file name must not be specified, and a file with a unique name is created. If no **STATUS=***sta* specifier is shown, the status defaults to **'UNKNOWN'**. This means to search for the file and to create a new file if it does not exist.

All the specifiers except the unit specifier are optional. If the file is opened with **ACCESS='DIRECT'**, the **RECL** specifier must be present to determine the record length of the file.

A file can be connected to a unit number by an **OPEN** statement in any program unit of an executable program, and once connected to a unit, can be referenced in any program unit.

The **OPEN** statement connects existing files to a unit number, creates and connects a named or scratch file, or changes the control specifiers on a file already connected to a unit by referencing the same file name or omitting it and specifying different

characteristics to the file. This is effectively the same as executing a **CLOSE** statement on the file that was previously connected to the unit number referenced in the **OPEN** statement. Conversely, once a file has been connected with a unit number, that file cannot be connected with a different number until the file is closed.

The specifiers that have default values if omitted from the control list are:

STATUS = 'UNKNOWN'
ACCESS = 'SEQUENTIAL'
BLANK = 'NULL'

EXAMPLES

Open a scratch file for direct access, with a user-defined record length of 80 characters, defaulting the remaining parameters:

```
OPEN(13, ACCESS='DIRECT', RECL=80, STATUS='SCRATCH')
```

Connect a file named **out1** to logical unit number **1**. The file **out1** exists as a sequential file for formatted input/output. Specify that all blanks should be treated as zeros.

```
OPEN(1, FILE='out1', STATUS='OLD', BLANK='ZERO')
```

Connect a direct access file named **tree** with a record size of 80 characters.

```
OPEN(8, FILE='tree', ACCESS='DIRECT', RECL=80)
```

Connect a direct access scratch file to Fortran unit **4**.

```
OPEN(4, STATUS='SCRATCH', ACCESS='DIRECT')
```

4.4.3 CLOSE Statement

The following control specifiers have meaning in the **CLOSE** statement:

ERR = *label*
IOSTAT = *ios*
STATUS = *sta*
UNIT = *unit*

The **CLOSE** statement terminates the connection of a unit to a file. For a detailed description of the syntax and meaning of each of the control specifiers, refer to Chapter 2, *Fortran Statements*.

Specifying a **STATUS** of **KEEP** or **DELETE** determines if the file will continue to exist or will be purged from the disk. A file whose status is **SCRATCH** is always deleted by the system when the file is closed or at normal program termination, even if a **CLOSE** statement is executed specifying **STATUS='KEEP'**. For named files, if the **STATUS=** specifier is omitted, the default specification is **KEEP**.

Stardent 1500/3000 Fortran offers extensions to Fortran 77 for the **STATUS** specifier, in order to maintain compatibility with other operating systems. These additional **STATUS** specifiers are **SAVE, PRINT, SUBMIT/DELETE,** and **PRINT/DELETE**. If the word **DELETE** is present the file will be deleted, otherwise the file is saved, as in specifying **KEEP**.

A **CLOSE** statement referencing a unit that does not exist can be executed, but no action is taken. If a file is not closed by a **CLOSE** statement in the execution of a program, the file is closed automatically upon normal program termination.

EXAMPLE

```
CLOSE(16, IOSTAT=ios, ERR=99, STATUS='DELETE')
```

This example disconnects the file that was connected to unit number **16** and specifies that the file should be deleted. If an error occurs, control transfers to the statement labeled **99** and the error code is stored in the variable **ios**.

4.4.4 INQUIRE Statement

The following control specifiers have meaning in the **INQUIRE** statement:

ACCESS	= *acc*
BLANK	= *blnk*
CARRIAGECONTROL	= *cc*
DIRECT	= *dir*
ERR	= *label*
EXIST	= *ex*
FILE	= *name*
FORM	= *fm*
FORMATTED	= *fmt*
IOSTAT	= *ios*
NAME	= *fn*
NAMED	= *nmd*
NEXTREC	= *nr*
NUMBER	= *num*

OPENED	= *od*
RECL	= *rcl*
SEQUENTIAL	= *seq*
UNFORMATTED	= *unf*
UNIT	= *unit*

The **INQUIRE** statement requests properties of a file or device by either specifying the unit number or by specifying the file name in the control list. The **INQUIRE** statement can return information on a file that is not connected to a unit, as well as on a connected file or device.

The **INQUIRE**-by-file version of the **INQUIRE** statement requires exactly one file name specifier and any of the other specifiers except the unit specifier. The **INQUIRE**-by-unit version requires exactly one unit specifier and one each of the other optional specifiers as desired, excluding the file name specifier. Refer to Chapter 2, *Fortran Statements* for a general description of each of the control specifiers. Table 4-1 lists all of the possible combinations for each specification in the **INQUIRE** statement.

CAUTION
It is true that you can execute a Fortran INQUIRE statement on an unopened file in the VMS system because VMS appears to carry around its file information. Unfortunately, BSD and System V systems do not implement this kind of handling. If you execute an INQUIRE statement on UNIX data files, be sure that you open those files first (by using OPEN statement).

Table 4-1. INQUIRE Statement Specifications

KEYWORD	UNOPENED UNIT	UNOPENED FILE	NONEXISTENT FILE	OPENED UNIT
EXIST	false	true	false	true
OPENED	false	false	false	true
NAMED	true	true	true	false, if a scratch file; true otherwise
NUMBER	FTN unit	undefined	undefined	FTN unit
NAME	undefined	file name	file name	file name (if named)
ACCESS	undefined	undefined	undefined	see **OPEN**
SEQUENTIAL	UNKNOWN	UNKNOWN	UNKNOWN	see **OPEN**
DIRECT	UNKNOWN	UNKNOWN	UNKNOWN	see **OPEN**
ASYNC	UNKNOWN	UNKNOWN	UNKNOWN	see **OPEN**
FORM	undefined	undefined	undefined	see **OPEN**
FORMATTED	UNKNOWN	UNKNOWN	UNKNOWN	see **OPEN**
UNFORMATTED	UNKNOWN	UNKNOWN	UNKNOWN	see **OPEN**
BLANK	undefined	undefined	undefined	see **OPEN**
RECL	undefined	undefined	undefined	bytes (if direct)
NEXTREC	undefined	undefined	undefined	record number (if file)

In general, upon execution of an **INQUIRE**-by-file statement, if the file name is illegal or if the file does not exist, the specifiers **NAMED, NAME, SEQUENTIAL, DIRECT, FORMATTED** and **UNFORMATTED** are defined. If the file exists and is connected to a unit, *ex* and *od* return *true, num* becomes defined, and the variables *acc, m, rcl, nr* and *blnk* become defined if they are included in the **INQUIRE**-by-file statement. Upon execution of an **INQUIRE**-by-unit statement, if the unit exists and is connected to a file, the specifiers *num, nmd, fn, acc, seq, dir, fm* and *blnk* become defined.

The specifiers **EXIST=***ex* and **OPENED=***od* always become defined with a *true* or *false* value if no error condition is encountered.

EXAMPLE

```
INQUIRE(FILE='exfl',IOSTAT=ios,ERR=99,EXIST=ex,
 OPENED=iop,NUMBER=num,ACCESS=acc)
```

This example requests information on the specified properties of the file named **exfl**. If **exfl** exists and is connected to a unit in the program, the variables **ex** and **iop** return the value *true* , the unit number is stored in **num**, and the character variable **acc** is defined. If **exfl** does not exist, **ex** and **iop** return the value *false,* **ios** is 0 if there was no error, **ios** is a system- determined value greater than 0 if there was an error, and the other specifiers are not defined.

4.5 File Positioning Statements

The **BACKSPACE, REWIND,** and **ENDFILE** statements control the position of the file pointer within a file. The following specifiers have meaning in these statements:

UNIT	= *unit*
IOSTAT	= *ios*
ERR	= *label*

Exactly one external unit specifier must be included in the control list of the file positioning statements. The unit specified must be connected for sequential access. The **IOSTAT** and **ERR** parameters are optional.

The **BACKSPACE** statement causes the file pointer to be positioned before the preceding record. As a Stardent 1500/3000 Fortran extension, backspacing over records written using list-

directed formatting is permitted. If the file is connected but does not exist, the **BACKSPACE** statement is not allowed.

The **REWIND** statement causes the file pointer to be positioned at the initial point of the file. The **BACKSPACE** and **REWIND** statements are for sequential files and are not allowed on files connected for direct access.

The **ENDFILE** statement writes an end-of-file record as the next record of the file. If an **ENDFILE** statement is executed on a file that is connected but does not exist, the file is created. The **ENDFILE** statement is not allowed on files connected for direct access.

EXAMPLES

```
BACKSPACE 10
```

Moves the file pointer of unit **10** to the previous record.

```
REWIND (13,IOSTAT=ios,ERR=99)
```

This example moves the file pointer to the initial point in the file connected to logical unit **13**. If an error occurs, control is transferred to statement **99** and the error code is stored in the variable *ios*.

NOTE
If the 2nd and 3rd examples appeared in sequence in a program unit, the effect would be to delete the information in the file.

```
ENDFILE 13
```

Writes an end-of-file record as the next record of the file connected to unit number **13**.

4.6 Internal Files

Internal files provide a means of memory-to-memory data transfer and are used to perform formatting operations on character variables. An internal file can be a character variable, a character array element, a character substring, or a character array. Each variable or array element is considered to be one record.

An internal file is accessed by a sequential formatted input/output statement. The name of the internal file appearing as the value of the **UNIT** parameter identifies the file.

EXAMPLE

```
      WRITE(UNIT=address_var,FMT='(I10)') street_address
```

writes the value of the variable **street_address** into the first **10** positions of the internal file **address_var**. (**address_var** must be a variable or array of type **CHARACTER**). If **address_var** has a length greater than **10**, the rest of the record is filled with blanks.

Another example of writing to a character variable is shown in this program, followed by the program's output:

```
      PROGRAM in1
      CHARACTER*14 ifmt
      INTEGER iarray(5)
      DATA  iarray/1,2,3,4,5/
      n = 10
      m = 5
      WRITE (ifmt,10) n,m
 10   FORMAT ('(',I2,'X,',I2,'(I2,X))')
      WRITE (6,ifmt) iarray
      WRITE (6,*) ifmt
      END
```

The results of running this program are:

```
    1  2  3  4  5
 (10X, 5(I2,X))
```

In addition to becoming defined by a **WRITE** statement, a record of an internal file can become defined in an assignment statement as shown below. This example shows that an internal file record can be manipulated in the same way as any other variable.

```
      CHARACTER bufr*20
      READ(10,'(A)') bufr                ! Input into bufr
      READ(bufr,'(I10)',ERR=99) value    ! bufr numerical?
      IF (value .LT.  0) THEN. . .       ! Yes
C       (more)
   99 IF (buffer .EQ.  'end') THEN. . .! No
```

Internal files should be used in place of the **DECODE** and **ENCODE** statements of some earlier versions of Fortran.

4.7 Preconnected Files

When a program is started, three files are opened automatically, or *preconnected*. In Stardent 1500/3000 Fortran, unit numbers 5, 6 and 0 are initially preconnected to the UNIX standard input, output and error units respectively. These preconnected files are all connected for sequential formatted I/O. They are normally connected to the terminal, but can be redirected to files or pipes, as described in the appropriate system reference manual.

Standard preconnected files remain open for the duration of Fortran program execution. A **CLOSE** statement executed on any of these has no effect.

A standard unit number (5, 6 or 0) can be reused by performing an **OPEN** statement that assigns it to a new file. However, error messages and other items may still be written to standard output or standard error and some control information may be received from standard input (that is, after **PAUSE**).

4.8 General File Examples

The following examples demonstrate the use of several options of the file manipulation statements.

Example 1

The following program computes the mean of all the data items in the disk file **dat**. The file contains an unknown number of records, each containing one real number.

```
      PROGRAM mwfi1
      sum = 0.0      !Initialize
      n = 0          !Initialize
      OPEN (3, IOSTAT = ios, ERR = 99, FILE = 'dat',
     +  ACCESS='SEQUENTIAL',STATUS='OLD')
   10 READ (3,22, END = 88, IOSTAT = ios,
     + ERR = 99) anum
   22 FORMAT (F10.5)
      sum = sum + anum  !Add data entries
      n = n + 1         !Count entries
      GO TO 10          !Loop

C Out of loop

   88 avg = sum/n
      WRITE(6,33) avg   !Output to preconnected terminal
   33 FORMAT ('The average is ', F12.6)
      CLOSE (3)
      STOP

C If output error in the OPEN or READ,
C output to a preconnected terminal.

   99 WRITE(6,44) ios
   44 FORMAT ('Error encountered =',I6)
      END
```

If file **dat** contains

```
1.0
2.0
3.0
4.0
```

Running **mwfi1** produces

```
The average is      2.500000
```

However, if file **dat** does not exist, running **mwfil** produces

```
Error encountered = 69
```

Example 2

The following example inserts a single number data entry in the proper position in a sorted sequential file. A direct access scratch file stores the temporary results prior to rewriting the original data file.

```
      PROGRAM mgw
C  Declare and initialize variables
      IMPLICIT NONE
      REAL anum, fnum
      INTEGER nrec,wrec,ios1,ios2,i
      nrec = 1
      wrec = 1
      ios1= 0
C  Open the scratch file to 17 and the sequential data file to 18
      OPEN (18,FILE='mwdata',STATUS='UNKNOWN',
     X IOSTAT=ios1,ERR=99)
      OPEN (17,STATUS='SCRATCH',ACCESS='DIRECT',
     X IOSTAT=ios2,ERR=99,RECL=16)
      READ *,anum ! Enter number and begin reading file
      DO WHILE (ios1 .GE. 0) ! EOF
         READ (18,*,END=100,IOSTAT=ios1,ERR=99) fnum
         nrec = nrec + 1
         IF (anum .LE. fnum) THEN
           WRITE(17,REC=wrec) anum
           wrec = wrec + 1
   20      WRITE(17,REC=wrec) fnum
           wrec = wrec + 1
           READ (18,*,END=150,IOSTAT=ios1,ERR=99) fnum
           nrec = nrec + 1
           GOTO 20
         ELSE
           WRITE (17,REC=wrec) fnum
           wrec = wrec + 1
         END IF
      END DO
C  The file is empty or the item goes at the end of file
 100  WRITE (17,REC=wrec) anum
C  Copy the scratch file to the data file
 150  REWIND 18
      DO  i = 1,nrec
         READ (17,'(F16.6)',REC=i) fnum
         WRITE (18,*) fnum
      END DO
      CLOSE (18)
      CLOSE (17)
      STOP 'All Done'
C  Error handling section
 99   WRITE (6,'("ERROR = ",2I6)') ios1,ios2
      END
```

4.9 Environment Setting Variables

To be compatible with the VMS Fortran compiler, the Stardent 1500/3000 Fortran compiler writes its unformatted files in the same format the VMS compiler does. In both Stardent 1500/3000 format and VMS format defaults, all records are an even multiple of words (in 4-byte quantites) in length. If you request that a record be written which is not an even multiple of four bytes in length, both systems pad that record so that it is an even multiple of words in length. For variable length records, both Stardent 1500/3000 and VMS add extra 4-byte fields to the beginning and the end of each record to hold the size of the record. The value stored in these fields holds the record length in words. These fields are only for variable length records; both Stardent 1500/3000 and VMS assume that you know the size of fixed length records and need not to specify that record length in the file.

There are users who wish to write programs on the Stardent 1500/3000 that can read unformatted files that have been written on other machines—most commonly, machines that run on BSD UNIX. Because most UNIX compilers use a format that differs from the Stardent 1500/3000 format, this cannot be done without first converting the files to be read form BSD format to the Stardent 1500/3000 format. Most UNIX systems do not pad records to a 4-byte multiple, and most attach fields to both fixed and variable length records indicating the record size. Also, this record size is usually measured in bytes rather than words.

4.9.1 *UNFORMATTED_IO, UNFORMATTED_INPUT, UNFORMATTED_OUTPUT*

By using the IO library, the Stardent 1500/3000 system provides you with a mechanism for reading or writing unformatted files from most BSD UNIX systems. When programs are relinked with the IO library (there is no need to recompile), you can set three environment variables to control the format used to read or write unformatted files. At the time a program is run, the IO library reads three environment variables:

UNFORMATTED_IO
UNFORMATTED_INPUT
UNFORMATTED_OUTPUT

If none of these are set, or if they are set to the value VMS, the IO library uses its default format when reading or writing unformatted files.

If the variables are set to the value BSD, the IO library uses instead the format described above for BSD UNIX files.

If INPUT is set to BSD, read operations assume BSD format; if OUTPUT is set to BSD, write operations assume BSD format. Thus, by setting INPUT to BSD and leaving OUTPUT unset, it is very easy to write a program to convert a file from BSD format to Stardent 1500/3000 format. If the IO variable is set, its value overrides INPUT and OUTPUT, and files are both read and written in the specified format.

NOTE
For additional information on the environment setting variables, please refer to Chapter 10, *User Commands*, in the section called *FC(1)*.

EXAMPLE

With the IO library, if you want to use unformatted files generated on BSD systems, you need only type the command

setenv UNFORMATTED_INPUT BSD

or the command

setenv UNFORMATTED_OUTPUT BSD

or the command

setenv UNFORMATTED_IO BSD

depending on whether you just want to read files (if so, use the first command), or to write files (if so, use the second command), or to generate files to be read/written on a BSD system (if so, use the third command) before running your program.

The Stardent 1500/3000 can handle files generated on most BSD UNIX systems. However, note that while the Stardent 1500/3000 uses VMS style as its default for unformatted files, the Stardent 1500/3000 IO library cannot by default read unformatted files that have been written on a VMS system or write unformatted files that can be read on a VMS system. The VMS assumes an unusual format in its number representation that does not translate directly to most machines, including the Stardent 1500/3000.

PROCEDURES AND SUBPROGRAMS

CHAPTER FIVE

A subprogram can be a procedure or a subprogram. It is a self-contained computational or data unit that requires activation by the main program or another subprogram. Fortran subprograms perform special functions (such as solving a mathematical problem, performing a sort, outputting standard headings, and so on) and provide initial values for variables and array elements in named common blocks.

Block data subprograms initialize variables in labeled common blocks and cannot contain executable statements. A block data subprogram predefines, by means of **DATA** declarations, values for variables listed in **COMMON** declarations. Preassignment of data to common regions can be done only in a block data unit.

Procedures can be grouped into these two main categories:

- Subroutine subprograms.
- Functions:
  - Function subprograms.
  - Statement functions.
  - Intrinsic functions.

Subroutine and function subprograms can be written in languages other than Fortran and can come from the system library. For more information, refer to the *Programmer's Guide* in Chapter 10, *Language Interfacing*, and also in this manual in Appendix B under the section called *Language Calling Conventions*. Subroutines and functions differ in how they are referenced and how their values are returned.

5.1 Subroutine Subprograms

Subroutine subprograms are user-written procedures that perform a computational process or a subtask for another program unit. Values can be passed to the subroutine and returned to the calling program unit by using arguments or common blocks (specified by **COMMON** declarations, described later in this chapter). The subroutine subprogram is a program unit that has a **SUBROUTINE** statement (refer to Chapter 2, *Fortran Statements* for syntax) as its first statement.

A subroutine subprogram can contain any statement except another **SUBROUTINE** statement, a **BLOCK DATA**, **FUNCTION**, or **PROGRAM** statement. The last line of a subroutine subprogram must be an **END** statement. One or more **RETURN** statements may be included to return control to the calling program unit. If no **RETURN** statement is included in the subroutine, the subroutine **END** statement returns control to the calling program unit.

EXAMPLES

```
SUBROUTINE next(arg1,arg2)
SUBROUTINE last(a, *, *, b, i, k, *)
SUBROUTINE noarg
```

Values are passed to a subroutine subprogram by dummy arguments. Examples of dummy arguments as shown in the previous example are

```
arg1
arg2
a
b
* (alternate return form)
```

The alternate return form of dummy arguments is described later in this chapter under the title, *Alternate Returns from a Subroutine*.

5.1.1 Referencing a Subroutine

A subroutine subprogram is executed when a **CALL** statement (refer to Chapter 2, *Fortran Statements* for syntax) is encountered in a program unit.

EXAMPLES

```
CALL next (x, y)
CALL last (a, *10, *20, b, i, k, *30)
CALL noarg
```

When the subroutine is executed, its dummy arguments are replaced by references to the actual arguments found in the call to that subroutine subprogram. The above subroutine subprogram calls reference the actual parameters to the subprogram: **a, *10, *20, b, i, k,** and ***30**. Their values are used within the subroutine. (Refer to *Procedure Communication* later in this chapter.)

5.1.2 Alternate Returns from a Subroutine

Normally, control is returned from a subroutine to the calling program unit at the statement following the **CALL** statement. Specifying alternate return statements allows return to the calling program unit at any labeled executable statement. An alternate return is specified by the **RETURN** statement with an integer expression (which can be an integer constant) that identifies the number of a statement label in the **CALL** statement. The **SUBROUTINE** statement must contain one or more asterisks corresponding to alternate return labels in the **CALL** statement. (Refer to *CALL Statement* and *RETURN Statement* in Chapter 2, *Fortran Statements* for the syntax of calls with alternate return statements.)

EXAMPLE

Following is an example of a **CALL** and its associated **SUBROUTINE** and alternate return statements. Upon execution of a **RETURN** statement with an expression whose value is either less than 1 or greater than the number of alternate return labels in the **CALL** statement, control is returned to the statement following the **CALL** statement.

```
      CALL sub (a, *10, *20, *30)
C        ...
      SUBROUTINE sub (a, *, *, *)
C        ...
      RETURN n
```

Control returns to statement **10**, **20**, or **30**, depending on whether **n** evaluates to 1, 2, or 3.

5.2 Functions

A function can be intrinsic (refer to *Intrinsic Functions* later in this chapter) or defined in a user-written function subprogram. Execution of a function reference in an expression causes the evaluation of the specified function and a value to be defined for the function name. As with a subroutine, a function can return values through its arguments or common blocks.

When a function is evaluated, the function name is associated with a value in the same manner as a variable. Because this inherently types the function name, its type must be declared explicitly or implicitly, just as with other data names.

5.2.1 Function Subprograms

A function subprogram is a user-written Fortran function incorporated in a Fortran program. A function subprogram is a program unit that has a **FUNCTION** statement (refer to Chapter 2, *Fortran Statements* for syntax) as its first statement. A function subprogram can contain any statement except another **FUNCTION** statement, a **BLOCK DATA**, **SUBROUTINE**, or **PROGRAM** statement.

Because a value is assigned to the function subprogram name, it must have a type. The type associated with the function name is determined in one of three ways:

- If the type is mentioned as the first part of the **FUNCTION** statement, the name is assigned that type. If the type is specified in the **FUNCTION** statement, it must not appear in a type statement. A name must not have its type explicitly specified more than once in a program unit.

- If the type is not mentioned in the **FUNCTION** statement, the function name may be mentioned in a type statement within the function subprogram. (A type statement is the only nonexecutable statement in which a function name can

appear. Refer to *Type Statement* in Chapter 2, *Fortran Statements*.)

- If the function name is not mentioned in a type statement and the type is not included in the **FUNCTION** statement itself, the type is assigned implicitly according to the first letter of its name.

The type associated with the function name in each referencing program unit must agree with the type of the function as determined by the above methods.

A value must be assigned to the function subprogram name during the execution of the function subprogram. The name may be used as an ordinary variable. The value last assigned to the name of the function at the time a **RETURN** statement is executed within the subprogram is the value retained by the function name.

The last line of a function subprogram must be an **END** statement. One or more **RETURN** statements can be included to return control to the calling program unit. If no **RETURN** statement is included in the subroutine, the **END** statement returns control to the calling program unit. Alternate returns are not allowed in function subprograms. A function subprogram always returns to the expression from which it was invoked.

EXAMPLES

```
FUNCTION time()
INTEGER*4 FUNCTION add(k, j)
LOGICAL FUNCTION key_search(char_string, key)
```

Values are passed to function subprograms by arguments (**k** and **j**, **char_string** and **key** in the above examples). Refer to *Using the COMMON Statement* later in this chapter. Note that an argument list is not required.

```
      INTEGER FUNCTION fact(n)
      fact = 1
      DO 10 i = 2, n
        fact = fact*i
   10 CONTINUE
      RETURN
      END
```

The function name is associated with a value by appearing on the left side of an assignment statement. This **DO** loop may execute no times.

```
FUNCTION tot(num, sum)
REAL num
IF (num .GE. 0) THEN
  tot = sum + num
ELSE
  READ (5,*) tot
ENDIF
RETURN
END
```

The function name is associated with a value in one of two ways: by appearing on the left side of an assignment statement or by appearing in the input list of a **READ** statement.

```
FUNCTION next1(back)
IF (back .GT. 1.5) THEN
  CALL gtfwrd(next1)
ELSE
  CALL gtback(next1)
ENDIF
RETURN
END
```

The function name is associated with a value in one of two subroutine subprograms. Within the subroutines, **next1** must be assigned a value.

5.2.2 Statement Functions

A statement function is a user-defined, single-statement computation that applies only to the program unit that defines it. Its form is similar to that of an arithmetic, logical, or character assignment statement. Only one value is derived from a statement function. The statement function is referenced by using its symbolic name, with an actual argument list, in an arithmetic, logical, or character expression. A statement function can be referenced only in the program unit that contains it.

In a given program unit, all statement function definitions must precede the first executable statement of the program unit and must follow any specification statements in the program unit. The name of a statement function must not be a variable name or an array name in the same program unit.

The type of a statement function is determined in the same way as that of a variable. That is, it is either declared explicitly in a type statement or determined implicitly by the name. If the type of the statement function is not the same as that of the expression to the right of the assignment operator (the equals sign), the expression is converted to the proper type, following the rules in Table 2-2.

For example, in the following, **i** and **j** are integer expressions and **f** is real. Thus, the statement function is converted to real. The agreement rules are the same as those for assignment statements.

```
f(i) = i + j
```

All arguments in the dummy argument list are simple variables, and assume the value of the actual arguments in the same program unit when the function is invoked. These variables are completely distinct from variable, array, function, subroutine or common block names in the program unit or in other statement functions (that is, they are local to the statement function). Variables in the statement function and not included in the argument list assume the current value of the variable name in the program unit. For syntax and more information about statement functions, refer to *Statement Function Statement* in Chapter 2, *Fortran Statements*.

EXAMPLES

```
root(a,b,c) = (-b + SQRT(b*b - 4.*a*c))/(2.*a)
disp(c,r,h) = c*3.1416*r*r*h
indexq(a,j) = IFIX(a) + j - ic
```

5.2.3 Intrinsic Functions

An intrinsic function is one provided by Fortran and available to any program. Intrinsic functions perform such operations as value type conversions. Intrinsic functions also perform basic mathematical functions, such as finding sines, cosines, and square roots of numbers. Intrinsic functions available to Fortran are listed in Table 6-1 in Chapter 6, *Intrinsic Functions*. The table gives the definition of each function, the number of arguments, the generic name for each group of functions, the specific name for each function, the types of arguments allowed, and the argument and function type. Declaring an intrinsic function in a type statement has no effect on the type of the intrinsic function.

In addition, note that the Stardent 1500/3000 compiler has its own internal names for intrinsic functions that differ from those specified in the source program (e.g. the single precision **sin** function is called **_MA_SIN** in the internal library). Thus, if you port code which contain function names that are the same as some of the Stardent 1500/3000 non-Fortran 77 intrinsics (e.g. ATAND), you may have to make minor changes to the program.

EXAMPLE

```
INTRINSIC float
INTEGER float
x = float(y)
```

The type of float is **REAL**, not **INTEGER**.

5.2.3.1 Generic Names

Generic names simplify the referencing of intrinsic functions because the same name can be used with more than one type of argument.

The highest level of generic function name allows the use of one name for any precision integer or real variable argument. For example, in the function **ABS(*var*)**, *var* can be an integer, short integer, byte, real, double precision, complex, or double complex constant, variable, or expression.

A second, lower level of generic function name allows the use of one function name for any precision of a particular type of argument. For example, in the expression **IABS(*ivar*)**, *ivar* is an integer, short integer,or byte, constant, variable, or expression.

The type of the generic function result is determined by the type of its arguments. An **IMPLICIT** statement or type declaration does not change the type of an intrinsic function. If a generic or specific name appears as a dummy argument, then that name does not identify an intrinsic function in that program unit or statement function.

EXAMPLE

```
      SUBROUTINE x(log,f)
      EXTERNAL log
C      ...
      f = log(f)
C      ...
      END
```

In this context, **log** is not an intrinsic function.

5.2.4 Referencing a Function

A function is executed when its name appears in an expression.

SYNTAX

name ([*arg1*, *arg2*, ...])

where

name is the name of the function.

arg is an actual argument.

A function reference returns a specific value of the type associated with the function and is equivalent in usage to a variable reference of the same type. When a function reference is encountered during the evaluation of an expression, control is passed to the referenced function. The function is executed using the actual arguments listed in the function reference. The function name is assigned a value and is passed back to the referencing expression, which continues its evaluation, using the passed value where the function reference appeared.

The length of a character function in a character function reference must be the same as the length of the character function in the referenced function. A length of (*) matches all references.

EXAMPLES

```
a = z + root(a,b,c)
```

root is a user-defined function (defined in a function subprogram) that uses the values of **a**, **b**, and **c** to compute a value for **root**.

```
s = SIN(6.5)
```

SIN is an intrinsic function that is used here to compute the sine of 6.5.

```
b = stuff()
```

stuff is a user-defined function that has no arguments.

In the above examples, a real number is returned as the value of SIN(6.5).

5.3 Procedure Communication

Values are passed between a calling program and a procedure in an argument list. In addition, values can be passed through common blocks to and from subprograms.

5.3.1 Using Arguments

The arguments passed by the calling program are called *actual arguments*. The procedure, which is structured with dummy arguments, uses the actual arguments passed to it to replace the dummy arguments and perform the computation. For example, when the call is made to the function subprogram from the following calling program unit:

```
a = 6.5
b = 8.3
r = rfunc(a,b)*3.14159
```

a and **b** are passed to the subprogram. The subprogram could be the following:

```
FUNCTION rfunc(c,d)
rfunc = (c*d) + (d**3)
RETURN
END
```

Variables used as dummy arguments (like **c** and **d** in the above example) are said to be *passed by reference*. This means they refer to the storage locations of the actual arguments. Changing the values of the dummy arguments passed by reference changes the actual arguments in the calling program unit.

Actual arguments in a subroutine call or function reference should agree in number, order, and type with the corresponding dummy arguments. An actual argument must be a variable (simple or subscripted), array name, substring, procedure name, constant, or expression. The expression can be a character expression, except for one involving concatenation of an operand whose length specification is (*), unless that operand is the symbolic name of a constant. The actual arguments for statement functions are limited to variables, constants, and expressions. The argument of a subroutine subprogram can also be an alternate return specifier (described in *Alternate Returns from a Subroutine* earlier in this chapter).

When a procedure name is used as an actual argument, it does not pass a value as do other actual arguments. Instead, it passes the

actual subprogram name to the referenced subroutine or function. A subprogram name used as an actual argument must appear in an **EXTERNAL** statement. Intrinsic function names used as actual arguments must appear in an **INTRINSIC** statement. A dummy procedure argument that is never referenced should also appear in an **EXTERNAL** statement.

EXAMPLES

```
func(q,r,s) = q*r/s
```

dummy parameters within a statement function can only be simple variables.

```
FUNCTION nect (z, i, j)
DOUBLE PRECISION*8
DIMENSION j(10)
```

z is a simple variable of type real, **i** is a double precision variable, and **j** is a 10-element integer array.

```
SUBROUTINE add(q, f, get)

EXTERNAL get

q = get(f)
```

q and **f** are real variables and **get** is a function name.

Examples of Argument Correspondence

```
      CALL sub1(q, x, i, r(1), fcn)
C      ...
      END

      SUBROUTINE sub1(arry, r, in1, tmp,f)
      DIMENSION arry (20)
C      ...
      r = f(in1,tmp)
```

q is an array name and the dummy parameter array must be dimensioned in the subprogram. **x** is a real variable; the dummy parameter in the second position of the subroutine argument list (**r**) must also be a real variable. **i** corresponds to in1. **r(1)** is an element of array **r** and can correspond to a single variable name (not dimensioned) or an array (dimensioned) in the dummy argument list (**tmp**). **fcn** is a function name; **f**, therefore, must be used in the context of a function in the subprogram.

Examples of Array Passing

```
      PROGRAM main
      DIMENSION x(10,10)
C      ...
      CALL colz(x(1,5))
C      ...
      END
```

The main program unit dimensions an array **x** having 10 rows and 10 columns. One element of **x** appears in the argument list of the reference to **SUBROUTINE colz** (address of the element in the fifth column of the first row of **x**). **colz** dimensions the dummy argument **colx** to have 10 elements, thus corresponding to the entire fifth column of the actual array **x**. **colz** then sets each element of the fifth column of **x** to **0** and returns.

```
      SUBROUTINE colz(colx)
      DIMENSION colx(10)
C      ...
      DO 10 i = 1, 10
   10   colx(i)=0.0
      END
```

Here is an example of a character argument.

```
      FUNCTION size(string)
      CHARACTER*10 string
```

All variable names are local to the program unit that defined them (and, if it is a statement function, local to that statement), and, similarly, dummy arguments are local to the subprogram unit or statement function containing them. Thus, they can be the same as names appearing elsewhere in another program unit. No element of a dummy argument list can occur in a **COMMON** (except as a common block name), **EQUIVALENCE**, or **DATA** statement. When an array name is used as a dummy argument, the dummy argument array name must be dimensioned in a **DIMENSION** or type statement within the body of the subprogram.

If the actual argument is a constant, symbolic name of a constant, function reference, expression involving operators, or expression enclosed in parentheses, the associated dummy argument **must not** be redefined within the subprogram.

5.3.2 Using the COMMON Statement

A common block can pass values between a calling program unit and a subprogram. (Refer to *COMMON Statement* in Chapter 2, *Fortran Statements* for more information on using common blocks.) The example below shows how common blocks can pass values to and from subprograms.

EXAMPLE

```
PROGRAM hypotenuse
COMMON q, r, side
READ *, q, r
CALL tri
PRINT *, side
END

SUBROUTINE tri
COMMON x, y, side
side = SQRT(x**2 + y**2)
RETURN
END
```

The variable **q** in the main program shares storage space with **x** in the subroutine. When a value for **q** is determined by the **READ** statement, **x** automatically shares this value. Similarly, **r** and **y** also share storage space, as does the variable **side** in the main program and the subprogram. The subroutine uses the values input for **q** and **r** to compute the length of the hypotenuse of a right triangle.

5.3.3 Arrays in Subprograms

Because only the name of an array appears in the dummy argument list of a subprogram, an array declarator must appear in a **DIMENSION** or type statement for that array name. The number and size of dimensions of an actual argument array can differ from the number and size of the dimensions in the associated dummy argument array. The size of the dummy argument array must not exceed the size of the actual argument array. Because array bounds across separate compilation units are not checked at run-time, no warning is issued if the dummy array size exceeds the actual array size. Normally, array bounds are specified by integer constants and the bounds are fixed by the values of these constants. It is possible, however, to use *adjustable array declarators* in subprograms. In this case, one or more of the array bounds is specified by an expression.

The last upper bound of a dummy argument array is not used; Fortran allows this bound to be an asterisk. A declarator of this type is called an *assumed-sized array declarator*. Because the last bound of the last dimension is not specified, it can take any value. If the last bound exceeds that of the actual argument, the results are unpredictable.

A variable that dimensions a dummy argument in a bounds expression in a subprogram can appear either in a common block or as a dummy argument, but not both.

Adjustable and assumed-sized array declarators cannot be used in **COMMON** statements. If a dummy argument is of type character, the associated actual argument must be of type character and the length of the dummy argument must be less than or equal to the length of the actual argument. If the length of a dummy argument of type character is less than the length of an associated actual argument, then the characters associated with the dummy argument are the left most number of characters of the actual argument up to the length of the dummy argument. For example, if an actual character argument is a variable assigned the value **abcdefgh** and the length of the dummy argument is 4, then the characters **abcd** are associated with the dummy argument.

If a dummy argument of type character is an array name, the restriction on length is for the entire array and not for each array element. The length of an individual array element in the dummy argument array can be different from the length of an array element in an associated actual argument array, array element, or array element substring, but the dummy argument array must not extend beyond the end of the associated actual argument array.

If an actual argument is a character substring, the length of the actual argument is the length of the substring. If an actual argument is the concatenation of two or more operands, its length is the sum of the lengths of the operands.

The length of a dummy argument can be declared by an asterisk, as in this example:

```
SUBROUTINE sub(char_dummy)
CHARACTER*(*) char_dummy
```

In such a case, the dummy argument assumes the length of the associated actual argument for such reference of the subroutine or function. If the associated actual argument is an array or array element name, the length assumed by the dummy argument is the

length of an array element in the associated actual argument array.

EXAMPLE

The following example demonstrates the use of assumed-sized array declarators. The last subscript of the array **z** can take any value 1 to 10. The last upper bound of the array **arry** can take any value from 0 to 6.

```
      PROGRAM arry_declar
C      ...
      DIMENSION a(10,10)
      CALL sub 1(a)
C      ...
      END


      SUBROUTINE sub1(z)
      DIMENSION z(10,*)
C      ...
      INTEGER num(5,10,0:6)
      j = 5
      i = func1(num,j)
C      ...
      END
      FUNCTION func1(arry,k)
      INTEGER arry(k,10,0:*)
C
C  assign value to func1
C      ...
      END
```

5.3.4 Built-In Functions

Communicating with subprograms written in a language other than Fortran can be achieved with built-in functions. There are two kinds of built-in functions: argument list built-in functions and the %LOC built-in function.

5.3.4.1 Argument List Built-In Functions

Sometimes it is necessary to call subprograms written in languages other than Fortran, and the actual arguments for the call may need to be passed in a form that differs from the ones used by Fortran. The form of the argument can be changed by using the built-in functions %VAL, %REF, and %DESCR in the argument list of a function reference or **CALL** statement. These functions tell the subprogram how the argument is being passed.

The only context in which these built-in functions can be used is in an argument list of a **CALL** statement or function reference.

The function **%VAL** passes the argument as a 32 bit value. The function **%REF** passes the argument by reference, and the function **%DESCR** passes the argument by descriptor.

SYNTAX

```
%VAL(a)
%REF(a)
%DESCR(a)
```

where

a is an actual argument.

Use of the Fortran intrinsic function, **%DESCR**, causes the variable, *a*, to be passed to a subprogram or function as if the variable were a character variable whose length is determined by the data type of the variable itself.

In the program fragment below, note that a matching dummy argument must be declared as a character data type, whose length is compatible with the actual argument.

EXAMPLE

```
INTEGER I

CALL  THING(%DESCR(I))
   .
   .
   .
SUBROUTINE THING(C)

CHARACTER*(*)C
```

Within the subroutine the storage affected by operations on the dummy argument is in fact the storage allocated to the actual argument. This occurs even though the actual argument is not of type character.

5.3.4.2 %LOC Built-In Function

The %LOC built-in function computes the internal address of a storage element. This produces an INTEGER*4 value that can be used as an element in an arithmetic expression. The INTEGER*4 value that is produced represents the location of its argument.

SYNTAX

```
%LOC(arg)
```

where

arg is a variable, array element, character substring, array name reference, aggregate reference, or external procedure name.

5.4 ENTRY Statement

An **ENTRY** statement permits a procedure reference to begin with a particular executable statement within the function or subroutine subprogram in which it appears, and is a nonexecutable statement. It can appear anywhere within a function subprogram or a subroutine subprogram after, respectively, the **FUNCTION** or **SUBROUTINE** statement, with the exception that it cannot appear within an **IF** block or a **DO** loop.

A subprogram can have any number of **ENTRY** statements.

For the syntax of the **ENTRY** statement, refer to *ENTRY Statement* in Chapter 2, *Fortran Statements*.

5.4.1 Referencing an External Procedure by Entry Name

An entry name in an **ENTRY** statement in a function subprogram identifies an external function within the executable program and can be referenced as an external function. An entry name in an **ENTRY** statement in a subroutine subprogram identifies a subroutine within the executable program and can be referenced as a subroutine. When an entry name references a procedure, execution of the procedure begins with the first executable statement that follows the **ENTRY** statement with that entry name. An entry name can be used for reference by any program unit of an executable program.

The order, number, type, and names of the dummy arguments in an **ENTRY** statement can differ from those of the dummy arguments in the **FUNCTION** or **SUBROUTINE** statement and of other **ENTRY** statements in the same subprogram. However, each reference to a function or subroutine must use an actual argument list that agrees in order, number, and type with the dummy argument list in the corresponding **FUNCTION, SUBROUTINE,** or **ENTRY** statement. (Using a subroutine name or an alternate return specifier as an actual argument is an exception to the rule requiring agreement of type.)

5.4.2 Entry Association

Within a function subprogram, all variables whose names are also the names of entries are associated with each other and with the variable, if any, whose name is also the name of the function subprogram. Therefore, any such variable that becomes defined causes all associated variables of the same type also to become defined and all associated variables of different type to become undefined. Such variables need not be of the same type, but a variable that references the function must be defined at the time a **RETURN** or **END** statement is executed in the subprogram. An associated variable of a different type must not become defined during execution of the function reference.

5.4.3 ENTRY Statement Restrictions

Within a subprogram, an entry name cannot appear both as an entry name in an **ENTRY** statement and as a dummy argument in a **FUNCTION, SUBROUTINE,** or **ENTRY** statement. It cannot appear in an **EXTERNAL** statement.

In a function subprogram, a variable name that is the same as an entry name cannot appear in any statement that precedes the appearance of the entry name in an **ENTRY** statement (except in a type statement).

If an entry name in a function subprogram, or the subprogram name itself, is of type character, each entry name and the name of the function subprogram must also be of type character. If the name of the function subprogram or any entry in the subprogram is declared to be a character string of length (*), all such entities must also be declared to be of length (*). In all other cases, all such entities must have a length specification of the same integer value.

If a dummy argument appears in an executable statement, the execution of that statement is permitted during the execution of a reference to the function or subroutine only if the dummy argument appears in the dummy argument list of the procedure name referenced.

5.5 Block Data Subprograms

Block data subprograms define the size and reserve storage space for each labeled common block and, optionally, initialize the variables and arrays declared in the common block. It is recognized by its opening declaration, **BLOCK DATA** [*name*], and ends with an **END** statement.

The **BLOCK DATA** statement, described in Chapter 2, *Fortran Statements*, must be the first noncomment statement in a block data subprogram. Each named common block referenced in an executable Fortran program can be defined in a block data subprogram.

NOTE
Stardent 1500/3000 Fortran allows common blocks to be initialized in any program unit. This is an extension of Fortran 77.

In general, only specification statements and data initialization statements are allowed in the body of a block data subprogram. Allowable statements include **COMMON**, **DIMENSION**, type statements (**INTEGER*4**, **REAL*8**, and so on), **DATA**, **SAVE**, **PARAMETER**, and **IMPLICIT** statements. **EXTERNAL** and **INTRINSIC** statements are illegal. (Blank common variables can not be initialized in a block data subprogram.)

EXAMPLE

```
      BLOCK DATA George
      COMMON /xxx/ x(5),b(10),c
      COMMON /set1/ iy(10)
      DATA iy /1,2,4,8,16,32,64,128,256,512/
      DATA b/ 10*1.0/
C      ...
      END
```

George is the optional name of a block data subprogram to reserve storage locations for the named common blocks **xxx** and **set1**. Arrays **iy** and **b** are initialized in the **DATA** statements shown. The remaining elements in the common block can optionally be initialized or typed in the block data subprogram.

INTRINSIC FUNCTIONS

CHAPTER SIX

Stardent 1500/3000 Fortran supports a wide range of mathematic intrinsic functions. Intrinsic functions have symbolic names that are predefined by the compiler. They perform a variety of functions ranging from basic mathematical functions (i.e. sines, cosines, ...) to a complete set of logical conversion functions. In addition, they handle type conversions as well. For example, a type conversion can be something like a conversion from an integer to a real type or a real to an absolute data type.

6.1 Intrinsic Functions

Table 6-1 lists the intrinsic functions of Stardent 1500/3000 Fortran. The table gives the definition of each function, the number of arguments, the generic name for each group of functions, the specific name for each function, the types of arguments allowed, and the argument and function type.

In the following table these equalities are applicable:

Integer is **INTEGER*2** or **INTEGER*4**

Logical is **LOGICAL*2** or **LOGICAL*4**

Complex is **COMPLEX*8** or **COMPLEX*16**

Real is **REAL*4**

Double is **REAL*8**

NOTE
Because intrinsic functions are built into the language, the Fortran compiler maps the specified name into something it knows uniquely (e.g. the single precision **sin** function becomes **_MA_SIN**). These routines can be directly called from C, but they do not have exactly the same name as the Fortran intrinsic functions.

Table 6-1. Intrinsic Functions

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Type Conversion	Conversion to integer INT(a) [See Note **]	1	INT	INT	Integer	Integer
					Real	Integer
					Complex	Integer
			IFIX	IIFIX*	REAL*4	INTEGER*2
				JIFIX*	REAL*4	INTEGER*4
	Conversion to real [See Note †]	1	FLOAT	FLOATI*	INTEGER*2	REAL*4
				FLOATJ*	INTEGER*4	REAL*4
				REAL	SNGL	Double
					Real	COMPLEX*8
			DREAL*	COMPLEX*16	Real	
	Conversion to double precision [See Note ‡]	1	DBLE	DBLE	INTEGER*2	Double
					Real	Double
					Double	Double
					COMPLEX*8	Double
					COMPLEX*16	Double
			DFLOAT	DFLOTI*	INTEGER*2	Double
				DFLOTJ*	INTEGER*4	Double

* An asterisk (*) following the name of an intrinsic function indicates that this function is an extension to Fortran 77.

** For *a* of type integer or short integer, **int**(*a*) = *a*. For *a* of type real, or double precision, there are two cases:

if |*a*| <1, **int**(*a*) = 0; or

if |*a*| >1, **int**(*a*) is the integer whose magnitude is the magnitude of *a* and whose sign is the same as the sign of *a*. For example:

```
int(-3.7) = -3
```

For *a* of type complex, **int**(*a*) is the value obtained by applying the above rule to the real part of *a*. For *a* of type real, **IFIX**(*a*) is the same as **INT**(*a*).

† For *a* of type real, **Real**(*a*) is *a*. For *a* of type integer or double precision, **Real**(*a*) is as much precision of the significant part of *a* as a real datum can contain. For *a* of type complex, **Real**(*a*) is the real part of *a*.

For *a* of type integer, **FLOAT**(*a*) is the same as **Real**(*a*).

‡ For *a* of type double precision, **DBLE**(*a*) is *a*. For *a* of type integer or real, **DBLE**(*a*) is as much precision of the significant part of *a* as a double precision datum can contain. For *a* of type complex, **DBLE**(*a*) is as much precision of the significant part of the real part of *a* as a double precision datum can contain. For *a* of type double complex, **DBLE**(*a*) is the real part of *a*.

Table 6-1. Intrinsic Functions (continued)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Type Conversion (cont.)	Conversion to COMPLEX*8 [See Note §]	1,2	CMPLX		[See Note §]	COMPLEX*8
	Conversion to COMPLEX*16 [See Note §]	1,2	DCMPLX*		[See Note §]	COMPLEX*16
	Conversion to integer [See Note §§]	1		ICHAR	CHARACTER	INTEGER*4
	Conversion to CHARACTER	1		CHAR	Int./LOGICAL*1	CHARACTER

* An asterisk (*) following the name of an intrinsic function indicates that this function is an extension to Fortran 77.

§ **CMPLX** can have one or two arguments. If there is one argument, it can be of type integer, real, double precision, or complex. If there are two arguments, they must both be of the same type and can be of type integer, real, or double precision.

For *a* of type complex, **CMPLX(*a*)** is *a*. For *a* of type integer, real, or double precision, **CMPLX(*a*)** is the complex value whose real part is **Real(*a*)** and whose imaginary part is zero. For *a* of type double complex, **CMPLX(*a*)** is

```
CMPLX (Real (a), Real(AIMAG(a)))
```

CMPLX(*a*,*b*) is the complex value whose real part is **Real(*a*)** and whose imaginary part is **Real(*b*)**.

These rules also apply to **DCMPLX**. For *a* of type complex, **DCMPLX(*a*)** is

```
DCMPLX (DBLE (a), DBLE(DIMAG(a)))
```

§§ **ICHAR** converts from a character to an integer, based on the internal representation of the character. Characters in the ASCII character set have the standard ASCII values.

The value of **ICHAR(*a*)** is an integer in the range $0 \le |\textbf{ICHAR}(a)| \le 255$, where *a* is an argument of type character and length 1.

Table 6-1. Intrinsic Functions (continued)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Truncation	INT(a) [See note **]	1	INT	IINT*	REAL*4	INTEGER*2
				JINT*	REAL*4	INTEGER*4
				IIDINT*	REAL*8	INTEGER*2
				JIDINT*	REAL*8	INTEGER*4
			IDINT	IIDINT*	REAL*8	INTEGER*2
				JIDINT*	REAL*8	INTEGER*4
			AINT	AINT	Real	Real
				DINT	Double	Double
Nearest Whole Number	INT(a+.5) if a≥0 INT(a-.5) if a<0	1	ANINT	ANINT	Real	Real
				DNINT	Double	Double
Nearest Integer	INT(a+.5) if a≥0 INT(a-.5) if a<0	1	NINT	ININT*	REAL*4	INTEGER*2
				JNINT*	REAL*4	INTEGER*4
				IIDNNT*	Double	INTEGER*2
				JIDNNT*	Double	INTEGER*4
			IDNINT	IIDNNT*	REAL*8	INTEGER*2
				JIDNNT*	REAL*8	INTEGER*4

* An asterisk (*) following the name of an intrinsic function indicates that this function is an extension to Fortran 77.

** For *a* of type integer or short integer, **int(*a*)** = *a*. For *a* of type real, or double precision, there are two cases:

if $|a|<1$, **int(*a*)** = 0; or

if $|a|>1$, **int(*a*)** is the integer whose magnitude is the magnitude of *a* and whose sign is the same as the sign of *a*. For example:

```
int(-3.7) = -3
```

For *a* of type complex, **int(*a*)** is the value obtained by applying the above rule to the real part of *a*. For *a* of type real, **IFIX(*a*)** is the same as **INT(*a*)**.

Table 6-1. Intrinsic Functions (continued)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Absolute Value	|a| [See Note †]	1	ABS	IIABS*	INTEGER*2	INTEGER*2
				JIABS*	INTEGER*4	INTEGER*4
				ABS	Real	Real
				DABS	Double	Double
				CABS	COMPLEX*8	Real
				CDABS*	COMPLEX*16	Double
			IABS	IIABS*	INTEGER*2	INTEGER*2
				JIABS*	INTEGER*4	INTEGER*4
Remainder	a-INT(a/b)*b [See Note **]	2	MOD	IMOD*	INTEGER*2	INTEGER*2
				JMOD*	INTEGER*4	INTEGER*4
				AMOD	Real	Real
				DMOD	Double	Double
Transfer of Sign	|a| *if* b≥0 -|a| *if* b<0	2	SIGN	IISIGN*	INTEGER*2	INTEGER*2
				JISIGN*	INTEGER*4	INTEGER*4
				SIGN	Real	Real
				DSIGN	Double	Double
			ISIGN	IISIGN*	INTEGER*2	INTEGER*2
				JISIGN*	INTEGER*4	INTEGER*4

* An asterisk (*) following the name of an intrinsic function indicates that this function is an extension to Fortran 77.

** For *a* of type integer or short integer, **int**(*a*) = *a*. For *a* of type real, or double precision, there are two cases:

if |*a*|<1, **int**(*a*) = 0; or

if |*a*|>1, **int**(*a*) is the integer whose magnitude is the magnitude of *a* and whose sign is the same as the sign of *a*. For example:

```
int(-3.7) = -3
```

For *a* of type complex, **int**(*a*) is the value obtained by applying the above rule to the real part of *a*. For *a* of type real, **IFIX**(*a*) is the same as **INT**(*a*).

† A complex value is expressed as an ordered pair of reals or doubles, (*ar*,*ai*), where *ar* is the real part and *ai* is the imaginary part. **ABS** or **CABS** is defined as:

SQRT (ar^2+ai^2)

Table 6-1. Intrinsic Functions (continued)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Positive Difference	a-b *if* a > b 0 *if* a ≤ b	2	DIM	IIDIM*	INTEGER*2	INTEGER*2
				JIDIM*	INTEGER*4	INTEGER*4
				DIM	Real	Real
				DDIM	Double	Double
			IDIM	IIDIM*	INTEGER*2	INTEGER*2
				JIDIM*	INTEGER*4	INTEGER*4
Double Precision Product	a × b	2		DPROD	Real	Double
Choosing Largest Value	max(a,b,...)	≥2	MAX	IMAX0*	INTEGER*2	INTEGER*2
				JMAX0*	INTEGER*4	INTEGER*4
				AMAX1	Real	Real
				DMAX1	Double	Double
			MAX0	IMAX0*	INTEGER*2	INTEGER*2
				JMAX0*	INTEGER*4	INTEGER*4
			AMAX0	AIMAX0*	INTEGER*2	REAL*4
				AJMAX0*	INTEGER*4	REAL*4
			MAX1	IMAX1*	REAL*4	INTEGER*2
				JMAX1*	REAL*4	INTEGER*4
Choosing Smallest Value	min(a,b,...)	≥2	MIN	IMIN0*	INTEGER*2	INTEGER*2
				JMIN0*	INTEGER*4	INTEGER*4
				AMIN1	Real	Real
				DMIN1	Double	Double
			MIN0	IMIN0*	INTEGER*2	INTEGER*2
				JMIN0*	INTEGER*4	INTEGER*4
			AMIN0	AIMIN0*	INTEGER*2	REAL*4
				AJMIN0*	INTEGER*4	REAL*4
			MIN1	IMIN1*	REAL*4	INTEGER*2
				JMIN1*	REAL*4	INTEGER*4

* An asterisk (*) following the name of an intrinsic function indicates that this function is an extension to Fortran 77.

Table 6-1. Intrinsic Functions (continued)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Length	Length of character entity	1		LEN	CHARACTER	INTEGER*4
Index of a substring	Location of substring b in string a [See Note **]	2		INDEX	CHARACTER	INTEGER*4
Imaginary Part of a Complex Argument	ai [See Note †]	1		AIMAG DIMAG*	COMPLEX*8 COMPLEX*16	REAL*4 Double
Conjugate of a Complex Argument	(ar,-ai) [See Note ‡]	1	CONJG	CONJG DCONJG*	COMPLEX*8 COMPLEX*16	COMPLEX*8 COMPLEX*16
Square Root	$\sqrt{}$a	1	SQRT	SQRT DSQRT CSQRT CDSQRT*	Real Double COMPLEX*8 COMPLEX*16	Real Double COMPLEX*8 COMPLEX*16
Exponential	e^a	1	EXP	EXP DEXP CEXP CDEXP*	Real Double COMPLEX*8 COMPLEX*16	Real Double COMPLEX*8 COMPLEX*16

* An asterisk (*) following the name of an intrinsic function indicates that this function is an extension to Fortran 77.

** **INDEX**(*a*,*b*) returns an integer value representing the starting position within character string *a* of a substring identical to string *b*. If *b* occurs more than once within *a*, **INDEX**(*a*,*b*) returns the starting position of the first occurrence.

If *b* does not occur in *a*, the value **0** is returned. If **LEN**(*a*)<**LEN**(*b*), **0** is also returned.

† A complex value is expressed as an ordered pair of reals or doubles, (*ar*,*ai*), where *ar* is the real part and *ai* is the imaginary part. **ABS** or **CABS** is defined as:

$$\text{SQRT}\ (ar^2+ai^2)$$

‡ **CONJG** is defined as (*ar*,-*ai*); refer to note †.

Table 6-1. Intrinsic Functions (continued)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Natural Logarithm	ln(a) [See Note §]	1	LOG	ALOG DLOG CLOG CDLOG*	Real Double COMPLEX*8 COMPLEX*16	Real Double COMPLEX*8 COMPLEX*16
Common Logarithm	log(a)	1	LOG10	ALOG10 DLOG10	Real Double	Real Double
Sine	sin(a)	1	SIN	SIN DSIN CSIN CDSIN*	Real Double COMPLEX*8 COMPLEX*16	Real Double COMPLEX*8 COMPLEX*16
Sine (degrees)	sin(a)	1	SIND	SIND* DSIND*	Real Double	Real Double
Cosine	cos(a)	1	COS	COS DCOS CCOS CDCOS*	Real Double COMPLEX*8 COMPLEX*16	Real Double COMPLEX*8 COMPLEX*16
Cosine (degrees)	cos(a)	1	COSD	COSD* DCOSD*	Real Double	Real Double
Tangent	tan(a)	1	TAN	TAN DTAN	Real Double	Real Double
Tangent (degrees)	tan(a)	1	TAND	TAND* DTAND*	Real Double	Real Double

* An asterisk (*) following the name of an intrinsic function indicates that this function is an extension to Fortran 77.

§ All angles in trigonometric functions are expressed in radians.

Table 6-1. Intrinsic Functions (continued)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Arcsine	arcsin(a)	1	ASIN	ASIN DASIN	Real Double	Real Double
Arcsine (degrees)	arcsin(a)	1	ASIND	ASIND* DASIND*	Real Double	Real Double
Arccosine	arccos(a)	1	ACOS	ACOS DACOS	Real Double	Real Double
Arccosine (degrees)	arccos(a)	1	ACOSD	ACOSD* DACOSD*	Real Double	Real Double
Arctangent	arctan(a)	1	ATAN	ATAN DATAN	Real Double	Real Double
Arctangent (degrees)	arctan(a)	1	ATAND	ATAND* DATAND*	Real Double	Real Double
Arctangent	arctan(a/b)	2	ATAN2	ATAN2 DATAN2	Real Double	Real Double
Arctangent (degrees)	arctan(a/b)	2	ATAN2D	ATAN2D* DATAN2D*	Real Double	Real Double
Hyperbolic Sine	sinh(a)	1	SINH	SINH DSINH	Real Double	Real Double
Hyperbolic Cosine	cosh(a)	1	COSH	COSH DCOSH	Real Double	Real Double
Hyperbolic Tangent	tanh(a)	1	TANH	TANH DTANH	Real Double	Real Double

* An asterisk (*) following the name of an intrinsic function indicates that this function is an extension to Fortran 77.

Table 6-1. Intrinsic Functions (continued)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Lexically Greater Than or Equal To	a ≥ b [See Note **]	2		LGE	CHARACTER	LOGICAL*4
Lexically Greater Than	a > b [See Note **]	2		LGT	CHARACTER	LOGICAL*4
Lexically Less Than or Equal To	a ≤ b [See Note **]	2		LLE	CHARACTER	LOGICAL*4
Lexically Less Than	a < b [See Note **]	2		LLT	CHARACTER	LOGICAL*4
Bitwise Logical Shift	[See Note †]	2	ISHFT	IISHFT* JISHFT*	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
Bitwise Circular Shift	[See Note ‡]	3	ISHFTC	IISHFTC* JISHFTC*	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4

* An asterisk (*) following the name of an intrinsic function indicates that this function is an extension to Fortran 77.

** **LGE**(*a,b*) returns the value *true* if *a*=*b* or if *a* follows *b* in ASCII collating sequence; otherwise it returns the value *false*.

LGT(*a,b*) returns the value *true* if *a* follows *b* in ASCII collating sequence; otherwise it returns the value *false*.

LLE(*a,b*) returns the value *true* if *a*=*b* or if *a* precedes *b* in ASCII collating sequence; otherwise it returns the value *false*.

LLT(*a,b*) returns the value *true* if *a* precedes *b* in ASCII collating sequence; otherwise it returns the value *false*.

If the operands for **LGE**, **LGT**, **LLE**, and **LLT** are of unequal length, the shorter operand is treated as if padded on the right with blanks to the length of the longer operand.

† **ISHFT**(*a,b*) is defined as the value of the first argument (*a*) shifted by the number of bit positions designated by the second argument (*b*). If *b*>0, shift left; if *b*<0, shift right; if *b*=0, don't shift. Bits shifted out from the left or right end are lost, and zeros are shifted in from the opposite end. The type of the result is the same as the type of *a*.

‡ As a MIL-STD-1753 standard extension to Fortran 77, **ISHFTC**(*a,b,c*) is defined as the rightmost *c* bits of the argument *a* shifted circularly *b* places. That is, the bits shifted out of one end are shifted into the opposite end. No bits are lost. The unshifted bits of the result are the same as the unshifted bits of the argument *a*. The absolute value of the argument *b* must be less than or equal to *c*. The argument *c* must be greater than or equal to 1 and less than or equal to 16 if *a* is **INTEGER*2**, or 32 if *a* is **INTEGER*4**.

Table 6-1. Intrinsic Functions (continued)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Bit Extraction	[See notes **, †, and ‡]	3	IBITS	IIBITS* JIBITS*	INTEGER*2 INTEGER*4	INTEGER*2 INTEGER*4
Bit Test	[See notes **, †, ‡, and §]	2	BTEST	BITEST* BJTEST*	INTEGER*2 INTEGER*4	LOGICAL*2 LOGICAL*4

* An asterisk (*) following the name of an intrinsic function indicates that this function is an extension to Fortran 77.

** Functions **IBITS, BTEST, IBSET, IBCLR,** and **MVBITS** are defined by the MIL-STD-1753 definition, in which bit positions are numbered from right to left, with the rightmost (least significant) bit numbered 0.

As a MIL-STD-1753 standard extension to Fortran 77, the bit move subroutine, **CALL MVBITS**(*a,b,c,d,e*), moves *c* bits from positions *b* through *b+c-1* of arguments *a* to positions *e* through *e+c-1* of argument *d*. That is, *c* bits are moved from bit *b* of *a* to bit *e* of *d*. The portion of argument *d* not affected by the movement of bits remains unchanged.

All arguments are integer expressions, except *d*, which must be a variable or array element. Arguments *a* and *d* are permitted to be the same numeric storage unit. The values of *b+c* and *e+c* must be less than or equal to the lengths of *a* and *d*, respectively. Both *a* and *d* must be **INTEGER*4**.

The bit move subroutine, **CALL MVBITS2**(*a,b,c,d,e*), is also available. **MVBITS2** is semantically the same as **MVBITS** except that *a* and *d* are **INTEGER*2**

† As a MIL-STD-1753 standard extension to Fortran 77, bit subfields can be extracted from a field. Bit subfields are referenced by specifying a bit position and a length. Bit positions within a numeric storage unit are numbered from right to left and the rightmost bit position is numbered 0. Bit fields cannot extend from one numeric storage unit into another numeric storage unit, and the length of a field must be greater than 0.

The function **IBITS**(*a,b,c*) extracts a subfield of *c* bits in length from *a* starting with bit position *b* and extending left *c* bits. The result field is right-justified and the remaining bits set to 0. The value of *b+c* must be less than or equal to 16 if *a* is **INTEGER*2**, or 32 if *a* is **INTEGER*4**.

‡ As a MIL-STD-1753 standard extension to Fortran 77, individual bits of a numeric storage unit can be tested and changed with the bit processing routines described in Notes 18, and 19. Each function has two arguments, *a* and *b*, which are integer expressions. The value *a* specifies the binary pattern, and *b* specifies the bit position (rightmost bit is bit 0).

§ The function **BTEST** is a logical function. The *b*th bit of argument *a* is tested. If it is 1, the value of the function is *true*; if it is 0, the value is *false*. If *b*>16 for **INTEGER*2** or *b*>32 for **INTEGER*4**, the result is *false*.

Table 6-1. Intrinsic Functions (continued)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Bit Set	[See notes **, †, ‡, and §§]	2	IBSET	IIBSET*	INTEGER*2	INTEGER*2
				JIBSET*	INTEGER*4	INTEGER*4
Bit Clear	[See notes **, †, ‡, and ††]	2	IBCLR	IIBCLR*	INTEGER*2	INTEGER*2
				JIBCLR*	INTEGER*4	INTEGER*4

* An asterisk (*) following the name of an intrinsic function indicates that this function is an extension to Fortran 77.

** Functions **IBITS, BTEST, IBSET, IBCLR,** and **MVBITS** are defined by the MIL-STD-1753 definition, in which bit positions are numbered from right to left, with the rightmost (least significant) bit numbered 0.

As a MIL-STD-1753 standard extension to Fortran 77, the bit move subroutine, **CALL MVBITS**(*a,b,c,d,e*), moves *c* bits from positions *b* through *b+c-1* of arguments *a* to positions *e* through *e+c-1* of argument *d*. That is, *c* bits are moved from bit *b* of *a* to bit *e* of *d*. The portion of argument *d* not affected by the movement of bits remains unchanged.

All arguments are integer expressions, except *d*, which must be a variable or array element. Arguments *a* and *d* are permitted to be the same numeric storage unit. The values of *b+c* and *e+c* must be less than or equal to the lengths of *a* and *d*, respectively. Both *a* and *d* must be **INTEGER*4**.

The bit move subroutine, **CALL MVBITS2**(*a,b,c,d,e*), is also available. **MVBITS2** is semantically the same as **MVBITS** except that *a* and *d* are **INTEGER*2**

† As a MIL-STD-1753 standard extension to Fortran 77, bit subfields can be extracted from a field. Bit subfields are referenced by specifying a bit position and a length. Bit positions within a numeric storage unit are numbered from right to left and the rightmost bit position is numbered 0. Bit fields cannot extend from one numeric storage unit into another numeric storage unit, and the length of a field must be greater than 0.

The function **IBITS**(*a,b,c*) extracts a subfield of *c* bits in length from *a* starting with bit position *b* and extending left *c* bits. The result field is right-justified and the remaining bits set to 0. The value of *b+c* must be less than or equal to 16 if *a* is **INTEGER*2**, or 32 if *a* is **INTEGER*4**.

‡ As a MIL-STD-1753 standard extension to Fortran 77, individual bits of a numeric storage unit can be tested and changed with the bit processing routines described in Notes 18, and 19. Each function has two arguments, *a* and *b*, which are integer expressions. The value *a* specifies the binary pattern, and *b* specifies the bit position (rightmost bit is bit 0).

†† The result of the function **IBCLR**(*a, b*) is equal to the value of *a* with the *b*th bit set to 0. If *b*>16 for **INTEGER*2** or *b*>32 for **INTEGER*4**, the result is *a*.

§§ The result of the function **IBSET**(*a,b*) is equal to the value of *a* with the *b*th bit set to 1. If *b*>16 for **INTEGER*2** or *b*>32 for **INTEGER*4**, the result is *a*.

Table 6-1. Intrinsic Functions (continued)

Intrinsic Function	Definition	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Result
Bitwise AND	a b	2	IAND	IIAND*	INTEGER*2	INTEGER*2
				JIAND*	INTEGER*4	INTEGER*4
Bitwise OR	a b	2	IOR	IIOR*	INTEGER*2	INTEGER*2
				JIOR*	INTEGER*4	INTEGER*4
Bitwise Exclusive OR	[See Note **]	2	IEOR	IIEOR*	INTEGER*2	INTEGER*2
				JIEOR*	INTEGER*4	INTEGER*4
Bitwise Complement	¬ a	1	NOT	INOT*	INTEGER*2	INTEGER*2
				JNOT*	INTEGER*4	INTEGER*4
Zero		1	ZEXT	IZEXT*	INTEGER*2	INTEGER*2
					LOGICAL*2	INTEGER*2
					LOGICAL*1	INTEGER*2
				JZEXT*	INTEGER*4	INTEGER*4
					INTEGER*2	INTEGER*4
					LOGICAL*4	INTEGER*4
					LOGICAL*2	INTEGER*4
					LOGICAL*1	INTEGER*4

* An asterisk (*) following the name of an intrinsic function indicates that this function is an extension to Fortran 77.

** **IEOR** is defined as the bitwise modulo-2 sum (exclusive or) of the two arguments. **REAL*8** value.

6.1.1 General Type Rules for Intrinsic Functions

If the argument type matches the function type as specified in Table 6-1, then the function maintains the same type.

For example, if *a* is a **REAL*8, EXP(*a*)** is a **REAL*8** value.

If the argument type does not match the function type, the compiler default type determines the function result type (that is, **OPTION SHORT Integer** or **OPTION LONG Integer** compiler options).

For example, if **OPTION SHORT Integer** (short integer) is selected, **INT(XNUM)** results in a short integer value. Also, if *k* is of type short integer, then the generic function **ABS(*k*)** results in a short integer value.

To avoid ambiguity of interpretation, only when **OPTION SHORT integers** is specified, can the MIL-STD-1753 extension routines operate on INTEGER*2 arguments or produce INTEGER*2 results. Otherwise, regardless of the declared or actual size of the arguments, they are treated as INTEGER*4. When **OPTION SHORT Integer** is specified, all arguments must be INTEGER*2 for an INTEGER*2 result.

6.2 Additional Library Subroutines

Stardent 1500/3000 Fortran provides eight subroutines that can be called in the same manner as a user-written subroutine. Integer arguments in these subroutines can be either INTEGER*2 or INTEGER*4, but must be the same integer declaration within each subroutine.

6.2.1 DATE Subroutine

This subroutine returns the current date. This is in the form of day-month-year.

SYNTAX

CALL DATE(*arr*)

where

arr is a 9 byte long character variable. The date is returned in a 9 byte ASCII string in the form of:

dd-mmm-yy
where

dd is a two digit day date.

mmm is a three letter month.

yy is the last two digits of the year.

6.2.2 IDATE Subroutine

This subroutine returns the current date. This is in the form of three integers representing the month, day and year.

SYNTAX

CALL IDATE(*i, j, k*)

where

i is a numeric representation of the month.

j is a numeric representation of the day.

k is the last two digits of the year.

6.2.3 EXIT Subroutine

This subroutine causes the termination of a program. This also closes all files and returns control to the operating system.

SYNTAX

CALL EXIT[(*status*)]

where

status is a integer used to specify an exit value.

6.2.4 SECNDS Function

This function subprogram returns the system time in seconds. The value returned is expressed as a single precision, floating point number.

SYNTAX

y = SECNDS(x)

where

y is the returned value of the subroutine equal to the time in seconds since midnight, minus the value of x.

x is a user supplied single precision, floating point number, to be subtracted from y.

This intrinsic function reports times in 1/100th of a second units. A floating point value cannot hold as many 1/100th seconds as there are in a day completely accurately. The difference between times that are very far apart is accurate to about 1.5/100 of a second.

6.2.5 TIME Subroutine

This subroutine returns the current system time as an ASCII string.

SYNTAX

CALL TIME(*arr*)

where

arr is an 8 byte long character variable. The time is returned in an 8 byte ASCII string in the form of:

hh:mm:ss where

hh is a two digit hour indicator.

mm is a two digit minute indicator.

ss is the two digits second indicator.

6.2.6 RAN Function

This function is a random number generator of the multiplicative congruential type. A floating point number results from this functions which is in the range of 0.0 inclusive and 1.0 exclusive.

SYNTAX

y = **RAN**(*i*)

where

y is the returned value that is set equal to the value associated with the *i*.

i is an INTEGER*4 or INTEGER*4 array element.

The value of *i* should initially be set to a large, odd integer value, and then re-initialized with a different value on another run of the function. The value of the seed passed as the parameter is updated by **RAN** using the following algorithm:

SEED = 69069 * SEED + 1 (MOD 2**32)

where **SEED** is a 32 bit number whose high order 24 bits are converted to floating point and returned as the result.

RAN is an intrinsic function.

6.2.7 MVBITS and MVBITS2 Subroutines

The subroutine **MVBITS** is defined by the MIL-STD-1753 definition, in which bit positions are numbered from right to left, with the rightmost (least significant) bit numbered 0.

As a MIL-STD-1753 standard extension to Fortran 77, the bit move subroutine, **CALL MVBITS**(*a,b,c,d,e*), moves *c* bits from positions *b* through *b+c-1* of arguments *a* to positions *e* through *e+c-1* of argument *d*. That is, *c* bits are moved from bit *b* of *a* to bit *e* of *d*. The portion of argument *d* not affected by the movement of bits remains unchanged.

All arguments are integer expressions, except *d*, which must be a variable or array element. Arguments *a* and *d* are permitted to be the same numeric storage unit. The values of *b+c* and *e+c* must be less than or equal to the lengths of *a* and *d*, respectively. Both *a* and *d* must be **INTEGER*4**.

The bit move subroutine, **CALL MVBITS2**(*a,b,c,d,e*), is also available. **MVBITS2** is semantically the same as **MVBITS** except that *a* and *d* are **INTEGER*2**.

COMPILER OPTIONS

CHAPTER SEVEN

This chapter describes the Stardent 1500/3000 Fortran compiler options. These options include:

- Standard compilation options — section 7.2
- Input/Output options — section 7.3
- Debugging options — section 7.4
- Optimization options — section 7.5
- Porting options — section 7.6
- Miscellaneous options — section 7.7

7.1 Options List

All compilation options supported by the Stardent 1500/3000 Fortran are categorized into three groups: command line preprocessor options, command line loader options, and command line compiler options. The following tables briefly list and describe all options. Refer to specific sections thereafter for additional details and explanations.

Table 7-1. Command Line Preprocessor Options for Fortran*

Option	Negative Form	Description
-D*name*		Define *name* to have a value of 1
-D*name=val*		Define *name* to have a value of *val*
-E		Output expanded source
-I		Do not search for included files in default directory
-I*dir*		Search for included files in *dir*
-i		Suppress production of #ident information
-P		Preprocess only and copy to *file.i*
-U*name*		Undefine *name*

* These options are in effect only when the **-cpp** is specified.

Table 7-2. Command Line Loader Options for Fortran

Option	Negative Form	Description
-B *hhhhhhhh*		*a.out* file has bus address at *hhhhhhhh*
-D *hhhhhhhh*		*a.out* file has data address at *hhhhhhhh*
-e*sym*		Set default entry point address of *sym*
-L		Do not search for libraries in /lib or /usr/lib
-L*dir*		Search for libraries in *dir*
-l*tag*		Search library called *tag.a*. The default is first search in */lib* and then in */usr/lib*
-m		Generate a simple load map
-n		Generate NMAGIC file type
-opct		Produce count of FPU ops
-p	-noprofile	Generate profiling code
-r		Produce a relocatable output file
-s		Strip line numbers and symbol table information
-T *hhhhhhhh*		*a.out* file has text address at *hhhhhhhh*
-t		Turn off certain warnings
-y*name*		Trace uses and definitions in *name*

Table 7-3. Command Line Compiler Options for Fortran

Option	Negative Form	Description
-all_doubles		Use double precision for all undeclared, declared, and constant floating point variables. If they were complex, use double complex. Allocate 8 bytes of storage for each integer variable, rather than 4 bytes
-ast=*number*		Set the initial allocation for the program internal representation to the user defined *number* bytes. This option should not be used unless the program is terminated with an internal error indicating that you should invoke this option. The error message indicates the proper value to use for *number*
-blanks72		Pad source file lines with blanks
-c		Compile only; do not link
-case_sensitive	-nocase_sensitive	Allow user defined variables to be case sensitive. The default is -nocase_sensitive which causes all variables to be mapped to uppercase
-catalog=*name*.in		Create a database of functions to be inlined
-continuations=n		Specify acceptable number of continuation lines
-cpp		Invoke the C preprocessor
-cross_reference	-nocross_reference	Generate a cross-reference listing in listing file
-d_lines	-nod_lines	Compile debug lines that start with a **D** or **d** in column 1
-debug (-g)	-nodebug	Add debug data to object file. Force optimization level to zero. The default is -nodebug
-double_precision	-nodouble_precision	Use double precision for all undeclared, and declared floating point variables (REAL, not REAL*4). If they were complex, use double complex
-extend_source		Expand the statement field of a source line from columns 1 through 72 to columns 1 through 132
-fast		Perform some optimization which may lose small amounts of precision
-full_report		Invoke vector reporting facility
-fullsubcheck		Add code to check each linear array subscript
-g (-debug)	-nodebug	Add debug data to object file. Force optimization level to zero

Table 7-3. Command lIne Compiler Options for Fortran (continued)

Option	Negative Form	Descriptions
-implicit	-noimplicit	Untype all variables
-include_listing	-noinclude_listing	Add included files to the listing file
-inline		Enable automatic compiler inlining capability
-i4	-noi4	Interpret all INTEGER and LOGICAL as though declared *4
-list	-nolist	Generate a listing file
-messages	-nomessages	Allow printing of warning messages
-Npaths=*name*.in		Use the database of functions listed in the catalog *name*.in as the source for inlining
-no_assoc		Preserve the associativity of floating point arithmetic. Do NOT assume that it is associative.
-no_directive		Do not apply directives during compilation
-novector		Generate no vector code for any loop
-O0	-nooptimize	Turn off optimization
-*O1*		Do subexpression elimination. If nothing is specified on the command line, -O1 is the default setting of compiler optimization level
-O2		Do -O1 and vectorization
-O3		Do -O2 and parallelization
-O		Same as -O1
-o *filename*		Put output into filename
-object	-noobject	Generate an object file
-onetrip	-noonetrip	Make sure all DO loops execute at least once. The default is -noonetrip
-ploop		Profile each loop separately
-S		Generate assembly language source file
-safe=procs		Compile a procedure for parallel execution
-safe_strings		Generate code that corrects for mismatched string parameters
-save	-nosave	All variables declared are saved
-standard	-nostandard	Check for standard Fortran 77 usage
-subcheck		Add code to check linear array subscripts
-V		Print version information
-verbose		Use verbose message output
-vreport		Invoke vector reporting facility
-vsummary		Invoke vector reporting facility
-w		Suppress warning messages during compilation
-43		Use BSD system capability

7.1 Standard Compilation Options

This section describes several standard compilation options which are necessary just to compile, link, and run Fortran programs on the Stardent 1500/3000.

-c

Compile the source file and generate only object code output, but do not invoke the loader.

-cpp

Invoke the C preprocessor. If this is not used, all source lines with a pound sign (#) in column 1 are silently ignored. The options **-P** and **-E** always invoke the preprocessor.

NOTE
Line numbers on messages may be confused when **-cpp** is used. You can match up line numbers with the source file by using the **-list** option. Look at the first column of numbers in the *xx.L* file that is produced.

-D *hhhhhhhh*

Cause the loader to generate an *a.out* file with the data address at *hhhhhhhh.*

-D*name*

Define *name* to have the value of 1 to the preprocessor. It is effective only when used with **-cpp**.

-D*name=val*

Define *name* to have the value of *val* to the preprocessor. It is effective only when used with **-cpp**.

-E

Output expanded source to standard output. This option stops the compilation process after all macros and conditional compilation expressions have been expanded. A "real" compilation does not take place. Output of the preprocessor appears on standard output.

-g

Cause the compiler to generate information for the Stardent 1500/3000 debugger. It also forces optimization level to zero. The default is **-nodebug**. Options **-debug** and **-g** are synonymous.

-I

Suppress the default searching for preprocessor included files in */usr/include*. This option does not affect the INCLUDE statement; it only affects the C preprocessor.

-I*dir*

Search for preprocessor included files in *dir*. This does not affect the **INCLUDE** statement; it only affects the C preprocessor.

-i

Suppress the automatic production of #ident information.

-o *filename*

Cause the loader to place the output into *filename*.

-P

Invoke only the C preprocessor; no compiling or loading is done. The result of *x.f* goes into *x.i*.

-r

Cause the loader to produce a relocatable output file. This option is typically used to combine several object files into a single object file.

-S

Generate assembly files rather than object files. The result of compiling *x.f* goes into *x.s*, rather than *x.o*.

-U*name*

Cause *name* to be undefined in the C preprocessor. It is useful only when used with **-cpp**.

-w

Suppress warning messages during compilation.

-y*name*

Cause the loader to trace a file specified in *name* and print out all uses and definitions.

-43

Compile and link for executable code under Berkeley system instead of System V. Use */usr/lib/bsd/libc.a* instead of */usr/lib/libc.a*, and also use BSD header files.

7.2 Input/Output Options

The following options control the form of input received by the Fortran compiler.

-blanks72

Pad all lines in the source file on the right to column 72. Longer lines are untouched. This option does not work with **-cpp**, **-P**, or **-E**. Affects only *.f* files, and slows down compilation time. This option is useful if you have Holleriths or strings developed on card-oriented systems.

-continuations=*n*

Set the number of continuation lines the compiler accepts for any statement. The number *n* may range from 0 to 99; the default is 19.

-cross_reference

Generate a cross-reference if a listing is generated. The listing file for *x.f* is *x.h*. The default is **-nocross_reference**.

-nocross_reference

Turn off any cross-reference listing.

-d_lines

Compile source file lines with a **D** or **d** in column 1. The default is **-nod_lines**. The **D** or **d** is treated as if a blank were substituted for it.

-extend_source

Instruct the compiler to extend the statement field of a source line from columns 1 through 72 to columns 1 through 132.

-include_listing

List included source file as well as the original source file when generating a listing. The default is **-noinclude_listing**.

-noinclude_listing

Do not copy included source files to the listing.

-list

Generate a listing from the compilation. The listing file is named after the source file. For instance, the listing file for *x.f* is *x.L*. The default is **-nolist**.

-nolist

Do not generate a listing file.

7.3 Debugging Options

This section describes options that can be used to debug Fortran programs on the Stardent 1500/3000.

-debug

Generate information for the Stardent 1500/3000 debugger, and also forces optimization level to zero. The default is **-nodebug**. Options **-debug** and **-g** are synonymous.

-nodebug

Do not generate debugging information.

-fullsubcheck

CAUTION
Use of this option increases object code size and decreases execution speed.

Generate code to check that every subscript in every array reference is within the bounds of the appropriate array dimensions. For example, if the **DIMENSION** A(10,10) is declared, A(11,5) definitely violates the conditional definition of **-fullsubcheck** option even though it does not violate **-subcheck**.

-g

Generate information for the Stardent 1500/3000 debugger. It is synonymous to **-debug**.

-implicit

Make all variables in a program untyped. When using this option, all variables must be declared, or an error occurs. This option has the same effect as an **IMPLICIT NONE** statement at the beginning of each routine in a source file. The default is **-noimplicit**.

-noimplicit

Follow the ordinary Fortran 77 typing rules.

-standard

Check for standard Fortran 77 usage and flag Stardent 1500/3000 extensions with warnings. This option may appear in form 1 or 6. If form 6 is used, it may take suboptions shown in Table 7-4.

Table 7-4. Suboptions for -standard option

syntax	**nosyntax**
source_form	**nosource_form**
all	**none**

The default is **-nostandard**. If this option appears in form 1, it is the same as **-standard**=*all*. Refer to the end of this chapter for detailed information on form of options.

-nostandard

Turn off checking for extensions to Fortran 77. This option has the same structure as **-standard**.

-subcheck

Produce code to check at runtime to ensure that each array element accessed is actually part of the appropriate array. Consider the fragment

```
REAL SAM(10,20,30)
...
...
... = SAM(I,J,K) ...
```

If **I** has any value between 1 and 10, **J** between 1 and 20, and **K** between 1 and 30, then the access to an element of array **SAM** is legal. However, because of the way Fortran arrays are stored in memory, the reference still lies in the storage allocated for **SAM** when **I**=25, **J**=2 and **K**=4; in fact, the selected element is **SAM(5,4,4)**. In general, there are an infinity of ways to write subscripts that are not strictly legal, but which name an element lying in the array. If used, this option warns only about an access that falls completely outside of the storage for an array. In other words, this option does **not** check array subscripts individually, but simply checks that the result of the calculation of the array element's location is actually within the bounds of the array. The **-fullsubcheck** checks each array dimension individually.

CAUTION
At optimization level 02 and higher, **-subcheck** ignores the vector mask. This means that some operations may generate subscriptranges that are not actually in the code. If you want fully accurate subscriptranges checks, you should compile at lower optimization levels. Also note that use of this option increases object code size and decreases execution performance.

-V

Print compiler version information.

-verbose

Generate more messages tracking the progress of the compilation. As a default, these additional messages are suppressed.

7.4 Optimization Options

The Stardent 1500/3000 Fortran compiler supports several optimizing techniques. The following options allow you to optimize your source program.

-catalog=*filename*.in

Create a database of functions available to be inlined in subsequent compiles. Be aware that the use of **-catalog** creates two files: the database *filename*.in and an associated file *filename*.id. Each use of catalog adds to the catalog (it does not replace previous versions).

For example, to create a catalog in Fortran, specify the following on the command line:

```
fc -catalog=filename.in filename.f
```

-fast

At link time, this option causes a faster math library to be loaded; at compile time, it enables optimizations which may slightly change your computational results. For instance, **-fast** allows the compiler to convert "(a/b)/c" to "a/(b*c)". In addition, this option may cause math functions in your program to be potentially less accurate, as much as two decimal places in a double precision number. It is also not quite as observant of some math rules in the last bit or two. In general, this option allows optimization which may lose small amounts of precision.

-full_report

This invokes the vector reporting facility and shows you how things are vectorized, what code the compiler generated, and explains why things have been done. This is the most detailed report that is generated by the vector reporting facility. Refer to Chapter 8, *Stardent 1500/3000 Fortran Optimization Facilities* under the section titled *Vector Reporting Facility* for complete descriptions on this option.

-inline

When **-inline** is specified and a catalog is listed in **-Npaths**, the compiler inlines all legal functions found in the catalog. For example:

(1) If you specify `fc -inline` with no `-Npaths`

the compiler inlines from */usr/lib/libbF77.in* which has in it the BLAS subroutines. Additionally, the compiler inlines very short subprograms from your source code.

(2) If `fc -inline -Npaths=`*mycat*`.in`

then the compiler inlines from *mycat*.in and the standard system catalog */usr/lib/libbF77.in*.

(3) If `fc -inline -Npaths= -Npaths=`*mycat*`.in`

then the compiler inlines only from *mycat*.in and very short subprograms in your source code.

(4) If `fc -inline -Npaths=`

then the compiler inlines only from very short subprograms in your source code.

The Stardent 1500/3000 compiler can virtually inline any type of function into any language construct. However, there are some exceptions. The compiler does not inline character-valued functions or functions which take character arguments. C functions which appear to be *varargs* (the address of the argument is taken) cannot be inlined. Finally, the compiler does not inline any function into the **while** condition of a loop.

NOTE
Please refer to Chapter 8, *Stardent 1500/3000 Fortran Optimization Facilities* in the section titled *Inline Expansion* for further detailed information on the **-catalog**, **-inline**, and **-Npaths** options.

-Npaths=*filename*.in

instruct the compiler to use the database of available functions listed in the catalog (*filename*.in). Specifying **-Npaths=** avoids getting system functions.

-O0

Turn off all optimizations.

-O1

Perform common subexpression elimination and instruction scheduling. This is the optimization default setting of the compiler.

-O2

Perform **-O1** and vectorization.

-O3

Perform **-O2** and parallelization.

-O

This is the same as **-O1**.

-safe=procs

Cause the compiler to compile a procedure for parallel execution. Refer to Chapter 8, *Stardent 1500/3000 Fortran Optimization Facilities* in the section titled *User-Controlled Parallelism* for additional information on this option.

-vreport

This invokes the vector reporting facility and is used to tell the user what vectorization has been done to the program. A detailed listing is provided of exactly what the compiler did to each loop nest. This option does not include suggestions on how to achieve better performance. The output from this option is in Fortran-like notation. Refer to Chapter 8, *Stardent 1500/3000 Fortran Optimization Facilities* under the section titled *Vector Reporting Facility* for detailed descriptions on this option.

-vsummary

This invokes the vector reporting facility and is used to tell the user what vectorization has been done to the program. This option, **-vsummary**, prints out what statements are and are not vectorized in each loop nest. The output from this option is in Fortran-like notation. Refer to Chapter 8, *Stardent 1500/3000 Fortran Optimization Facilities* under the section titled *Vector Reporting Facility* for detailed descriptions on this option.

7.5 Porting Options

This section describes options which are helpful when porting source code to the Stardent 1500/3000.

-all_doubles

This option is intended to allow code written in single precision to easily run in double precision on the Stardent 1500/3000. It has the following effects:

(1) All single precision variables, whether declared REAL or REAL*4, are promoted to double precision values. COMPLEX variables are treated similarly.

(2) All single precision intrinsics and COMPLEX variables, whether generic or specific, are converted to double precision equivalents. For example, **AMINI** (single precision) is converted to **DMINI** (double precision). All single precision user functions are assumed to be double precision.

(3) All single precision constants are converted to double precision. Depending on context, this conversion may not occur at the time you expect. For instance, **AMINI(1.0e40, 1.0e50)** does not generate the value **1.0d40** under **-all_doubles**, but instead generates **+INF** (positive infinity). The reason is that both constants must be held as single precision in order to properly parse the single precision function. Both numbers are larger than can be represented in Stardent 1500/3000 single precision, so they become infinity. This single precision infinity eventually becomes a double precision infinity.

This effect can be avoided by using generic intrinsics, rather than specifics, and using **-double_precision** in addition to **-all_doubles**.

(4) Integers are allocated 8 bytes of storage instead of the usual 4 bytes. All integer arithmetic is still 4 bytes long; however, memory to memory copies copy the full 8 bytes.

-double_precision

Set all variables that are implicitly declared REAL or that appear in a REAL statement (but not REAL*4) to be double precision. COMPLEX is treated similarly. Generic intrinsics and constants are also treated appropriately. Integers are not affected. This option can be reasonably used with **-all_doubles**.

When using **-double_precision** you should keep in mind that **-double_precision** is applied only at the time the source file is scanned, not after the inlining occurs. Thus, compiling "-inline -double_precision -Npaths=samplefile.in" does not correctly handle the inlined functions if they are single precision functions—the inlined functions are unaffected by the **-double_precision**. The correct way to do is to apply the **-double_precision** option at the time the catalog is built.

Furthermore, be aware that use of this option affects functions and their arguments, which may cause them not to work. For example, *cputim(3F)* does not work with this option because a REAL is converted to a REAL*8. Make sure that you understand the implications of using this option with functions and their arguments.

-nodouble_precision

Treat floating point variables and constants by the Fortran 77 rules; specifically, compile them as REAL*4 unless otherwise specified in the source.

-safe_strings

Generate code that corrects for mismatched string parameters. Use of this option decreases execution performance.

The Fortran 77 standard states that actual arguments and formal arguments must agree in type. Because character variables were not explicitly present in Fortran 66, many users tend to pass Fortran strings to functions expecting doubles, etc., and to receive as strings arguments that are actually doubles, etc. The parameter-passing conventions of the Stardent 1500/3000 typically cause such invalid references to coredump. For instance, the following code

```
DOUBLE PRECISION X
CALL RECEIVE (X)
CALL PRINTOUT (X)
END
SUBROUTINE RECEIVE (X)
CHAR*4 X
X = 'abcd'
END
SUBROUTINE PRINTOUT (X)
CHAR*4 X
PRINT *, X
END
```

will fail if compiled and run under the Stardent 1500/3000 compiler, although it will run correctly under many other

compilers. Compilers which can handle this type of construct correctly pay a penalty — they typically must put in code at every call to check that a parameter being passed in is the type it is supposed to be and to convert it if not. The Stardent 1500/3000 compiler strictly enforces the standard of this aspect because it does not want to pay the execution penalty involved at every procedure call. In particular, strings are typically passed or received in a very small number of cases.

The option **-safe_strings** is available to the user to force the compiler to generates code that corrects for mismatched string parameters. If you suspect you have code similar to that above, then this option will cause the Stardent 1500/3000 compiler to generate code that allows your program to run. Since there is a performance penalty involved, you should use this option only if you think it is necessary.

7.6 Miscellaneous Options

The following section describes other options that have been explained previously but briefly in the table.

-ast=*number*

This flag enables the compiler to set the initial allocation for the program internal representation to the user defined *number* bytes. For example, if -ast=1000, a block of 1000 bytes is initially allocated to hold the program internal representation. Normally, this flag should not be invoked unless the program is terminated with the internal error. If the program is terminated, the internal error indicates exactly the option line to use.

-B *hhhhhhhh*

This loader option causes the *a.out* file to be generated with the bus address at *hhhhhhhh*.

-case_sensitive

Insist that user defined variables must be case sensitive. For example, if this option is invoked, the variables **BuFfEr** and **bUfFeR** are not the same. In another word, this option prohibits the case conversion operation on the variables. The default is **-nocase_sensitive**.

-nocase_sensitive

Do not enforce user defined variables to be case sensitive. Thus, all user variables are converted to uppercase equivalents. For example, two variables, **Buffer** and **buFFER** are identical. This is the default of the compiler mode.

-e*sym*

This loader option sets the default entry point address for the output file to be that of *sym*.

-i4

Interpret INTEGER and LOGICAL declarations as if they had been written INTEGER*4 and LOGICAL*4.

-noi4

Do not interpret INTEGER and LOGICAL declarations as if they had been written INTEGER*2 and LOGICAL*2. The default is **-i4**.

-L

Cause the loader not to search for **-l** libraries in */lib* or */usr/lib*.

-L*dir*

Cause the loader to search for **-l** libraries in *dir*.

-l*tag*

Cause the loader to search a library called *libtag.a*. The default is to search first in */lib* and then in */usr/lib*.

-m

Cause the loader to generate a simple load map on standard output.

-messages

Allow warning messages to be printed. The default is **-nomessages**.

-nomessages

Do not allow warning messages to be printed.

-n

Cause the loader to generate NMAGIC file type.

-no_directive

Do not apply compiler directives. The default is to apply directives.

-no_assoc

Cause the compiler to preserve the associativity of floating point operations. Some programs are extremely sensitive to slight changes in floating point results. Since floating point hardware is not associative, changing the order of operations (e.g. as happens in a sum reduction or dot product operation) can cause changes in the final results. The **-no_assoc** flag causes the compiler to assume that floating point operations are not associative, and it is very useful on very sensitive numeric programs.

-novector

Generate no vector code for any loop. This option is very useful at optimization level 03 when one desires parallel code but no vector code.

-object

Generate object files from this compilation. This is not quite the same as the **-c** option, which precludes loading; **-noobject** precludes even the generation of object files.

-noobject

Do not generate object files from this compilation; check correctness of the source code only. The default is **-object**.

-onetrip

Generate code to guarantee that all **DO** loops execute at least once. This is a compatibility feature for programs originally written for use with Fortran 66. The default is **-noonetrip**.

-noonetrip

DO loops may execute zero times.

-opct

At link time, this option inserts code into the program that produces a count of all the FPU ops executed by the program, dynamically.

-p

At link time, this option causes the compiler to generate code to profile the source file during execution. Refer to Chapter 9, *Profiling Programs* for additional information on this option.

-ploop

Allows loops (DO loops) within a single routine to profiled separately. Refer to Chapter 9, *Profiling Programs* for detailed information on this option.

-s

Cause the loader to strip line numbers and symbol table information from the output object file.

-save

Save all declared variables. The default is **-save**.

-nosave

This has no effect because Stardent 1500/3000 Fortran automatically saves all variables.

-T *hhhhhhhh*

Cause the loader to generate an *a.out* file with the text address at *hhhhhhhh*.

-t

Cause the loader to turn off warnings about multiply-defined symbols that are not the same size.

7.7 Form of Compiler Options

Stardent 1500/3000 compiler options, when used, take one of the forms shown in Table 7-5.

Table 7-5. Form of Compiler Options

Form	Option
1	*-option*
2	*-option symbol_or_file_path*
3	*-option number*
4	*-option symbol*
5	*-option=number*
6	*-option=symbol_or_file_path_list*

A *symbol_or_file_path_list* can be empty; if the list is not empty, the list should be a comma-separated list of symbols and file pathnames. Generally, options have the first, fifth, or sixth forms; the second, third, and fourth forms are loader options which are passed on to the loader.

A *symbol_or_file_path_list* must not include blanks. The symbols generally allow the use of letters, numbers, and special characters that are not otherwise meaningful to the shell or to the option parser.

Options are applied to the compiler in the order written on the command line, from left to right. Similarly, suboptions (the items to the right of an equal sign in form 6, for example, **-standard** can take several suboptions) are applied to an option in the order written, from left to right. Options for the loader are passed to the loader in the order specified on the command line from left to right.

Options and filenames may be intermingled so that libraries may be searched in the proper order with reference to particular object files; **-l***tag* loader options must sometimes precede filenames. However, other options are collected and applied all at once to each of the processes invoked; placement of an option such as **-O1** after a filename still causes the **-O1** option to be applied to all files compiled.

TITAN FORTRAN OPTIMIZATION FACILITIES

CHAPTER EIGHT

The Stardent 1500/3000 is a parallel computer, providing not only vector instructions but also multiple processing elements. While almost all programs runs very quickly on a Stardent 1500/3000, the programs that make the most effective use of the hardware capabilities are those that most fully exploit the parallelism available in the Stardent 1500/3000. This chapter describes the facilities provided by the Stardent 1500/3000 Fortran compiler that help you take full advantage of the Stardent 1500/3000 architecture. These facilities include:

- Vector reporting facility
- Program transformations
- Compiler directives
- Inline Functions
- User-controlled parallelism
- Asynchronous I/O

8.1 Vector Reporting Facility

At higher optimization levels (**-O2** for vector, **-O3** for vector and parallel), the Stardent 1500/3000 Fortran compiler automatically employs a number of very sophisticated program transformations to uncover vector and parallel loops in a program. For most programs, these transformations enable the compiler to generate code that effectively utilizes the Stardent 1500/3000 hardware. However, the Stardent 1500/3000 Fortran compiler is not omniscient, and it may occassionally choose a poor loop to vectorize. For instance, when loop lengths are unknown at compile-time, the Stardent 1500/3000 compiler may vectorize a loop that turns out to have some length at run-time, thereby slowing down the program slightly. Similarly, some small details of algorithm design

or coding style sometimes prevent the Stardent 1500/3000 Fortran compiler from vectorizing a loop. To help you uncover and fix problems like this, the Stardent 1500/3000 Fortran compiler contains an extensive vector reporting facility to help tune your program.

Stardent 1500/3000 Fortran provides three different compiler options which can be used to invoke the vector reporting facility. Through the use of these options, namely **-vsummary, -vreport,** and **-full_report,** you can see the parallelism that the compiler has found in your program as well as get an indication of why the compiler was unable to vectorize some loop. In addition, the **VREPORT** directive can be used to focus on specific loops without getting a report for a whole program. The output of the reporting facility goes into a file named the same of the source file, but with the .f replaced by .V. For instance, compiling a file "sample.f" for vector reporting creates a file "sample.V" that contains the vector report.

8.1.1 -vsummary

This option is designed for programs that have been tuned on other vector machines and are being ported to the Stardent 1500/3000. The user expectation in this case is that all the key loop nests vectorize and parallelize. The **-vsummary** option permits the user to easily confirm that this is the case or to quickly locate the loops that do not vectorize if this is not the case. Under this option, the compiler prints out a one line summary for each loop nesting, either stating that all statements vectorized or indicating which statements did not. The **-vsummary** option provides no diagnostics as to why loops did not vectorize, nor does it indicate exactly how a given loop nest vectorizes. The **-vreport** and **-full_report** options can be used to uncover this information about loops pinpointed by **-vsummary**.

EXAMPLE

The following is an example of a program fragment and the output produced by compiling with **-vsummary**.

```
DOUBLE PRECISION A(100,100), B(100,100), C(100,100)
INTEGER I, J, K
DO I = 1, 100
   DO J = 1, 100
      C(I,J) = 0.0
      DO K = 1, 100
         C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

```
      ENDDO
    ENDDO
  ENDDO
```

and the **-vsummary** report is

```
Vectorization summary for file x.f
**************************************
Line      3:  All statements vectorized.
```

The report indicates the line number at which the loop nest is located and a message indicating one of three cases: all statements vectorized (as above), the loop was not examined by the Stardent 1500/3000 vectorizer (which happens if the loop contains a branch that exits the loop or a function call), or some statements in the loop were not vectorized.

8.1.2 -vreport

Sometimes just knowing that all the statements in a loop vectorized is not enough. For instance, if there are multiple loops in a nest, it may be important that the Stardent 1500/3000 compiler vectorize the correct loop (it only vectorizes one) since there may be short loops or loops with bad strides (a stride is a constant distance in memory between elements). The **-vreport** option provides information on how and why a loop vectorizes, plus information on other transformations that the Stardent 1500/3000 compiler may have performed. To illustrate, following is the output from **-vreport** on the previous example.

EXAMPLE

```
        Vectorized Results from File x.f
                Origin - - Line 3

Line    Stmt    Time            Program

5       6       3               b2 = 100
7       10      3               b3 = 100
5       *       4               DO PARALLEL J=1, 100
4       *       4                  DO iv=1, 100, 32
4       *       8                     rv = MIN(100, 31 + iv)
*       *       9                     vl = rv - iv + 1
*       *       9                     vl = rv - iv + 1
*       *       6                     DO VECTOR I=iv, rv
6       8       33                       C(I, J) = 0.0D0
                                      END DO
7       *       4                     DO K=1, 100
4       *       6                        DO VECTOR I=iv, rv
8       12      119                         C(I, J) = C(I, J) + A(I, K) * B(K, J)
```

```
            END DO
          END DO
        END DO
      END DO
```

At the far left of the report is a listing of the line numbers of each statement in the original source program. In this particular example, the line number information is not of much value, but in some cases, the Stardent 1500/3000 compiler may change the statement order to enhance vectorization. In those cases, the line number information can be of great value.

Asterisks in the line number field indicate statements that were generated by the compiler and that do not appear in the original source program. The "Stmt" field indicates a unique statement number assigned by the compiler. Diagnostics and suggestions printed by the compiler are keyed to this field, since the "Line" field is not always unique (for instance, there are three statements derived from line 4 in the example). Again, asterisks indicate a statement which for some reason did not receive a statement number—such statements are always of little significance in the program execution. The "Time" column is a static estimation, by the compiler, of roughly the number of clock ticks the statement takes to execute. Static estimations are never very accurate, so you should not rely on this field to estimate the execution time of your program, but it can be used to indicate the statements that should receive your special attention. Thus, the 119 ticks required for each execution of statement 12 (since it is a vector statement, each execution operates on 32 elements) is probably not a good estimate of the actual time the statement took, but it does indicate that it is far more important that it vectorizes than statement 8, which only took 33 ticks.

The compiler attempts to print out the program listing itself (the rightmost column) in a form as close as possible to the original code. **DO** loops that have been parallelized (i.e. line 5) are indicated as **DO PARALLEL**; vector operations are indicated by a **DO VECTOR** loop (i.e. line 4) so that the full subscripts can be presented, rather than one linearized version. Unless the *-case_sensitive* flag is used, variables that are in the source program are printed in upper case and temporaries generated by the compiler are printed in lower case. Thus, **I**, **J**, and **C** are all variables from the original program, whereas **b1** and **b2** are temporaries generated by the compiler.

Some compiler temporaries hold information that may be useful to you. In general, variables that end with a number hold some

piece of information about the loop at the nesting level indicated by the number. The variable **b2** tells something about the loop at nesting level 2 (the original **DO J** loop) and **b3** indicates something about the level 3 loop (originally the **DO K** loop). In fact, **b** variables indicate the number of times the given loop will iterate: both the **DO J** and **DO K** loops iterate 100 times each time they are executed. There are also **r** variables for each loop (although the compiler has eliminated them in this example), which indicate the upper bound for each loop. When the Stardent 1500/3000 compiler vectorizes a loop, it "strip-mines" it to a vector length of 32. That is, it breaks the loop into two loops: an inner loop, always of length 32 or less (which becomes the actual vector operation), and an outer loop (called the strip loop) to sweep inner loop across the full range of the original loop. By performing this transformation, the compiler guarantees that all vector operations executed in the Stardent 1500/3000 hardware are less than 32 in length, thereby making vector register allocation much easier. In the vector report, strip loops are always controlled by either the variable **iv** or **ip**, and the actual vector loop is controlled by the original user loop variable.

In the example above, the strip loop is the outermost loop from line 4, which corresponds to the original **DO I** loop, and the actual vector operations are deeper inside. The variables **rv** and **rp** are used to hold the upper bound for each vector operation during one iteration of the strip loop. Finally, the variable **vl** is used to hold the length of vector operations for each iteration of the strip loop. In the example, the first three iterations of the strip loop uses a vector length of 32; the last iteration does a cleanup operation of length 4. **vl** holds these values on each respective iteration of the strip loop. For various technical reasons, the Stardent 1500/3000 compiler occassionally must generate multiple assignments to **vl** (as in the previous example). There is no need to worry about these cases, as they take a very small amount of execution time.

Looking back at the example output, you can tell already that the Stardent 1500/3000 Fortran compiler has significantly transformed the source program. The compiler decided that the **DO I** loop, the outermost loop in the original nest, was the best loop to vectorize, and stripped it to a length of 32. For statement 12, it switched the actual vector loop with the **DO K** loop, moving the **K** loop outward, to get better performance. The compiler then decided that the **DO J** loop (the original level 2 loop) was the best loop to run in parallel. Since the more computation a parallel loop contains, the more effective it is, the compiler then interchanged the parallel **DO J** with the strip loop from the **I** loop, so

that the parallel loop becomes the outermost loop in the nest. The resulting code makes very effective use of the Stardent 1500/3000 hardware.

There is one other important feature of **-vreport** that is not evident in this example. The Stardent 1500/3000 vectorizer is designed to be very machine-independent and vectorizes loops based on the properties of the statements in the loop, independent of the capabilities of the Stardent 1500/3000 processor. Of course, it is very difficult to execute a vector operation for which there is no vector hardware. As a result, the Stardent 1500/3000 compiler executes a later, machine-dependent pass that converts vector operations for which there is no vector hardware into an equivalent sequence of scalar operations. Obviously, the scalar operations do not run as fast as the vector operations, so the compiler also prints a message in the vector report indicating the vector operation it converted. The following is an example of such a message:

```
Following vector ops in line 4 run at scalar speed
                    DIV (integer)
```

This message lets you know that there are no vector divide instructions for integers on the Stardent 1500/3000, so that it might be worthwhile coding the loop in a different way. Similarly, some parallel loops require more memory bandwidth than the Stardent 1500/3000 hardware can deliver. In those cases, the Stardent 1500/3000 compiler runs the loop sequentially, and warns in the vector report with the following message:

```
Parallel loop at line 3 made scalar due to memory bandwidth.
```

8.1.3 -full_report

Sometimes the Stardent 1500/3000 Fortran compiler does not vectorize loops that you expect to vectorize, or vectorizes different loops than you expect. In those cases, it is valuable to understand why the Stardent 1500/3000 vectorizer made the choices it did. The **-full_report** option provides you with this information. In cases where loops vectorize, the **-full_report** option tells you why specific loops were chosen as vector and parallel loops. In cases where no loops vectorize, the **-full_report** option tells you why the compiler was unable to vectorize any loops, and makes suggestions that may speed up your program.

EXAMPLE

The following example uses the previous example fragment. The following is the output produced when **-full_report** is invoked, but with the program listing removed. The program listing for **-vreport** and **-full_report** are identical.

```
        Vector Parallel Statements

Stmt  Parallel Choices  Chosen  Reason  Vector Choices  Chosen  Reason

  8              1, 2       2                  1, 2       1    (2)
 12              1, 2       2               1, 2, 3       1    (2)


(2)   Stride one access influenced selection of vector loop.
```

In the sample program, everything vectorizes and parallelizes, so the information conveyed by the **-full_report** option relates mainly to its choice of loops to vectorize and parallelize. The **-full_report** option divides statements up into four categories: "Vector Parallel" (as above), "Vector", "Parallel", and "Neither Vector nor Parallel".

In the report above, the Stardent 1500/3000 compiler is informing you that it has detected that the outer two loops (loops 1 and 2) can be run in parallel, and that any of the loops can be run in vector around either of the statements. The Stardent 1500/3000 compiler selected the loop that was originally at nesting level 2 to run in parallel, and the loop that was originally at nesting level 1 to run in vector. It chose the vector loop because it has stride one access to the most memory locations. While most strides are fine for accessing memory, the Stardent 1500/3000 hardware has been designed to perform extremely well on stride one access. Once loop 1 is chosen as the vector loop, loop 2 is the best candidate for parallel execution.

Other examples of reporting output are used to describe program transformations. Below is a list of all the diagnostics that may be printed by the reporting facilities, as well as a rough description of its meaning.

(1) Length of vector operation influenced selection of vector loop.

This diagnostic indicates that the vector loop was chosen because it appeared to have the longest length of any of the vector candidates.

(2) Stride one access influenced selection of vector loop.

This diagnostic indicates that the vector loop was chosen because it appeared to have more stride one memory accesses than any of the vector candidates.

(3) Scatter-gathers influenced selection of vector loop.

This diagnostic indicates that the vector loop was chosen because it had fewer scatter-gather operations than any of the other vector candidates.

(4) Length of loop influenced selection of parallel loop.

This diagnostic indicates that the loop chosen to run in parallel was selected over other loops because it appeared to have the longest length.

(5) Outmost possible loop selected as parallel loop.

This diagnostic indicates that the compiler selected the outmost possible parallel loop. While this may appear to be the obvious choice, the Stardent 1500/3000 compiler often interchanges better parallel candidates to the outermost position and parallelize them.

(6) Nonlinear subscripts eliminated possible vector loops.

In early versions of the Stardent 1500/3000 compiler, nonlinear subscripts (i.e. A(I*I), where I is a loop variable) prevented vectorization of some loops. This diagnostic should no longer appear, as the Stardent 1500/3000 compiler now generates scatter-gather code for these cases.

(7) A function without a vector analog prevented vectorization.

In early versions of the Stardent 1500/3000 compiler, a function call that did not have a vector analog inhibited vectorization of that statement. That is no longer true; such function calls get devectorized as part of the machine-dependent phase of vectorization. As a result, this diagnostic should no longer appear.

(8) A function without a parallel version inhibited parallelism.

A function that is not "parallel-callable" inhibits parallelization of a loop. For technical reasons, such functions also inhibit some vectorization as well. Since all Stardent 1500/3000 math intrinsics are now parallel-callable, this diagnostic should no longer appear.

(9) PBEST directive mandated parallel loop selection.

The parallel loop was chosen because of a **PBEST** directive in the code.

(10) VBEST directive mandated vector loop selection.

The vector loop was chosen because of a **VBEST** directive in the code.

(11) Dependencies prevent vectorization and parallelization of some loops.

Some form of memory overlap involving this statement prevents vectorization and parallelization of some loops. If the **-full_report** option is used, the compiler prints out all dependencies that it thinks may be involved.

(12) No vector option inhibits vectorization.

The compiler did not vectorize any statements, either because of a **NO_VECTOR** directive or because of the **-no_vector** option on the command line.

(13) Control branches inhibit vectorization.

The Stardent 1500/3000 compiler cannot at present vectorize statements under the influence of conditional branches (goto's, assigned goto's, and so on).

(14) Cannot vectorize triangular loop — unable to interchange.

The compiler detected an outer loop that could be vectorized, but was unable to vectorize it because an inner loop has one of its upper bound, lower bound, or strides that depend on the outer loop. Interchanging those loops is an extremely complicated transformation, one that the Stardent 1500/3000 compiler cannot yet do.

(15) Loop was split into two loops to get vector and parallel execution.

The compiler could only find one loop to vectorize and parallelize, so it parallelized the strip loop. This does not always give very effective parallelization.

(16) Parallelization inhibited by optimization level.

The optimization level was too low to turn on parallelization.

(17) Wrap around scalar value inhibits parallelization.

A scalar that wraps around from one iteration to the next permitted parallelism, but not vectorization. For instance, the Stardent 1500/3000 compiler can vectorize the following:

```
T = A(N)
DO I = 1, N
   B(I) = A(I) + T
   T = A(I)
ENDDO
```

as

```
DO iv=1, N, 32
   tv_t(1) = T
   rv = MIN(N, 31 + iv)
   vl = rv - iv + 1
   DO VECTOR I=iv, rv
      tv_t(2 + I - iv) = A(I)
      B(I) = A(I) + tv_t(1 + I - iv)
   END DO
   T = tv_t(2 + rv - iv)
END DO
```

However, it cannot parallelize this loop because the scalar value wraps around from iteration to iteration. Doing some minor rewriting (unrolling one loop iteration) can typically improve the performance of such loops.

NOTE
Refer to next section titled *Reduction Recognition* for detailed information on *count reductions* and *idamaxes*.

(1) Reduction operation can only be done in vector.

In early versions of the Stardent 1500/3000 compiler, reduction operations such as sum reduction, dot product, and so on, could not be done in parallel. Now, only a few, like count reductions and idamaxes cannot be done in parallel.

(2) Only assignment statements can be vectorized.

CALL statements, GOTO statements, and so on, have no vector hardware support.

(3) Dependencies prevent vectorization.

The compiler detected some memory overlap that inhibited vectorization of this statement. **-full_report** prints out the dependencies the compiler found.

(4) Dependencies prevent parallelization.

The compiler detected some memory overlap that

inhibited parallelization of this statement. **-full_report** prints out the dependencies the compiler found.

(5) Loop length too small to justify parallel execution.

The compiler detected that the loop could be run in parallel, but it was so small as to not be worth it.

8.2 Program Transformations

The Stardent 1500/3000 is an extremely sophisticated compiler, and performs a number of program transformations to uncover and enhance the parallelism within a program. This section details many of those transformations, and gives examples of the resulting code in **vreport** (vector report) format. With these examples, you should be able to recognize the changes in your program effected by the vectorizer. If you need further detailed information on this topic, refer to the *Programmer's Guide*, Chapter 6, *Optimization*.

8.2.1 Induction Variable Elimination

The *loop induction variable* is the variable named in the loop control statement that steps through the iterations of the loop. The variable **I** in the loop below is the induction variable. A common programming practice, particularly in older Fortran programs, is to use an *auxiliary induction variable* to control some part of the loops execution. For instance, an auxiliary variable may be used to step through an array with strides of two, or to start at an offset deeper in an array, or (as in the following example with variable **IN**) to step through an array backwards.

```
IN = N
DO I = 1, N
   A(IN) = B(I) + C(I)
   IN = IN - 1
END DO
```

The loop adds arrays **B** and **C** and assigns the result to **A** but in **reverse** order. In this form, the loop cannot be directly vectorized, because **IN** hides the variance of array **A** on the loop. If you compile the above program fragment to obtain the vector report, you see the following:

```
DO iv=1, N, 32
   rv = MIN(N, 31 + iv)
   vl = rv - iv + 1
   DO VECTOR I=iv, rv
      A(1 - I + IN) = B(I) + C(I)
```

```
      END DO
   END DO
```

The Stardent 1500/3000 vectorizer has replaced the variable IN within the loop with an expression that varies directly with the loop variable, and as a result, the loop gets vectorized. The Stardent 1500/3000 compiler is able to recognize and replace a broad range of auxiliary induction variables, so normally you need not worry about this phase. If, however, you have an auxiliary induction variable which the Stardent 1500/3000 compiler does not eliminate (due to equivalences or common, perhaps), then you want to eliminate it by hand. No use of an auxiliary induction variable (that is not eliminated by induction variable substitution) vectorizes.

8.2.2 Constant Propagation

One of the key hindrances to automatic code vectorization is a lack of information regarding the values of variables. When a variable whose value is only known at runtime is used in a subscript of an array, the Stardent 1500/3000 compiler must assume the worst case with regard to dependencies, which often means that code does not vectorize. To alleviate this problem, the Stardent 1500/3000 compiler performs global constant propagation. When the compiler can prove that a particular use of a variable always has the same constant value, it replaces that use with the constant value. For example, consider a slight variation of the previous example:

```
N = 100
IN = N
DO I = 1, N
   A(IN) = B(I) + C(I)
   IN = IN - 1
END DO
```

The **-vreport** output for this fragment is as follows:

```
DO iv=1, 100, 32
   rv = MIN(100, 31 + iv)
   vl = rv - iv + 1
   DO VECTOR I=iv, rv
      A(101 - I) = B(I) + C(I)
   END DO
END DO
```

The Stardent 1500/3000 Fortran compiler has eliminated the auxiliary induction variable, and has recognized that the use of N is constant both in the bound of the loop and in the replacement

expression generated for **IN**. In general, the constant propagation algorithm employed by the Stardent 1500/3000 compiler is very effectiver. If you see variables that you believe are constant valued that the compiler has not recognized, there is probably a control path or equivalence that you have missed.

8.2.3 Dead Code Elimination

Dead code is code where the results are never used during the program execution. Users rarely create dead code, but many compiler transformations can. For instance, in the previous example, the assignment to **N** is dead once all uses of **N** have been replaced by 100. The Stardent 1500/3000 compiler eliminates such code automatically.

Similarly, some parts of a program may be unreachable at runtime. Again, users rarely create such code, but transformations such as procedure inlining can. The Stardent 1500/3000 compiler eliminates unreachable code to save space in the object file. Since the code is never executed, eliminating it obviously does not affect the execution time. When this elimination occurs in user code, a warning message is issued by the compiler since this condition usually reflects a user oversight. When the elimination occurs in inlined code, the compiler silently removes it.

8.2.4 Loop Distribution

Sometimes the compiler can vectorize parts but not all of the loop. Consider the following loop.

```
      DO 80 I = 1, N
         A(I) = A(I-1) + A(I)
         B(I) = C(I) + D(I)
80    CONTINUE
```

The assignment to **B** is vectorizable, but the assignment to **A** is not because it contains a recurrence. The Stardent 1500/3000 compiler partially vectorizes this loop by *distributing* the loop around the two statements.

NOTE
A **recurrence** is a cycle in a dependence graph; it occurs when a statement **depends** upon itself, reusing its own results.

```
      DO 80 I = 1, N
         A(I) = A(I-1) + A(I)
80    CONTINUE
      DO 85 I = 1, N
         B(I) = C(I) + D(I)
85    CONTINUE
```

It then vectorizes the second loop, producing the following vreport:

```
DO iv=1, N, 32
   rv = MIN(N, 31 + iv)
   vl = rv - iv + 1
   DO VECTOR I=iv, rv
      B(I) = C(I) + D(I)
   END DO
END DO
DO I=1, N
   A(I) = A(I - 1) + A(I)
END DO
```

Note that in the process of vectorizing the loop, the Stardent 1500/3000 vectorizer changed the order in which the statements are executed.

8.2.5 Loop Interchange

The Stardent 1500/3000 Fortran compiler, unlike many other vectorizing compilers, considers any loop in a nest to be a viable candidate for vectorization. The transformation which enables this is loop interchange. The Stardent 1500/3000 compiler is able to determine when the order in which loops are executed can be safely and profitably changed, and may often execute loops in a different order than you specified. Matrix multiplication is a common example that illustrates the value of this transformation.

```
DO I = 1, N
   DO J = 1, N
      C(J,I) = 0.0
      DO K = 1, N
         C(J,I) = C(J,I) + A(J,K)*B(K,I)
      END DO
   END DO
END DO
```

Any of the loops can be vectorized in this example, but the best loop to vectorize around both statements is the **J** loop. The **K** loop is a dot product reduction, which is slightly less efficient than a regular vector operation, and the **I** loop accesses many arrays with non-unit stride. The Stardent 1500/3000 compiler recognizes the **J** loop as being the best vector loop, producing the following vreport:

```
b2 = MAX(N, 0)
b3 = MAX(N, 0)
DO I=1, N
   DO iv=1, N, 32
      rv = MIN(N, 31 + iv)
      vl = rv - iv + 1
      DO VECTOR J=iv, rv
         C(J, I) = 0.0D0
```

```
      END DO
      DO K=1, N
         DO VECTOR J=iv, rv
            C(J, I) = C(J, I) + A(J, K) *
         END DO
      END DO
   END DO
END DO
```

Note that the **DO VECTOR J** loop has been interchanged inside the **K** loop. This vectorization generates the fastest possible code for this formulation of matrix multiplication.

In general, the ability to interchange loops permits the Stardent 1500/3000 Fortran compiler to generate extremely efficient vector and parallel code. However, this ability may occassionally cause the compiler to generate poor code, particularly for programs that have been tuned for compilers that can only vectorize innermost loops. For instance, in the following code fragment:

```
DO I = 1, M
   DO J = 1, N
      A(I,J) = A(I,J) * A(I,J)
   ENDDO
ENDDO
```

the Stardent 1500/3000 compiler vectorizes the outer loop as

```
b2 = MAX(N, 0)
DO iv=1, M, 32
   rv = MIN(M, 31 + iv)
   vl = rv - iv + 1
   DO J=1, N
      DO VECTOR I=iv, rv
         A(I, J) = A(I, J) * A(I, J)
      END DO
   END DO
END DO
```

since it does not know the loop lengths, and the outer loop provides unit stride access to memory. If at run time **M** happens to be very small (say 3) and **N** happens to be very large (say 1000), then this vectorization eventually runs much slower than vectorizing the inner loop. This situation tends to happen most often on code that has been tuned for compilers that only vectorize inner loops. A simple **VBEST** directive inserted on the inner loop lets the Stardent 1500/3000 compiler know that it is the best vector loop, causing it to generate the best vector code.

8.2.6 Scalar Expansion

The compiler cannot vectorize loops that reuse computations. Scalar temporaries provide one way for a loop to reuse its computations and, therefore, to inhibit vectorization and parallelization. Suppose that the previous matrix multiplication had been coded to take optimal advantage of a scalar machine by using a scalar temporary to accumulate the results.

```
      DO 100 I=1, N
         DO 101 J=1, N
            T = 0.0
            DO 102 K=1, N
               T = T + A(J,K)*B(K,I)
102         CONTINUE
               C(J,I) = T
101      CONTINUE
100   CONTINUE
```

This loop cannot be directly executed in vector as it stands, because the scalar temporary blocks correct use of the vector unit. The Stardent 1500/3000 compiler recognizes this blockage, and overcomes it by expanding the scalar temporary into a vector temporary:

```
b2 = MAX(N, 0)
b3 = MAX(N, 0)
DO I=1, N
   DO iv=1, N, 32
      rv = MIN(N, 31 + iv)
      vl = rv - iv + 1
      DO VECTOR J=iv, rv
         tv_t(2 + J - iv) = 0.0D0
      END DO
      DO K=1, N
         DO VECTOR J=iv, rv
            tv_t(2 + J - iv) = tv_t(2 + J, K) + A(J, K) * B(K, I)
         END DO
      END DO
      DO VECTOR J=iv, rv
         C(J, I) = tv_t(2 + J - iv)
      END DO
   END DO
END DO
```

When the Stardent 1500/3000 compiler expands a scalar, it preprends the prefix **tv_** to the scalar name for the new vector name (i.e. **tv_t** for the variable **T** above). As a rough rule, the Stardent 1500/3000 compiler is able to expand any scalar where there is a clear definition before any use in the loop (as above) or where all uses come before the first definition. If uses and definitions are

mixed haphazardly, then the Stardent 1500/3000 compiler cannot figure out a correct expansion. The Stardent 1500/3000 compiler does not expand scalars that are equivalenced.

8.2.7 Reduction Recognition

Many important computations require a loop to use results that it has previously computed. The most common of these operations reduces a vector into a single element as in the following loop.

```
      DO 110 I=1, N
         T = T + A(I)
110   CONTINUE
```

The operation's use of the result of the previous iteration prohibits execution on the vector unit. Because of this operation's importance, Stardent 1500/3000 contains special hardware, called *reduction hardware,* to speed up its computation. This is represented in the vector report by a reduction operation:

```
DO iv=1, N, 32
   rv = MIN(N, 31 + iv)
   vl = rv - iv + 1
   DO VECTOR I=iv, rv
      T = T + SUM_REDUCTION(A(I))
   END DO
END DO
```

The Stardent 1500/3000 compiler recognizes the following reduction recognitions for which there is special purpose hardware:

- SUM_REDUCTION — taking the sum of all the elements of a vector.
- PROD_REDUCTION — taking the product of all the elements of a vector.
- DOT_PRODUCT — taking the inner or dot product of two vectors.
- MAX — finding the maximum element of a vector.
- MIN — finding the minimum element of a vector.
- ANY_REDUCTION — taking the OR reduction of a boolean vector.
- ALL_REDUCTION — taking the AND reduction of a boolean vector.

- PARITY_REDUCTION — taking the exclusive OR reduction of a boolean vector.
- COUNT_REDUCTION — counting up the non-zero entries in a vector.

Additionally, the Stardent 1500/3000 compiler also recognize special index reductions for which there are vectorized subroutine calls. For instance, the vreport for the following code:

```
DO I = 1, N
   IF (ABS(A(I)) .GT. MAX) THEN
      INDEX = I
      MAX = ABS(A(I))
   ENDIF
ENDDO
```

shows that the compiler has converted the bulk of the work into a vectorized library call:

```
r1 = N
cs6 =  IDAFMAX(b1, A, 1)
IF (ABS(A(1 + cs6)) .GT. MAX .AND. b1.NE. 0) INDEX = 1 + cs6
IF (ABS(A(1 + cs6)) .GT. MAX .AND. b1.NE. 0) MAX = ABS(A(1 + cs6))
```

The routine IDAFMAX (this name is an artifact of vreport conversions; the actual internal name is _idafmax) returns the index of the first occurrence in the vector **A** of the element which has the greatest absolute value. The rest of the code is cleanup code that resets MAX and INDEX only when the indexed value is greater than the original value of MAX.

The following special index routines are in the run-time library, and patterns similar to that above are recognized by the Stardent 1500/3000 vectorizer and converted to calls to these functions:

```
_idafmax   _idfmax   _iiafmax   _iifmax   _isafmax   _isfmax
_idafmin   _idfmin   _iiafmin   _iifmin   _isafmin   _isfmin
_idalmax   _idlmax   _iialmax   _iilmax   _isalmax   _islmax
_idalmin   _idlmin   _iialmin   _iilmin   _isalmin   _islmin
```

These routines have been named using a convention similar to that in Linpack:

- The first letter is always **i**, indicating that the routines return integer values.

- The second letter indicates the type of array searched: **d** for a double precision array, **s** for a single precision array, and **i** for an integer array.
- If the third letter is **a**, the routine takes the absolute value of all the array elements before searching.
- If the fourth letter is **f**, the routine returns the index of the first element that satisfies the request. This arises from a greater than operation in the Fortran source code for maximum. If the letter is **l**, the routine returns the index of the last element that satisfies this request (the Fortran source code had a greater than equal for maximum).
- If the last three letters are **max**, the routine hunts for a maximum. If they are **min**, it hunts for a minimum.

Thus, the routine **_idfmax** searches a double precision array for the index of the element with the maximum value, and returns the index of the first such element it finds. The routine **_islmin** searches a single precision array for the index of the element with the minimum value, and returns the index of the last such element it finds.

8.3 Compiler Directives

In addition to extensive reporting facilities, the Stardent 1500/3000 Fortran compiler also provides an extensive set of directives that allow you to guide the compilation of the program. The Stardent 1500/3000 directives have been designed to look like comments to other compilers, and the Stardent 1500/3000 compiler does not attempt to issue any error messages regarding directives (since something that looks similar to a Stardent 1500/3000 directive may be a comment). As a result, it is often useful to examine the vector report, where all directives found are listed, to ensure that a typographic error has not caused the compiler to overlook an intended directive. The following sections discuss the formats and functions of the Stardent 1500/3000 Fortran compiler directives.

8.3.1 ASIS

The **ASIS** directive tells the compiler not to optimize the following loop for vector hardware. The vectorizer leaves an **ASIS** loop alone. This directive can be used for loops with very sensitive numeric properties. Note that this directive orders the vectorizer

to leave all statements in the following loop alone, and not change them in any way. Thus, it inhibits vectorization of all loops outside of the controlled loop, as well as those inside the loop.

SYNTAX

C$DOIT ASIS

8.3.2 INLINE

The **INLINE** directive tells the compiler to inline the named functions in the following loops, if possible.

SYNTAX

C$DOIT INLINE *function1, function2, ...*

where

function1 is the function to be inlined. The named function must be in the current source file.

8.3.3 IVDEP

The **IVDEP** directive tells the compiler to ignore any array dependencies. Any defined scalars are located either in a straightforward recurrence or need to be expanded into temporaries.

SYNTAX

C$DOIT IVDEP

8.3.4 IPDEP

The **IPDEP** directive is the analog to the **IVDEP** directive, and tells the compiler to ignore the dependencies that appear to inhibit parallelization in the following loop. Any defined scalars are assumed to be temporaries local to each processor.

SYNTAX

C$DOIT IPDEP

The **PPROC** is a dual-usage directive. The two examples that follow illustrate how it can be used.

(1) In code to be compiled at optimization level 03, the directive **PPROC** assures to the compiler that a subprogram has been compiled for parallel execution. For example,

```
C$DOIT PPROC DOEM
       DO J = 1, 6000
       CALL DOEM(A,B,C,J,K)
       END DO
```

This directive tells the compiler that **DOEM** has been compiled safely for parallel execution. As a result, the compiler parallelizes the **DO** loop around **DOEM**.

(1) If the procedure named in the **PPROC** directive is the same as the procedure being compiled, the compiler interprets the directive as requesting that the procedure be compiled safely for parallel execution. As a result, reentrant code is generated for the procedure, just as through the compiler option **-safe=procs** had been specified. For example,

```
      SUBROUTINE DOEM(M,N,X,Y,Z)
C$DOIT PPROC DOEM
                .
                .
                .
```

In this case, the compiler compiles this subroutine for parallel execution. The dual use of the **PPROC** directive allows you to use one include file across a large number of procedures in many files.

SYNTAX

C$DOIT PPROC *DOEM*

where

DOEM is a procedure name, or a list of procedure names.

8.3.6 THREADLOCAL

The **THREADLOCAL** directive cause the Fortran compiler to move a named common block into threadlocal storage. Using the **THREADLOCAL** directive on a common block has the equivalent effect as Cray **TASK$COMMON** variables.

NOTE
Refer to next section titled *User-controlled Parallelism* for the definition of **Threadlocal.**

SYNTAX

```
      COMMON /X0/ A, B, C
C$DOIT THREADLOCAL X0
```

where

X0 is the threadlocal storage holding variables **A**, **B**, and **C**, and these variables are externally known. *Externally known* means that two different procedures can access the variables.

8.3.7 STATIC

The **STATIC** directive hides the name of the common block so that it is not externally known.

SYNTAX

```
      COMMON /X0/ A, B, C
C$DOIT STATIC X0
```

where

X0 is the static storage holding variables **A**, **B**, and **C**.

8.3.8 VBEST

The **VBEST** directive indicates the best loop to vectorize if the loop is vectorizable. It does **not** mean that the loop can be vectorized, and the compiler ignores it if the loop cannot be vectorized.

SYNTAX

```
C$DOIT VBEST
```

8.3.9 PBEST

The PBEST directive is similar to VBEST. It indicates the best loop to parallelize rather than to vectorize.

SYNTAX

C$DOIT PBEST

8.3.10 VPROC

The VPROC directive tells the compiler that the named function has an assembly language version that can take vector arguments, following the Stardent 1500/3000 vector calling convention. Upon encountering the VPROC directive, the compiler assumes that the named function has no side effects and attempts to vectorize any loop containing a call to that function. If vectorizable, the compiler replaces the function with the vector function name *vfname*, changing all the scalar arguments into vector arguments.

SYNTAX

C$DOIT VPROC *fname, vfname*

where

fname is the function to be vectorized.

vfname is the vectorized function used by the compiler. This function must be written in assembler, and must either be in an Ardent supplied library or must be provided by the user.

8.3.11 VREPORT

The VREPORT directive invokes the vector reporting facility on the next loop and is used to tell the user what vectorization has been done to the program. VREPORT provides a detailed listing of exactly what the compiler did to each loop nest. This option allows you to get a report listing on one loop without having to get a listing on the entire program.

SYNTAX

C$DOIT VREPORT

8.3.12 NO_PARALLEL

The **NO_PARALLEL** directive tells the compiler not to run the next loop in parallel.

SYNTAX

C$DOIT NO_PARALLEL

8.3.13 NO_VECTOR

The **NO_VECTOR** directive tells the compiler not vectorize the following loop. This directive is very useful at optimization level 03 when you desire parallel code but no vector code.

SYNTAX

C$DOIT NO_VECTOR

8.3.14 OPT_LEVEL

The **OPT_LEVEL** directive tells the compiler to set the optimization level for a specific procedure within a file.

SYNTAX

C$DOIT OPT_LEVEL *N*

where

N is the optimization level numbered 0, 1, 2, or 3. When compiling a procedure which contains or is preceeded by this directive, the compiler resets the optimization level for that procedure, regardless of the optimization level with which the compiler was invoked.

8.3.15 SCALAR

The **SCALAR** directive tells the compiler to run the next loop in scalar. It does not allow vectorization or parallelization on the loop. This differs from the **ASIS** directive in that the compiler performs all optimizations on the next loop; it simply does not vectorize or parallelize that loop. Also, the **SCALAR** directive has no effect on any other loops, unlike the **ASIS** directive.

SYNTAX

C$DOIT SCALAR

8.3.16 Cray Directives

Directives for the Cray compiler have existed for roughly a decade. Because their functionality is very similar to Stardent 1500/3000's directives, the Stardent 1500/3000 Fortran compiler accepts Cray directives. The following directives have been added to the Stardent 1500/3000 Fortran compiler for compatibility with Cray directives.

8.3.16.1 IVDEP

The **IVDEP** directive is a Cray directive specifying that the compiler should ignore vector dependencies. This directive does not guarantee vector execution.

SYNTAX

CDIR$ IVDEP

8.3.16.2 NORECURRENCE

The **NORECURRENCE** directive is a Cray directive inhibiting generation of vector recurrence code for loops that are below a given size *N*. It is similar to the **NOVECTOR** mode.

SYNTAX

CDIR$ NORECURRENCE = *N*

8.3.16.3 NOVECTOR

The **NOVECTOR** directive is a Cray directive inhibiting vectorization of loops below a given size *N*.

SYNTAX

CDIR$ NOVECTOR = *N*

8.3.16.4 VECTOR

The **VECTOR** directive is a Cray directive toggling **NORECURRENCE** and **NOVECTOR** between default and specified values.

SYNTAX

CDIR$ VECTOR

8.3.17 Inline Expansion

When the Stardent 1500/3000 compiler cannot gather enough information to determine whether a loop can safely vectorize or parallelize, it assumes the worst possible case and processes the loop sequentially. Procedure calls, by their data abstraction, hide vital information from the compiler. They cannot be executed on a vector unit. The following loop

```
      DO 120 I = 1, N
         A(I) = HYPOT(B(I), C(I))
 120  CONTINUE
         .
         .
         .
      FUNCTION HYPOT(A,B)
         HYPOT = A*A + B*B
      RETURN
```

does not vectorize because of the lack of information about the effects of the call and because of the procedure call itself. The Stardent 1500/3000 compiler contains options which allow you to cause procedures such as this to be substituted *inline* automatically. For instance, when the Stardent 1500/3000 compiler is told to inline **HYPOT**, the resulting vreport is as follows:

```
DO iv=1, N, 32
   rv = MIN(N, 31 + iv)
   vl = rv - iv + 1
   DO VECTOR I=iv, rv
      tv_in_a_8(2 + I - iv) = B(I)
      tv_in_b_7(2 + I - iv) = C(I)
      tv_return_type_6(2 + I - iv) = tv_in_a_8(2 + I- iv) * tv_in_a_8(2 + I - iv) +
                                     tv_in_b_7(2 + I - iv) * tv_in_b_7(2 + I - iv)
      A(I) = tv_return_type_6(2 + I - iv)
   END DO
   IF (rv .EQ. N) THEN
      Return_Type_6 = tv_return_type_6(2 + rv - iv)
   END IF
END DO
```

The report is a little difficult to read, because the compiler creates a number of temporaries to handle the inlining process correctly (all variables which have **in_** in them were created for inlining), but the loop has now vectorized, and runs much faster than with the procedure call in it. This optimization allows you to use lots of small functions (which generally create code that is easier to read and maintain) but still get the efficiency of having large procedures.

There are two ways to invoke the inlining facilities of the Stardent 1500/3000 compiler:

(1) In the simplest form, you merely need to add the option **-inline** to the command line. When inlining is enabled, the Stardent 1500/3000 automatically inlines any functions found in the same source file that appear to be good candidates for inlining. "Good candidates" include very short functions which have no other function calls or input/output statements as well as functions specified in **INLINE** directives.

(2) A mechanism that gives the user more control over the inlining process is the use of an inline catalog. Using this mechanism, you can create a database or catalog of functions that you wish to inline frequently. Thereafter, you can refer to that catalog in a **-Npaths=** option on the command line, and the compiler automatically inlines all functions found in that catalog.

The **HYPOT** example above illustrates the first type of inlining. If **HYPOT** and the main function are both in the same file, compiling with **-inline** is enough to automatically inline **HYPOT**. To use the same example to illustrate the second method, assume that the function **HYPOT** is in a file *hypot.f* and the main procedure is in a file *main.f*. The first step is to put **HYPOT** into a catalog, which is accomplished by using the **-catalog** option on the compiler. By default, the suffix for inline catalogs is assumed to be ".in", so *hypot.in* is a reasonable name for the catalog. The following command creates the catalog:

```
fc -c -catalog=hypot.in hypot.f
```

Note that this command actually creates two files, *hypot.in* and *hypot.id*. In general, every ".in" file has an associated ".id" file; neither of these files works without the other. At this point, **HYPOT** can be inlined in any file by compiling that file with the **-inline** option and specifying *hypot.in* on the **-Npaths** option:

```
fc -c -inline -Npaths=hypot.in main.f
```

The compiler prints out warning messages whenever it inlines files, letting you know the origination of the inlined file.

There are a number of important points regarding inline catalogs:

(1) It generally does not matter what optimization level is used in the creation of the catalog, because the inlining process occurs early in a compilation, long before any other optimizations are done. As a result, inlined code gets fully optimized at the time it is inlined; there is little value in optimizing it twice.

(2) The **-catalog** option adds to an existing catalog as well as create a new one. When inlining, the compiler takes that **last** entry when multiple entries exist for a routine.

(3) Both files associated with a catalog must be present for the catalog to work. If either is missing, the compiler does not inline. Also, notice that the use of the two characters "in" and the 14 character filename limited in System V may conflict if long file names are used, causing the ".in" and ".id" files to overwrite themselves.

(4) Most of the time, the object file created during the creation of a catalog can be thrown away. However, in a few instances, it cannot. If the function to be inlined contains initialized data or a static variable whose value is to be retained from call to call, the compiler generates an external name for that variable, and uses it in the inlined code. In such cases, the object file used to create the catalog must be linked at load time, because it contains the definition for that variable. To illustrate with a concrete example, consider the following sample random number generator:

```
function rand()
data seed /314159/
seed = seed * 27818
rand = seed
return
end
```

For this case, since you want the variable **seed** modified at every call, regardless of whether it is inlined or not, the compiler creates an external variable with a unique name to replace it, both in the inlined version and in the object file. The name of this external variable is based on the procedure

name in order to avoid the possibility of accidentally creating the same variable in two different files. For instance, the name created for seed above is

```
$$IE_RAND_$$IE__lcl_Block_SEED_$$1
```

The definition of this variable is in the object file created with the catalog. As a result, if **RAND** is inlined from this catalog, the resulting object file does not link without the *rand.o* file created in the catalog step because **$$IE_RAND_$$IE__lcl_Block_SEED_$$1** is undefined.

(1) Whenever you receive new compiler versions, it may be necessary to recreate your catalogs using the new compiler. Occassional changes to the intermediate representation used in the compiler may invalidate existing catalogs. Note that the compiler clearly warns you when this occurs; it does not inline from an existing catalog if that catalog is out of date.

(2) By default, the **-inline** option uses the supplied library of inline functions which located in */usr/lib/libbF77.in*. This default can be overridden by specifying a blank Npaths, that is **-Npaths=**, which acts the same way as the **-I** option acts with the cpp. A list of the functions provided in the the system inline catalog is provided in the following section, called *Inline Functions*, in this chapter.

(3) As long as the calling conventions are respected, Fortran functions can be correctly inlined in C and C functions can be correctly inlined in Fortran.

(4) There are a small number of function types that cannot be inlined. The compiler does not inline character valued functions, functions that have Fortran character strings as parameters, or C vararg functions. It also does not inline a function into the while condition of a **WHILE** loop.

8.4 Inline Functions

The best functions to inline are small functions that are often used in loops. The Stardent 1500/3000 compiling system includes a sample library of inline functions that illustrate both the utility and the power of inlining in the Stardent 1500/3000 compiler. This library is comprised of the BLAS (Basic Linear Algebra Subroutines), a set of vector linear algebra routines.

These functions are automatically candidates for inlining whenever the **-inline** option is used. The following table lists all the routines in the BLAS inline library.

NOTE
The inline functions library is located in */usr/lib/libbF77.in*, and also in compiled form in */usr/lib/libbF77.a*.

Table 8-1. BLAS Function Names

caxpy	dcopy	isamax	sscal
ccopy	ddot	izamax	sswap
cdotc	dmach*	sasum	zaxpy
cdotu	dnrm2	saxpy	zcopy
cmach*	drot	scasum	zdotc
crotg	drotg	scnrm2	zdotu
cscal	dscal	scopy	zdrot
csrot	dswap	sdot	zdscal
csscal	dzasum	smach*	zmach*
cswap	dznrm2	snrm2	zrotg
dasum	icamax	srot	zscal
daxpy	idamax	srotg	zswap

The first letter of any function indicates the data type it works on: "i" indicates integer, "s" indicates single precision, "d" indicates double precision, "c" indicates complex, and "z" indicates double complex. The rest of the name indicates the function provided. Following is an argument description of the single precision routines; the analagous routines in other precisions have the same arguments (but with different types of course) and perform the same function. If you want additional information concerning these inline functions (e.g. how they are derived, etc.), please refer to the *LINPACK User's Guide*.

* This function is not required by Linpack proper. It is the Stardent 1500/3000 function proper used in testing only.

8.4.1 isamax

integer function isamax(n,x,incx)
integer n,incx
real x(1)

isamax function finds the element with largest absolute value in the vector. If n ≤ 0, the function sets the result to zero and returns immediately. **isamax** is a real type function.

8.4.2 sasum

real function sasum(n,x,incx)
integer n,incx
real x(1)

sasum function computes the sum of magnitudes of vector components. This function takes summation only on the absolute values of the vector components when n > 0. If n ≤ 0, the function returns immediately. **sasum** is a real type function.

8.4.3 saxpy

subroutine saxpy(n,a,x,incx,y,incy)
integer n,incx,incy
real x(1),y(1),a

saxpy subroutine performs an elementary vector operation such as **y = ax + y**. If a = 0 or if n ≤ 0, this subroutine returns immediately. **saxpy** is a real type subroutine.

8.4.4 scopy

subroutine scopy(n,x,incx,y,incy)
integer n,incx,incy
real x(1),y(1)

scopy subroutine copies a vector x component into y component. If n ≤ 0, the subroutine returns immediately. **scopy** is a real type subroutine.

8.4.5
sdot

subroutine sdot(n,x,incx,y,incy)
integer n,incx,incy
real x(1),y(1)

sdot function computes the dot product of a vector. If n ≤ 0, the result of the dot product is zero. **sdot** is a real type function.

8.4.6
smach

real function smach(job)
integer job

where

job is 1, 2, or 3 for *epsilon, tiny, or huge,* respectively.
If *epsilon* then return value is 1.19209e-07
If *tiny* then return value is 9.8608e-30
If *huge* then return value is 4.0565e+29

smach computes machine parameters of floating point arithmetic for use in testing only. It is not required by Linpack proper. **smach** is a real type function.

8.4.7
snrm2

real function snrm2(n,x,incx)
integer n,incx
real x(1)

snrm2 function computes the 2-Norm (Euclidean Length) of a vector. This function uses only the absolute value of the vector component to do its computation when n > 0. If n ≤ 0, the function returns immediately. **snrm2** is a real type function.

8.4.8
srot

subroutine srot(n,x,incx,y,incy,c,s)
integer n,incx,incy
real x(1),y(1),c,s

srot subroutine applies a plane rotation on vector components. The subroutine returns immediately if n ≤ 0 or if c = 1 and s = 0. **srot** is a real type subroutine.

8.4.9 srotg

subroutine srotg(a,b,c,s)
real a,b,c,s

srotg subroutine constructs Givens Plane Rotation. **srotg** is a real type function.

8.4.10 sscal

subroutine sscal(n,a,x,incx)
integer n,incx
real a,x(1)

sscal subroutine performs a vector scaling (multiply a vector with a constant). The subroutine returns immediately if n ≤ 0. **sscal** is a real type subroutine.

8.4.11 sswap

subroutine sswap(n,x,incx,y,incy)
integer n,incx,incy
complex x(1),y(1)

sswap subroutine interchanges vector x components with y components. If n ≤ 0, the subroutine returns immediately. **sswap** is a real type subroutine.

8.5 User-Controlled Parallelism

For many programs, the automatic parallelism available at the DO loop level is sufficient to give excellent speed increases. However, there are a number of codes which can benefit from a higher level of parallelism, running user subroutines in parallel. The Stardent 1500/3000 compiler normally does not attempt to parallelize a loop which contains a user-supplied subroutine, because it is very difficult for the compiler to determine the side effects of a user call. The compiler does contain a set of directives and options that permit you to convey that a procedure call is safe, thereby allowing the compiler to parallelize such calls. This section is intended to give you an introduction to parallelism on the Stardent 1500/3000. If you need further information or detailed examples, please refer the *Programmer's Guide*, Chapter 8, *Explicit Parallel Programming.*

A critical issue in user-controlled parallelism is storage allocation. There are three different kinds of storage available to parallel programs: static storage, stack storage, and threadlocal storage.

8.5.1 Static Storage

Static storage is storage that has one definite physical location in memory, and is addressed by the compiler during compile-time. When a value is placed in a variable in static storage, that variable retains that value until a new value is stored into it, regardless of any procedure calls made in the process. By default, the Stardent 1500/3000 compiler allocates all local variables in Fortran to static storage. As a result, user procedures are usually not reentrant.

8.5.2 Stack Storage

A variable in stack storage does not have a definite physical location in memory, but instead is accessed as an offset from the *stack pointer*, which is a register maintained by all procedures. Whenever a procedure call is made, the stack pointer is incremented by the amount of stack storage used by the procedure. When a procedure returns, it decrements the stack pointer by that amount. Since stack variables are addressed by an offset from the stack pointer, their actual addresses in memory can vary during execution. This means that stack variables can retain the same memory location during one procedure execution, but if that procedure returns and is called again, there is no guarantee that the stack pointer points to the same place. As a result, stack variables do not retain values across a procedure return. One of the optimizations performed by the Fortran compiler is movement of Fortran local variables from static storage to stack storage. It does this only when it can definitely determine that a variable does not need to retain its value across a procedure return.

8.5.3 Threadlocal Storage

Threadlocal storage is static storage with one important difference. When two different processors are working on the same program, and they access a static variable, they get the same memory address. When two different processors access a threadlocal variable, they get different addresses. Threadlocal storage is different from stack storage because any time one specific processor accesses a threadlocal variable, it gets the same memory location. In other words, threadlocal storage is storage that is **local** to each processor; each processor has its own private copy of the variable.

8.5.4 Parallelization in Stardent 1500/3000 Fortran

Since the key goal of user-controlled parallelism is to get different instances of the same procedure on different processors, controlling the allocation of storage is critical. Fortran locals go into static storage by default, and running copies of the same procedure in parallel causes all sorts of confusion because different processors are accessing the same memory location in random order. To run a Fortran procedure in parallel and have reproducible results, you must arrange for variables to be allocated either in threadlocal storage or in stack storage. Through the use of the microtasking library (discussed at the end of this section) each processor can have its own stack.

NOTE
Refer to later section titled *Compiler Directives*, for additional information on **PPROC**, **STATIC**, and **THREADLOCAL** directives. For the **-safe=procs** compiler option, refer to Chapter 7, *Compiler Options*.

Thus, the first step in converting a procedure to be executed in parallel is to move its local variables from static storage to either stack storage or threadlocal storage. Most of the time you want local variables to go to stack storage, with only a few special cases on threadlocal storage. The Stardent 1500/3000 Fortran compiler has a compiler option for just this case. Invoking **-safe=procs** on the compile line causes a procedure to be compiled for parallel execution. This option causes all local variables (except those that are equivalenced, members of namelist groups, or initialized by data statements) to go into stack storage rather than static storage An alternative way of obtaining the same result is to insert a **PPROC** directive inside the procedure stating that it should be compiled for parallel execution.

In a procedure that has been compiled for parallel execution, COMMON blocks are the mechanism to get variables into static and threadlocal storage. Normal COMMON blocks in Fortran go into static storage, whose address is known externally (externally known means that at least two procedures can access the assigned variable).

The Stardent 1500/3000 Fortran compiler provides two directives for creating threadlocal storage and static storage that are not externally known.

The **THREADLOCAL** directive causes the Fortran compiler to move a named common block into threadlocal storage. Thus, the following sequence:

```
      COMMOM /X0/ A, B, C
C$DOIT THREADLOCAL X0
```

causes the variables **A**, **B**, and **C** to be placed in threadlocal storage and also to be externally known. Each processor has its own copies of **A**, **B**, and **C**, and procedures running on the same processor that have the same declarations can communicate values through them. There are two important points to note about threadlocal variables:

(1) They cannot be initialized in block data statements. Threadlocal storage is not created until the program goes parallel; thus, static initializations are useless. Threadlocal variables have an initial value of zero.

(2) They cannot be correctly assigned outside of parallel regions because they do not come into existence until the program goes parallel. While it would be nice to have all the copies inherit values from a global location, it is not possible in the Stardent 1500/3000 microtasking scheme.

The **STATIC** directive hides the name of a common block, so that it is not externally known. To illustrate, consider procedure **X** having the declaration

```
      COMMON /X0/ A, B, C
C$DOIT STATIC X0
```

and procedure **Y** has the declaration

```
      COMMON /X0/ A, B, C
```

Without the **STATIC** directive, storing into **A** in procedure **X** would modify the value of **A** in **Y**. With the **STATIC** directive, the **A** in **X** is completely different from that in **Y**; **X0** becomes a local common block within **X**. In **X**, **X0** is still in static storage, so that if procedure **X** is run in parallel, different invocations may well interfere with each other if they are all writing to variables in **X0**.

The major portion of microtasking is getting the storage allocation correct. Once that has been done, the compiler automatically parallelizes a loop containing a procedure call if that procedure is listed in a **PPROC** directive (although an **IPDEP** directive may also be necessary in some cases). There are a number of other support functions that may prove useful.

NOTE
STATIC and THREADLOCAL directives can be both applied to a single COMMON block, so it is possible to create a threadlocal common block not visible outside of the procedure.

8.5.5 Microtasking Library

There are a number of support functions to help in microtasking. Their descriptions and formats are discussed in the following section.

8.5.5.1 MT_INIT

SYNTAX

CALL MT_INIT(*stack_size*)

where

stack_size is the number of bytes required to hold a parallel procedure.

NOTE
If you need to know more about how to use these microtasking library functions, please refer to the *Programmer's Guide* in the chapter titled *Explicit Parallel Programming*. There are several detailed examples on how to use these functions for parallel programming.

DESCRIPTION

This procedure must be called before invoking a parallel function. It initializes separate stacks for all the threads, creating a stack of *stack_size* bytes for each parallel procedure. If this procedure is not called ahead of time, or if it is unable to create stacks for all processors, the program terminates at the point of the parallel call with an appropriate message. Guessing the appropriate *stack_size* depends upon the depth of the calling chain and is sometimes difficult. When in doubt, use about 10,000 bytes for most shallow applications, and about 100,000 bytes for anything that does not have lots of local arrays. (See *Special Notes on Microtasking* at the end of this section.)

8.5.5.2 MT_FINI

SYNTAX

CALL MT_FINI()

DESCRIPTION

This procedure frees up the parallel stacks created by MT_INIT. However, you do not really need to call this procedure because the deallocation of memory is done automatically when a program is ended.

8.5.5.3 MT_LOCK

SYNTAX

CALL MT_LOCK(*semaphore*)

where

semaphore is a data item which, when accessed through system functions, provides exclusive ownership to the accessing function (in this case, MT_LOCK).

DESCRIPTION

This routine waits for the passed-in semaphore until the value of the semaphore becomes 1. At that point, the semaphore is set to zero, and the routine returns. Any integer variable can be a semaphore. It should be initialized to 1.

8.5.5.4 MT_UNLOCK

SYNTAX

CALL MT_UNLOCK(*semaphore*)

where

semaphore is a data item that is eventually unlocked by MT_UNLOCK.

DESCRIPTION

This routine unlocks the semaphore by atomically setting it to a value of 1.

8.5.5.5 MT_NUMBER_OF_PROCS

SYNTAX

N = MT_NUMBER_OF_PROCS( )

where

N is the number of physical processors in the machine returned by function MT_NUMBER_OF_PROCS.

DESCRIPTION

This function returns the number of physical processors in the machine.

8.5.5.6 MT_SERIAL

SYNTAX

CALL MT_SERIAL()

DESCRIPTION

If the file is opened in a serial section, all the threads can have access to that file. However, if the file is opened after the program goes parallel (or after a loop goes parallel), only the thread that opened that file can have access to it. Thus, for users who absolutely need to open files after going parallel and access them from different threads, the microtasking function **MT_SERIAL** allows the program to return back to its serial execution. (See *Special Notes on Microtasking* at the end of this section.)

8.5.5.7 MT_SET_THREAD_NUMBER

SYNTAX

CALL MT_SET_THREAD_NUMBER(*no_of_threads*)

where

no_of_threads is the number of threads that a parallel process uses. In other words, it is the same as the number of processors running on your Stardent 1500/3000 machine.

DESCRIPTION

This routine allows you to specify the number of processors you want to use to run on the Stardent 1500/3000. You do not have to call this routine unless you want to use fewer processors than your Stardent 1500/3000 has available. If you want to use fewer processors, you must call this routine before going parallel. Once the program has gone parallel, the number of processors remains fixed. Since MT_INIT goes parallel to initialize the stacks the moment it is called, MT_SET_THREAD_NUMBER must be called before MT_INIT. *no_of_threads* should be an integer whose value is greater than zero and no larger than 4.

8.5.5.8 MT_SET_THREAD_PROCS

SYNTAX

CALL MT_SET_THREAD_PROCS(*no_of_threads, procs*)

where

no_of_threads is the number of threads specifically set to run on *procs*.

procs is the selected processor which run the number of threads specified in *no_of_threads*.

DESCRIPTION

This procedure sets the number of threads that are used, just like MT_SET_THREAD_NUMBER, but then causes threads to run on the listed processors. Thus, if you would like three threads, two

running on processor 1 and 1 running on processor 2, you can use this call. *procs* is a 4 element integer array, storing the processor number to use for each thread. Only the first *no_of_threads* entries are examined.

8.5.5.9 Special Notes on Microtasking

When using microtasking in Fortran to accessing/writing files, some care must be taken with regard to the timing of the opening of the file. If the program goes parallel, then opens a file, only the thread that opened the file has access to it; all other threads do not have access to that file.

For example, if you open a file after calling **MT_INIT** (all microtasking functions are described in this section) or after a loop that goes parallel, you may get unexpected results. You may think that you can access that file from all threads, but that is not true. If the file is opened in a serial section, then all the threads eventually inherit it naturally, which is the behavior that is expected. If you absolutely need to open files after going parallel and then access them from different threads, **MT_SERIAL** is the microtasking function that allows you to go back to serial execution.

8.6 Asynchronous I/O

A sequence of input/output in a program is to read data, execute on data, and then to write data. Normally under the UNIX operating system, as soon as a write has been started, then other pending computations may continue. In the case of a read, a read does not return until the data is actually transferred into memory from the disk. To summarize, writing and processing are overlapped and reading and processing are not overlapped.

8.6.1 Definition

Asynchronous I/O enables reading to be overlapped with processing and writing. For example, when the first input is read in and completed, the read for the second input begins. The processing for the first input now starts followed by the write for the first input. This is then followed by the read for the third input, the processing for the second input, and so on. This cycle continues, always reading one input ahead. The major usage advantage with asynchronous I/O is that you must check to see if the I/O is completed before using the data.

There are benefits to using asynchronous I/O. The primary benefit when using asynchronous I/O is that the CPU is used while disk transfer is occurring. A secondary benefit is that the data is transferred directly from disk to data storage with no intervening buffering. Therefore, there is one less copy of the data and correspondingly less CPU effort required.

8.6.2 Asynchronous Library Functions for Fortran

There are three Fortran library functions that are available for asynchronous I/O. They can be used to perform an asynchronous read, obtain the status of the file read, and wait for the read to complete. The following section provides the syntax of the three asynchronous I/O library functions.

8.6.2.1 ABLOCK

ABLOCK – wait for asynchronous read to complete

SYNOPSIS

```
EXTERNAL ABLOCK
INTEGER ABLOCK
INTEGER UNIT
INTEGER CODE
CODE = ABLOCK(UNIT)
```

DESCRIPTION

UNIT is the unit number used in an **OPEN** statement. The **OPEN** on this unit must have had *ACCESS='ASYNC'*, or it is an error. **ABLOCK** suspends execution of other process until no asynchronous file read is pending on the specified *UNIT*.

ABLOCK returns the count of bytes read. This count should be the value of *NBYTES* in the call to **AREAD**. *CODE* is the actual number of bytes being read.

8.6.2.2 AREAD

AREAD – asynchronous read

SYNOPSIS

```
EXTERNAL AREAD
CHARACTER BUFF*1(4096)
INTEGER UNIT, BLOCK_NO, NBYTES
CALL AREAD(UNIT, BUFF, BLOCK_NO, NBYTES)
```

DESCRIPTION

UNIT is the unit number used in an **OPEN** statement. The **OPEN** on this unit must have had *ACCESS='ASYNC'*, or it is an error.

On devices capable of seeking, **AREAD** attempts to read *NBYTES* asynchronously from the file associated with *UNIT*

at the 4096-byte block boundary specified by *BLOCK_NO* into the buffer *BUFF*. The block number specified in *BLOCK_NO* should start at 1, and each block has a size of 4096 bytes, independent of the record length specified in the **OPEN** statement.

AREAD returns immediately after the read request has been issued to the appropriate device. Upon return, completion of the read request is not guaranteed. No assumption should be made regarding the amount of data. You can be guaranteed that data transfer is completed only by using **ABLOCK** to wait for **AREAD** to complete.

The offset into the file must be exactly on the 4096-byte boundaries, the amount of the read request must be for multiples of 4096 bytes, and the user buffer *BUFF* must be on a 4-byte (word) boundary.

8.6.2.3 ASTATUS

ASTATUS – check status for asynchronous read

SYNOPSIS

```
EXTERNAL ASTATUS
INTEGER ASTATUS
INTEGER UNIT
LOGICAL AST
AST = ASTATUS(UNIT)
```

DESCRIPTION

UNIT is the unit number used in an **OPEN** statement. The **OPEN** on this unit must have had *ACCESS='ASYNC'*, or it is an error. **ASTATUS** returns the status of a pending asynchronous file read operation. The value of .TRUE. is returned if the read is completed, and .FALSE. is returned if it is still pending.

Asynchronous I/O Example

The following section illustrates a very basic example of how you can use **ABLOCK**, **AREAD**, and **ASTATUS**.

EXAMPLE

```
      PROGRAM MAIN
      CHARACTER*1 BUFFER(4096)
      EXTERNAL ABLOCK, AREAD, ASTATUS
      LOGICAL ASTATUS, DONE
      INTEGER ABLOCK, AREAD
      INTEGER STATUS

C     Open the file with asynchronous access.

      OPEN(2, FILE='DATA', ACCESS='ASYNC')
      CALL AREAD (2, BUFFER, 1, 4096)
      DONE = ASTATUS(2)
      IF (DONE) THEN
          PRINT *, 'READ FINISHED SUCCESSFULLY'
      ELSE
          PRINT *, 'READ TRANSFER NOT DONE, GOING TO ABLOCK'
          STATUS = ABLOCK(2)
          PRINT *, 'ABLOCK READ', STATUS
      ENDIF

C     buffer is now filled with data; however, if ABLOCK returns
C     a value less than 4096 bytes, then the buffer is not filled.

      END
```

PROFILING PROGRAMS

CHAPTER NINE

9.1 Profiling Programs

To speed up a program, optimize those sections of code in which your program spends the most time. You can determine where the time is spent in the code by creating an *execution profile* of your program.

you can specify certain options when invoking the Stardent 1500/3000 compiler to allow a profiler program to create a listing showing relative time spent in various parts of your program. These compile-time options are **-ploop**, and **-p**. You can use these options to select different forms of execution profiling.

Note that the compiler does not actually create the execution profile or listing. It simply facilitates the profiling process by inserting additional labels in your code. These labels are needed by the actual profiler program to perform its timing measurements. A separate program, *mkprof* modifies the object code that the compiler creates. When your object file is run, your program writes a special output file that can be interpreted by yet another program called *prof*.

Thus, the process of profiling a program is a four step process:

- Compile your program with the **-ploop** or **-p** option (**-ploop** generates a more thorough profile than **-p**). Example: **fc -ploop sample.f**. A successful compile creates an object code file named *a.out*.

- Run *mkprof* on the object code to create a modified output file that contains the code to produce a file called *mon.out*. Example: **mkprof a.out mysample.out** where *mkprof* is the command, *a.out* is the file to modify, and *mysample.out* is the name of the modified version of *a.out* that now contains code to produce *mon.out*. The timing information in *mon.out* can later be interpreted by the *prof* program.

- Run the executable file. Example: **mysample.out**. In the current directory a new file named *mon.out* is created.
- Now run *prof*, specifying the name of the executable file that you produced using the *mkprof* command. Example: **prof mysample.out**. This prints a listing of statistics from the program.

This chapter illustrates the use of the options **-ploop** and **-p** by showing how they are used to profile an actual program. Additionally, we also show how modifying a program changes the execution profile as an example of the possible uses of this technique.

Note that although the examples concentrate on Stardent 1500/3000 Fortran, the profiling options **-ploop** and **-p** are also available to the C programmer.

9.1.1 -ploop Option

The **-ploop** compiler option allows loops within a single routine to be profiled separately, and may be invoked from Fortran. This can help you to determine which loops dominate the execution time of a routine.

When this option is invoked at compile time, the compiler inserts labels into the generated code before and after all loop nests. The labels have the form

```
_$lp_InLoop_ . . .

and

_$lp_AfterLoop_ . . .
```

where the . . . indicates a label that identifies the source routine and the line number of the loop within that routine.

Use of this option causes the body of the loop to appear as a function, **_$lp_ InLoop_ . . .**, and the code following the loop to appear as **_$lp_AfterLoop _ . . .**.

EXAMPLE

Consider the following Fortran program:

```
(the program is named zzz.f)

      program main
      double precision a(100,100), b(100,100)
      data n/100/, m/100/
      data b/10000 * 1.0d0/
      if (m .eq. n) then
         do i = 1, n
            do j = 1, n
               a(i,j) = a(i,j) * b(i,j)
            enddo
         enddo
      else
         do i = 1, n
            do j = 1, m
               a(i,j) = a(i,j) / b(i,j)
            enddo
         enddo
      endif
      if (m .ne. n) then
         do i = 1, n
            do j = 1, n
               a(i,j) = a(i,j) * b(i,j)
            enddo
         enddo
      else
         do i = 1, n
            do j = 1, m
               a(i,j) = a(i,j) / b(i,j)
            enddo
         enddo
      endif
      end
```

The following four commands were issued on the command line to compile the program, create the executable for profiling, execute the program and create the profile file, and run the profiler.

(1) fc -O2 -ploop zzz.f
(2) mkprof a.out zzz.out
(3) zzz.out
(4) prof -v zzz.out

The preceeding statements are explained as follows:

1. *fc -02 -ploop zzz.f*

   Compile the Fortran file *zzz.f* at optimization level **-02** and profile loops within each routine. The optimization level can

be **-O3** (or any other) if you wish. The output from this compilation is *a.out*. Remember that *a.out* is the default output file; any previous *a.out* file is over-written.

2. *mkprof a.out zzz.out*

   Cause the *mkprof* to profile the object file *a.out* (obtained from step #1) and store the result in *zzz.out*. You may choose any name, however, ".out" must be appended to the end of any name you pick. For the sake of consistency, you should consider naming it the same as your source program.

3. *zzz.out*

   Run the executable file *zzz.out*. This run creates a default file called *mon.out* which contains data needed for the command *prof*. If you forget to do this step, the file *mon.out* is not created, thus causing an error message when you perform step #4.

4. *prof -v zzz.out*

   Cause the command *prof* to output the statistics of profiling. When **-v** is used, you request a verbose output. This verbose output adds two additional columns to the profiling printout: *the percentage of the total time*, and *the average time per call*. Without this option, these two items are not provided.

Profiling Output

Following is the output produced by the *prof* command on *zzz.out* without the **-v** option.

```
count    microsec.  NAME (1 proc.)
    1        20481  _$1p_InLoop_main_program_25
    1         1879  _$1p_InLoop_main_program_6
   31          422  sigset
    1          400  _IO_Initialize_FORTRAN_IO
    3          367  malloc
    3          281  _IO_Close
    3          190  _IO_Allocate_Unit
    3          132  calloc
    3           92  _IO_Can_This_File_Seek
    6           89  _IO_Find_Unit
    1           88  _IO_Finish_FORTRAN_IO
    3           83  ioctl
    3           76  fstat
    4           73  _IO_Unit_Walk
    2           64  sbrk
```

```
3          59  _IO_Change_Unit_Number
3          59  isatty
2          41  fflush
1          33  _IO_Initialize_Unit_Walk
1          29  _start
1          17  _fort_program
1          10  _$lp_AfterLoop_main_program_6
1           4  _$lp_AfterLoop_main_program_25
1           0  exit
```

Following is the output produced by the *prof* command on *zzz.out* with the **-v** option.

```
count    microsec.     %    time/call    NAME (1 proc.)
    1        20481  82.0        20481    _$lp_InLoop_main_program_25
    1         1879   7.5         1879    _$lp_InLoop_main_program_6
   31          422   1.7           13    sigset
    1          400   1.6          400    _IO_Initialize_FORTRAN_IO
    3          367   1.5          122    malloc
    3          281   1.1           93    _IO_Close
    3          190   0.8           63    _IO_Allocate_Unit
    3          132   0.5           44    calloc
    3           92   0.4           30    _IO_Can_This_File_Seek
    6           89   0.4           14    _IO_Find_Unit
    1           88   0.4           88    _IO_Finish_FORTRAN_IO
    3           83   0.3           27    ioctl
    3           76   0.3           25    fstat
    4           73   0.3           18    _IO_Unit_Walk
    2           64   0.3           32    sbrk
    3           59   0.2           19    _IO_Change_Unit_Number
    3           59   0.2           19    isatty
    2           41   0.2           20    fflush
    1           33   0.1           33    _IO_Initialize_Unit_Walk
    1           29   0.1           29    _start
    1           17   0.1           17    _fort_program
    1           10   0.0           10    _$lp_AfterLoop_main_program_6
    1            4   0.0            4    _$lp_AfterLoop_main_program_25
    1            0   0.0            0    exit
```

Output Description

The *prof* command produces five columns of information (with the **-v** option).

The information is presented in the order of highest usage. That is, the routines or loops that use processor time are listed in sequence from the most to the least time used. The first column, **count**, represents the number of times the routine is entered. The second column, **microseconds**, is exactly that, the total number of microseconds that were spent in the routine. The third column, %, is the percentage of total time that was spent in the routine. The fourth column, **time/call**, is the number of microseconds it

took to execute each call to the routine. The fifth column is the name of the procedure.

Note that the number of microseconds reported in the profiling summary is not intended to be an exact count. It is simply a reflection of the relative amount of time spent in each loop as compared to any other within the same program. Loops are defined as **DO** and **DO WHILE** statements in Fortran.

9.1.2 Interpreting Profiled Programs

The profile results show that the program spends most of its time (**82%**) in the loop starting at line **25**. It also shows that the program spends only **7.2%** of its time in the loop at line **6**. The sequential code between the loop starting at line 6 and the one at line 25 takes almost no time. What can you do next to decrease the execution time of your code? One way is to modify the loop that takes the most time. Obviously, the loop at line 25 is the best candidate for modification.

CAUTION
Please note that the modification in the following program does not express or imply a better way of writing a Fortran program, instead, it is done just to show you that you can use the information in the statistics to modify your code accordingly.

To illustrate, two **DO** loops at lines 6 and 25 in the program *zzz.f* (from the previous example) are modified. In each loop, the number of iterations is **n/2** instead of **n**, but the loop is re-entered again (see the program provided below).

EXAMPLE

```
(The program name is zzz2.f)

      program main
      double precision a(100,100), b(100,100)
      data n/100/, m/100/
      data b/10000 * 1.0d0/
      if (m .eq. n) then
         do i = 1, n/2
            do j = 1, n
               a(i,j) = a(i,j) * b(i,j)
            enddo
         enddo
         do i = 1, n/2
            do j = 1, n
               a(i,j) = a(i,j) * b(i,j)
            enddo
         enddo
      else
         do i = 1, n
            do j = 1, m
               a(i,j) = a(i,j) / b(i,j)
            enddo
        enddo
```

```
endif
if (m .ne. n) then
   do i = 1, n
      do j = 1, n
         a(i,j) = a(i,j) * b(i,j)
      enddo
   enddo
else
   do i = 1, n/2
      do j = 1, m
         a(i,j) = a(i,j) / b(i,j)
      enddo
   enddo
   do i = 1, n/2
      do j = 1, m
         a(i,j) = a(i,j) / b(i,j)
      enddo
   enddo
endif
end
```

The following four commands are executed on the command line:

(1) fc -O2 -ploop zzz2.f ; **then a.out is produced**
(2) mkprof -s16 a.out zzz2.out ; **then zzz2.out is produced**
(3) zzz2.out ; **then mon.out is produced**
(4) prof zzz2.out ; **then statitics is produced**

NOTE
Refer to previous section for explanations of each step of the four subsequent commands that were issued on the command line.

Profiling Output

Following are the statistics produced by the *prof* with the **-v** command option.

```
count     milisec.     %    time/call    NAME (1 proc.)
    1           12  50.0           12    _$lp_InLoop_main_program_30
    1            8  33.3            8    _$lp_InLoop_main_program_35
    3            4  16.7            1    malloc
    1            0   0.0            0    _$lp_AfterLoop_main_program_6
    1            0   0.0            0    _$lp_InLoop_main_program_11
    1            0   0.0            0    _$lp_AfterLoop_main_program_11
    1            0   0.0            0    _start
    1            0   0.0            0    _$lp_AfterLoop_main_program_30
    1            0   0.0            0    _fort_program
    1            0   0.0            0    _$lp_AfterLoop_main_program_35
    1            0   0.0            0    _IO_Initialize_FORTRAN_IO
    1            0   0.0            0    _IO_Finish_FORTRAN_IO
    6            0   0.0            0    _IO_Find_Unit
    3            0   0.0            0    _IO_Allocate_Unit
    3            0   0.0            0    _IO_Change_Unit_Number
    1            0   0.0            0    _IO_Initialize_Unit_Walk
    4            0   0.0            0    _IO_Unit_Walk
    3            0   0.0            0    _IO_Can_This_File_Seek
    3            0   0.0            0    _IO_Close
    1            0   0.0            0    _$lp_InLoop_main_program_6
    3            0   0.0            0    calloc
```

```
  1          0    0.0          0    exit
 31          0    0.0          0    sigset
  2          0    0.0          0    fflush
  3          0    0.0          0    fstat
  3          0    0.0          0    isatty
  2          0    0.0          0    sbrk
  3          0    0.0          0    ioctl
```

Output Description

Notice that most of the time was spent on the loop at line 30 (50.0%) and line 35 (33.3%). No time was spent in the loop at line 6. The loop at line 6 did not take any time at all in this statistics but took about 7.2% of the time in the previous example. As far as the results are concerned, the time spent in the loop at line 6 has been eliminated. If you continue to attack the loops at line 30 and 35 (was a DO loop at line 25 in the previous example), perhaps you can minimize the time spent in these loops.

9.1.2.1 Other Timing Options for mkprof

The *mkprof* command has two options that you can specify to control the timing intervals for profiling.

-s*N*, the scaling factor where the default for *N* is 4, giving microsecond granularity. Larger numbers allow longer times with less accuracy; each increase of *N* by 1 doubles the maximum range and halves the accuracy. If you are profiling long programs (more than a couple minutes), you might want to use this option. In the above case, since the *N* is 16, the millisecond timing is used.

-T*letter*, the timing option where *letter* can be *f*, *c*, or *e*, in either upper or lower case. If *letter* is *f*, *c*, or *e*, then the FPU time (the default setting), the CPU time, or the elapsed time is used, respectively. CPU time and elapsed time are always measured in ticks (hundredths of a second). The **-s** option is ignored when either **-T***c* or **-T***e* is specified. Please refer to the *Commands Reference Manual* for additional information.

9.1.3 -p Option

Unlike the **-ploop** option which allows loops within a single routine to be profiled separately, the **-p** option forces the Fortran compiler to generate profile code for the overall program. In other words, the **-p** generates profile information at a higher level than **-ploop**.

EXAMPLE

The program *zzz.f* from the previous example is used to illustrate the **-p** option (refer to the previous section for the source code). The following three commands were executed on the command line:

```
(1) fc -O2 -p zzz.f          ; then a.out is produced
(2) a.out                    ; then mon.out is produced
(3) prof -v a.out            ; then statistics is produced
```

The preceeding commands are explained as follows:

1. *fc -02 -p zzz.f*

   Compile the Fortran file *zzz.f* at optimization level -02 and generate the profile information. The output from this compilation is *a.out*. Remember that *a.out* is the default output file; previous *a.out* file is over-written.

2. *a.out*

   Run the executable file *a.out*. This run creates a default file called *mon.out* which contains data needed for the command *prof*. Remember that if you forget to do this step, the file *mon.out* is not created, thus causing an error message when you perform step #3.

3. *prof -v a.out*

   Cause the command *prof* to output the statistics from profiling. The **-v** causes a full printout report. This means that *the percentage of the total time*, and *the average time per call* are also reported. Without this option, these two items are not provided.

Notice that using the **-p** option meant that it was not necessary to run the *mkprof* command on the object file. This is because the **-p** option insert enough information in the object file to produce a correct *mon.out* file with the requested level of granularity on the timing.

Profiling Output

Following is the output produced by the *prof* command on *a.out* with the **-v** option.

```
count    microsec.     %    time/call    NAME (1 proc.)
    1        22355  89.3        22355    _fort_program
   31          503   2.0           16    sigset
    1          485   1.9          485    _IO_Initialize_FORTRAN_IO
    3          399   1.6          133    malloc
    3          200   0.8           66    _IO_Close
    3          140   0.6           46    _IO_Allocate_Unit
    3          133   0.5           44    calloc
    3          106   0.4           35    _IO_Can_This_File_Seek
    1           96   0.4           96    _IO_Finish_FORTRAN_IO
    4           92   0.4           23    _IO_Unit_Walk
    6           91   0.4           15    _IO_Find_Unit
    3           85   0.3           28    fstat
    3           73   0.3           24    ioctl
    2           69   0.3           34    sbrk
    3           56   0.2           18    isatty
    3           44   0.2           14    _IO_Change_Unit_Number
    1           41   0.2           41    _IO_Initialize_Unit_Walk
    2           36   0.1           18    fflush
    1           35   0.1           35    _start
    1            0   0.0            0    exit
```

Output Description

Compared to the statistics when you use the **-ploop** option, this option (**-p**) provides very little information. The above output indicates that it took **89.3%** of the time just to handle all the calls/returns within the program. About **10.7%** of time was used to handle several other things: I/O, memory allocation, initialization, and so on. Since *zzz.f* contains just one subroutine (theoretically speaking), the statistics do not yield as much information as you might expect. However, if your program contains a lot of subroutines, you might want to compile it with the **-p** option. And after you profile the object code with the *prof* command, then you might learn a little more from the statistics about the time spent for each subroutine in your program.

9.1.4 Special Notes

If you are profiling a program in an attempt to optimize it, you will probably use the **-ploop** option much more often than the **-p** option. Since Fortran programs normally consist of **DO** loops, the **-ploop** is a better compiler option to use.

There may be some occasions when you would like to know how the time is spent for each subroutine in the program rather than the time spent in each loop within the subroutine. If this is the case, **-p** is the right option to use to get that information.

USER COMMANDS

CHAPTER TEN

10.1 Fortran User Commands

This chapter describes several user commands that are important in the Fortran programming environment. These are:

- fc – the Fortran compiler. All Fortran compiler options are included, as well as the environment setting variables which allow you to determine how files can be read or written, either in VMS or BSD formats.
- fpr, asa – a filter that transforms files, formatted according to Fortran's carriage control conventions, into files formatted according to UNIX line printer conventions.
- fsplit – a utility program to split a multi-routine Fortran file into individual files.
- ratfor – a conversion program to convert a rational Fortran dialect into an ordinary irrational Fortran.

The following section describes their appropriate syntax and definitions. These commands are also listed in the *Commands Reference Manual*. You can also refer to the *Commands Reference Manual* for other useful user commands.

NAME

fc – Fortran compiler

SYNOPSIS

fc [options] [files] [options] [files]

DESCRIPTION

The *fc* command is an interface to the Stardent 1500/3000 Compilation System. The compilation tools consist of a preprocessor, compiler, assembler, and link editor. The *fc* command processes the supplied options and then executes the various tools with the proper arguments. The *fc* command accepts several types of files as arguments:

Files whose names end with **.f** are taken to be Fortran source programs and may be preprocessed, compiled, optimized, assembled, and link edited. The compilation process may be stopped after the completion of any pass if the appropriate options are supplied. If the compilation process runs through the assembler then an object program is produced and is left in the file whose name is that of the source with **.o** substituted for **.f.** However, the **.o** file is normally deleted if a single Fortran program is compiled and then immediately link edited. In the same way, files whose names end in **.s** are taken to be assembly source programs, and may be assembled and link edited; and files whose names end in **.i** are taken to be preprocessed Fortran source programs and may be compiled, optimized, assembled and link edited. Files whose names do not end in **.f, .s** or **.i** are handed to the link editor.

Since the *fc* command usually creates files in the current directory during the compilation process, it is necessary to run the *fc* command in a directory in which a file can be created.

The following options are interpreted by *fc*:

–all_doubles
Set all undeclared floating point variables, all declared floating point variables, and floating point constants to be as if they were declared **REAL*8.** If they were **COMPLEX*8,** they are to **COMPLEX*16.** Integers are allocated 8 bytes of storage instead of 4 bytes.

–ast=*number*
Allow the compiler to set the initial allocation for the program internal representation to the user defined *number* bytes. Normally, users should not invoke this flag unless the program is terminated with the internal error.

–blanks72
Pad with blanks on right to column 72.

–c Suppress the link editing phase of the compilation, and do not remove any produced object files.

–case_sensitive
Insist that user defined variables must be case sensitive. The default is *-nocase_sensitive.*

–catalog=*name***.in**
Create a database of functions available to be inlined. This option creates two files: the database *name***.in** and an associated file *name***.id.** Each use of catalog adds to the catalog; it does not replace previous versions.

–continuations=*n*
Set the number *n* of continuation lines the compiler accepts for any statement. The default is 19.

-cpp Invoke the C preprocessor on the source file before compiling.

-cross_reference
Generate a cross-reference, if a listing is generated.

-D*name*
Define *name* to have the value of 1, to the preprocessor.

-D*name=val*
Define *name* to have the value of *val*, to the preprocessor.

-debug
Add debug data to the object file. Force optimization level to zero. This is synonymous with **-g**.

-double_precision
Set all undeclared floating point variables and all declared floating point variables to be as if they were declared REAL*8. If they were COMPLEX*8, they are then set to COMPLEX*16. *-nodouble_precision* is the default.

-d_lines
Compile source file lines with a 'D' or 'd' in column 1.

-E Run only *cpp*(1) on the named C programs, and send the result to the standard output.

-extend_source
Extend the statement field of a source line from columns 1 through 72 to columns 1 through 132.

-fast Perform some optimization which may lose small amounts of precision.

-full_report
Produce a detailed vectorizer report.

-fullsubcheck
Generate code to check that every subscript in every array reference is within the bounds of the appropriate array dimensions.

-g Generate additional information needed for the use of *dbg*(1). Force optimization level to zero. This option is synonymous with **-debug**.

-I Suppress the default searching for preprocessor included files in */usr/include*.

-I*dir* Search for preprocessor include files in *dir*. Neither **-I** option affects the Fortran INCLUDE statement.

-i Suppress the automatic production of #ident information.

-implicit
Make all variables in a program untyped.

-include_listing
List included source files as well as the original source file when generating a listing.

-inline
Instruct the compiler to enable function inlining.

-i4 Interpret INTEGER and LOGICAL declarations as if they had been written INTEGER*4 and LOGICAL*4.

-list Generate a listing from the compilation.

-messages
Allow warning messages to be printed.

-Npaths=*name*.**in**
Instruct the compiler to make use of the database of functions listed in the catalog *name*.**in** as the source for inlining.

-no_assoc
Assume floating point is not associative. Force the floating point operation to be in the same order as the program specifies (e.g. no vectorized sum or dot product reductions).

-no_directive
Do not apply compiler directives.

-novector
Not generate no vector code for any loop. This option is especially useful at optimization level 03 when you desire parallel code but no vector code.

-O0 Turn off all optimizations.

-O1 Perform common subexpression elimination and instruction scheduling. If nothing is specified, this **-O1** is the default setting of compiler optimization level.

-O2 Perform **-O1** and vectorization.

-O3 Perform **-O2** and parallelization.

-O This is synonymous with **-O1**.

-o *filename*
Place the output into *filename*.

-object
Generate object files.

-onetrip
Generate code to guarantee that all **DO** loops execute at least once. This is a compatibility feature for programs originally written for use with Fortran 66. The default is *-noonetrip*.

-P Run only *cpp*(1) on the named Fortran programs and leave the result in corresponding files suffixed **.i**. This option is passed to *cpp*(1).

-p Generate code to profile the loaded program during execution. (See *prof*(1) and *mkprof*(1).)

-ploop
Generate code which allows loops within a single routine to be profiled separately.

-r Produce a relocatable output file.

-S Compile and do not assemble the named Fortran programs, and leave the assembler output in corresponding files suffixed **.s**.

-safe_strings
Generate code that corrects for mismatched string parameters. If the type of parameters being passed are truly matched, avoid this option because it causes some performance degradation.

-safe=procs
Cause the compiler to compile a procedure for parallel execution.

-save
All variables declared are saved.

–standard
Check for standard Fortran 77 usage.

–subcheck
Produce code to check at runtime to ensure that each array element accessed is actually part of the appropriate array. However, at optimization level 02 and higher, this option ignores the vector mask. This means that some operations may generate subscriptranges that are not actually in the code.

–U*name*
Undefine *name*.

–V Prints a single line of version information with the release number of the compilation system. Version numbers corresponding to the release number of each executing component are also printed. Version numbers vary independently of each other and of the release number.

–verbose
Use verbose message output.

–vreport
Invoke the vector reporting facility and tell the user what vectorization has been done. A detailed listing is provided for each loop nest and suggestions for achieving better performance are included.

–vsummary
Invoke the vector reporting facility and tell the user what vectorization has been done. Print out what statements are and are not vectorized in each loop. This output is in Fortran-like notation.

–w Suppress warning messages during compilation.

–y*name*
Print uses and definitions of *name*.

–43 Cause the compiler to use */usr/lib/bsd/libc.a* instead of */usr/lib/libc.a*. In another word, this option cause the compiler to use BSD library files instead of System V library files.

The *fc* command recognizes –B *hhhhhhh*, –D *hhhhhhh*, –e*sym*, –L, –L*dir*, –l*tag*, –m, –n, –o*filename*, –opct, –p, –r, –s, –T *hhhhhhh*, –t, –u*name*, –V, and –y*name* and passes these options and their arguments directly to the loader. See the manual pages for *cpp*(1) and *ld*(1) for descriptions.

Other arguments are taken to be Fortran compatible object programs, typically produced by an earlier *fc* run, or perhaps libraries of Fortran compatible routines and are passed directly to the link editor. These programs, together with the results of any compilations specified, are link edited (in the order given) to produce an executable program with name **a.out.**

FILES

file.f	Fortran source file
file.i	preprocessed Fortran source file
file.o	object file
file.s	assembly language file
a.out	link edited output
LIBDIR/fcrt0.o	start-up routine
LIBDIR/file.i	inlined functions file

LIBDIR/file.V	vector reporting facility file
LIBDIR/file.L	listing file
TMPDIR/*	temporary files
LIBDIR/cpp	preprocessor, *cpp*(1)
BINDIR/as	assembler, *as*(1)
BINDIR/ld	link editor, *ld*(1)
LIBDIR/libc.a	standard C library

LIBDIR is usually /lib
BINDIR is usually /bin

TMPDIR is usually /tmp but can be redefined by setting the environment variable **TMPDIR** [see *tempnam*() in *tmpnam*(3S)], and exporting it to the environment in which the programs run.

ENVIRONMENT VARIABLES

Setting a couple of shell environment variables to either **VMS** or **BSD** allows you to decide how files should be read and written. These variables are:

UNFORMATTED_IO
UNFORMATTED_INPUT
UNFORMATTED_OUTPUT

The setting of **UNFORMATTED_IO** takes precedence over the other two variables. If it is set, any setting of **UNFORMATTED_INPUT** or **UNFORMATTED_OUTPUT** is ignored. The available settings are described below:

UNFORMATTED_INPUT BSD
On input, the record length is measured in bytes; there is no padding.

UNFORMATTED_INPUT VMS
Default.
On input, the length is in words; padding rounds up to even 4-byte boundaries.

UNFORMATTED_OUTPUT BSD
On output, the record length is written in bytes; there is no padding.

UNFORMATTED_OUTPUT VMS
Default.
On output, the record length is written in words (length rounded up to nearest 4-byte boundary) and padding is up to the nearest 4-byte boundary.

UNFORMATTED_IO BSD
On input and output, the record length is both read and written in bytes; there is no padding.

UNFORMATTED_IO VMS
Default.
On input and output, the record length is read and written in words (rounded up to the nearest 4-byte boundary); padding is up to the nearest 4-byte boundary.

SEE ALSO

as(1), ld(1), cpp(1), prof(1), dbg(1)
Programmer's Guide
Fortran Reference Manual

NAME

fpr, asa – print Fortran file

SYNOPSIS

fpr
asa

DESCRIPTION

fpr (*asa*) is a filter that transforms files formatted according to Fortran's carriage control conventions into files formatted according to UNIX line printer conventions.

fpr (*asa*) copies its input onto its output, replacing the carriage control characters with characters that produce the intended effects when printed. The first character of each line determines the vertical spacing as follows:

Character	Vertical Space Before Printing
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

A blank line is treated as if its first character is a blank. A blank that appears as a carriage control character is deleted. A zero is changed to a newline. A one is changed to a form feed. The effects of a "+" are simulated using backspaces.

EXAMPLES

a.out | fpr | lp

a.out | asa | lp

fpr < fc.output | lp

BUGS

Results are undefined for input lines longer than 170 characters.

NAME

fsplit – split a multi-routine Fortran file into individual files

SYNOPSIS

fsplit [**-e** efile] ... [file]

DESCRIPTION

fsplit takes as input either a file or standard input containing Fortran source code. It attempts to split the input into separate routine files of the form *name.f*, where *name* is the name of the program unit (e.g. function, subroutine, block data or program). The name for unnamed block data subprograms has the form *blkdtaNNN.f* where NNN is three digits and a file of this name does not already exist. For unnamed main programs the name has the form *mainNNN.f*. If there is an error in classifying a program unit, or if *name.f* already exists, the program unit will be put in a file of the form *zzzNNN.f* where *zzzNNN.f* does not already exist.

Normally each subprogram unit is split into a separate file. When the *-e* option is used, only the specified subprogram units are split into separate files. E.g.:

fsplit -e *readit* -e *doit* prog.f

will split *readit* and *doit* subprograms into separate files.

DIAGNOSTICS

If names specified via the *-e* option are not found, a diagnostic is written to *standard error.*

AUTHOR

Asa Romberger and Jerry Berkman

BUGS

Fsplit assumes the subprogram name is on the first noncomment line of the subprogram unit. Nonstandard source formats may confuse *fsplit.*

It is hard to use *-e* for unnamed main programs and block data subprograms since you must predict the created file name.

NAME

ratfor – rational Fortran dialect

SYNOPSIS

ratfor [option ...] [filename ...]

DESCRIPTION

Ratfor converts a rational dialect of Fortran into ordinary irrational Fortran. *Ratfor* provides control flow constructs essentially identical to those in C:

statement grouping:
    { statement; statement; statement }

decision-making:
    if (condition) statement [else statement]
    switch (integer value) {
        case integer: statement
        ...
        [default:] statement
    }

loops:
    while (condition) statement
    for (expression; condition; expression) statement
    do limits statement
    repeat statement [until (condition)]
    break
    next

and some syntactic sugar to make programs easier to read and write:

free form input:
    multiple statements/line; automatic continuation

comments:
    # this is a comment

translation of relationals:
    >, >=, etc., become .GT., .GE., etc.

return (expression)
    returns expression to caller from function

define:
    define name replacement

include:
    include filename

SEE ALSO

B. W. Kernighan and P. J. Plauger, *Software Tools*, Addison-Wesley, 1976.

LIBRARY FUNCTIONS

CHAPTER ELEVEN

11.1 A Compendium of Library Functions

This chapter lists the UNIX library functions that are built into the Fortran support library. These include

- Fortran I/O functions
- File Access functions
- Asynchronous I/O functions
- System functions
- Timing functions
- UNIX Utility functions
- VMS Compatability functions

Intrinsic functions are provided separately in Chapter 6, *Intrinsic Functions*, and inline functions in Chapter 8, *Stardent 1500/3000 Fortran Optimization Facilities* under the section titled *Inline Functions*.

Stardent 1500/3000 Fortran also supports a small number of library routines to simulate VMS library functions for VMS code compatibility. These routines can be very useful, especially, in doing benchmarking tasks. These routines are located in */usr/lib/libVF77.a*. Please refer to the *VMS Programming Support Manual* for detailed information.

These functions may be directly called from your source program. The loader links the library functions into your executable file.

NAME

intro – introduction to Fortran library functions

DESCRIPTION

This section describes those functions that are in the Fortran run time library. The functions listed here provide an interface from Fortran programs to the system in the same manner as the C library does for C programs. They are automatically loaded as needed by the Fortran compiler *fc*(1).

Most of these functions are in *libuF77.a*. Some are in *libmF77.a* or *libiF77.a*. A few intrinsic functions are described for the sake of completeness. Refer to the *Fortran Reference Manual* in chapter 6, *Intrinsic Functions* for additional information.

Inline functions are located in */usr/lib/libbF77.in*, and also in compiled form in */usr/lib/libbF77.a*. Refer to the *Fortran Reference Manual* in chapter 9, *Stardent 1500/3000 Fortran Elements* under the section titled, *Inline Functions* for detailed information.

A small number of library routines are supported to simulate VMS library functions for VMS code compatibility. These routines can be located in */usr/lib/libVF77.a*. Please refer to the *VMS Programming Support Manual* for further information.

All Fortran character variables must have a constant length declared for them. In the routine synopses which follow, this length will be shown as CHARACTER*N or *M, as needed. In an actual program, N or M would have to be replaced with a constant value.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
ABLOCK	ablock.3f	asynchronous block read
ABORT	abort.3f	abnormal termination
ACCESS	access.3f	determine accessibility of a file
ALARM	alarm.3f	execute a subroutine after a specified time
AREAD	aread.3f	asynchronous read
ASTATUS	astatus.3f	status of an asychronous read
CHDIR	chdir.3f	change default directory
CHMOD	chmod.3f	change mode of a file
CLEAR_ALL_MASKS	exception.3f	arithmetic exception control routine
CLEAR_ARITHMETIC_INEXACT	exception.3f	arithmetic exception control routine
CLEAR_ARITHMETIC_OVERFLOW	exception.3f	arithmetic exception control routine
CLEAR_ARITHMETIC_UNDERFLOW	exception.3f	arithmetic exception control routine
CLEAR_DIVIDE_BY_ZERO	exception.3f	arithmetic exception control routine
CLEAR_INPUT_IS_NAN	exception.3f	arithmetic exception control routine
CLEAR_INVALID_OPERAND	exception.3f	arithmetic exception control routine
CPUTIM	cputim.3f	returns elapsed cpu time
CTIME	time.3f	return system time
DFFRAC	flmin.3f	return extreme values
DFLMAX	flmin.3f	return extreme values
DFLMIN	flmin.3f	return extreme values
DRAND	drand.3f	return double precision random values

DRAND48	drand48.3f	return double precision random values
DTIME	etime.3f	return elapsed execution time
ERAND48	drand48.3f	return double precision random values
ETIME	etime.3f	return elapsed execution time
EXIT	exit.3f	terminate process with status
FALLOC	malloc.3f	memory allocator
FDATE	fdate.3f	return date and time in an ASCII string
FFRAC	flmin.3f	return extreme values
FGETC	getc.3f	get a character from a logical unit
FLMAX	flmin.3f	return extreme values
FLMIN	flmin.3f	return extreme values
FLUSH	flush.3f	flush output to a logical unit
FORK	fork.3f	create a copy of this process
FPUTC	putc.3f	write a character to a Fortran logical unit
FPUTIM	cputim.3f	timing function
FREE	malloc.3f	memory allocator
FSEEK	fseek.3f	reposition a file on a logical unit
FSTAT	stat.3f	get file status
FTELL	fseek.3f	reposition a file on a logical unit
GERROR	perror.3f	get system error messages
GETARG	getarg.3f	return command line arguments
GETC	getc.3f	get a character from a logical unit
GETCWD	getcwd.3f	get pathname of current working directory
GETENV	getenv.3f	get value of environment variables
GETGID	getuid.3f	get user or group ID of the caller
GETHOSTID	gethostid.3f	returns unique processor ID numbers
GETLOG	getlog.3f	get user's login name
GETPID	getpid.3f	get process id
GETUID	getuid.3f	get user or group ID of the caller
GMTIME	time.3f	return system time
HOSTNM	hostnm.3f	get name of current host
IARGC	getarg.3f	return command line arguments
IDATE	idate.3f	return date or time in numerical form
IERRNO	perror.3f	get system error messages
INMAX	flmin.3f	return extreme values
IRAND	rand.3f, drand.3f	return random integer values; return double precision random integer values
ISATTY	ttynam.3f	find name of a terminal port
ITIME	idate.3f	return date or time in numerical form
JRAND48	drand48.3f	return double precision random values
KILL	kill.3f	send a signal to a process
LCONG48	drand48.3f	return double precision random values
LINK	link.3f	make a link to an existing file
LNBLNK	rindex.3f	tell about character objects
LOC	loc.3f	return the address of an object
LONG	long.3f	integer object conversion
LRAND48	drand48.3f	return double precision random values
LSTAT	stat.3f	get file status
LTIME	time.3f	return system time
MALLOC	malloc.3f	memory allocator
MCLOCK	mclock.3f	return Fortran time accounting
MRAND48	drand48.3f	return double precision random values
NRAND48	drand48.3f	return double precision random values
PERROR	perror.3f	get system error messages

PUTC	putc.3f	write a character to a Fortran logical unit
QSORT	qsort.3f	perform quick sort
RAND	rand.3f,	return random integer values;
	drand.3f	return random double precision values
READ_FLOATING_MASK		
	exception.3f	arithmetic exception control routine
RENAME	rename.3f	rename a file
RINDEX	rindex.3f	tell about character objects
SEED48	drand48.3f	return double precision random values
SET_ALL_MASKS		
	exception.3f	arithmetic exception control routine
SET_ARITHMETIC_INEXACT		
	exception.3f	arithmetic exception control routine
SET_ARITHMETIC_OVERFLOW		
	exception.3f	arithmetic exception control routine
SET_ARITHMETIC_UNDERFLOW		
	exception.3f	arithmetic exception control routine
SET_DIVIDE_BY_ZERO		
	exception.3f	arithmetic exception control routine
SET_FLOATING_MASK		
	exception.3f	arithmetic exception control routine
SET_INVALID_OPERAND		
	exception.3f	arithmetic exception control routine
SET_INPUT_IS_NAN		
	exception.3f	arithmetic exception control routine
SHORT	long.f	integer object conversion
SIGNAL	signal.3f	change the action for a signal
SLEEP	sleep.3f	suspend execution for an interval
SRAND	rand.3f	return seeded random integer values
SRAND48	drand48.3f	return double precision random values
STAT	stat.3f	get file status
SYSTEM	system.3f	execute a Unix command
SYSTIM	cputim.3f	timing function
TCLOSE	topen.3f	Fortran tape I/O
TIME	time.3f	return system time
TOPEN	topen.3f	Fortran tape I/O
TREAD	topen.3f	Fortran tape I/O
TREWIN	topen.3f	Fortran tape I/O
TSKIPF	topen.3f	Fortran tape I/O
TSTATE	topen.3f	Fortran tape I/O
TTYNAM	ttynam.3f	find name of a terminal port
TWRITE	topen.3f	Fortran tape I/O
UNLINK	unlink.3f	remove a directory entry

NAME

ablock – wait for asynchronous read to complete

SYNOPSIS

```
EXTERNAL ABLOCK
INTEGER ABLOCK
INTEGER UNIT
INTEGER CODE
CODE = ABLOCK(UNIT)
```

DESCRIPTION

UNIT is the unit number used in an *OPEN* statement. The *OPEN* on this unit must have had *ACCESS='ASYNC'* or it is an error.

ABLOCK suspends execution of other process until no asynchronous file read is pending on the specified *UNIT*. *ABLOCK* returns the count of bytes read. This count should be the value of *NBYTES* in the call to *AREAD*. *CODE* is the actual number of bytes being read.

SEE ALSO

AREAD(3F), ABLOCK(3F), ASTATUS(3F), read(2), astat(2)

NAME

abort – terminate Fortran program

SYNOPSIS

```
EXTERNAL ABORT
CALL ABORT ( )
```

DESCRIPTION

ABORT terminates the program which calls it, closing all open files truncated to the current position of the file pointer. Abort usually results in a core dump.

DIAGNOSTICS

When invoked, *ABORT* prints "Fortran abort routine called" on the standard error output. The shell prints the message "abort - core dumped" if a core dump results.

SEE ALSO

abort(3C), sh(1)

NAME

access – determine accessibility of a file

SYNOPSIS

```
EXTERNAL ACCESS
INTEGER ACCESS
CHARACTER*N NAME
CHARACTER*M MODE
INTEGER CODE
CODE = ACCESS(NAME,MODE)
```

DESCRIPTION

ACCESS checks the given file, *NAME*, for accessibility with respect to the caller according to *MODE*. *MODE* may include in any order and in any combination one or more of

r	test for read permission
w	test for write permission
x	test for execute permission
(blank)	test for existence

An error code is returned if either argument is illegal, or if the file cannot be accessed in all of the specified modes. 0 is returned if the specified access would be successful.

FILES

/usr/lib/libuF77.a

SEE ALSO

access(2), PERROR(3F)

BUGS

Pathnames can be no longer than PATH_MAX characters as defined in <*limits.h*>.

NAME

alarm – execute a subroutine after a specified time

SYNOPSIS

```
EXTERNAL ALARM
INTEGER ALARM
INTEGER TIME
EXTERNAL PROC
INTEGER REMAIN
REMAIN = ALARM(TIME,PROC)
```

DESCRIPTION

This routine arranges for subroutine *PROC* to be called after *time* seconds. If *TIME* is "0", the alarm is turned off and no routine will be called. The returned value will be the time remaining on the last alarm.

FILES

/usr/lib/libuF77.a

SEE ALSO

ALARM(3C), SLEEP(3F), SIGNAL(3F)

BUGS

ALARM and *SLEEP* interact. If *SLEEP* is called after *ALARM*, the *ALARM* process will never be called. SIGALRM will occur at the lesser of the remaining *ALARM* time or the *SLEEP* time.

NAME

aread – asynchronous read

SYNOPSIS

```
EXTERNAL AREAD
CHARACTER BUFF*1(4096)
INTEGER UNIT, BLOCK_NO, NBYTES
CALL AREAD(UNIT, BUFF, BLOCK_NO, NBYTES)
```

DESCRIPTION

UNIT is the unit number used in an *OPEN* statement. The *OPEN* on this unit must have had *ACCESS='ASYNC'* or it is an error.

On devices capable of seeking, *AREAD* attempts to read *NBYTES* asynchronously from the file associated with *UNIT* at the 4096-byte block boundary specified by *BLOCK_NO* into the buffer *BUFF*. The block number specified in *BLOCK_NO* should start at 1, and each block has a size of 4096 byte, independent of the record length specified in the *OPEN* statement.

AREAD returns immediately after the read request has been issued to the appropriate device. Upon return, completion of the read request is not guaranteed. No assumption should be made regarding the amount of data. You can be guaranteed that data transfer is completed only by using *ABLOCK* to wait for *AREAD* to complete.

The offset into the file must be exactly on 4096-byte boundaries, the amount of the read request must be for multiples of 4096 bytes, and the user buffer *BUFF* must be on a 4-byte (word) boundary.

SEE ALSO

ABLOCK(3F), ASTATUS(3F), read(2), astat(2), await(2)

NAME

astatus – check status of an asynchronous read

SYNOPSIS

```
EXTERNAL ASTATUS
LOGICAL ASTATUS
INTEGER UNIT
LOGICAL AST
AST = ASTATUS(UNIT)
```

DESCRIPTION

UNIT is the unit number used in an *OPEN* statement. The *OPEN* on this unit must have had *ACCESS='ASYNC'* or it is an error.

ASTATUS returns the status of a pending asynchronous file read operation.

RETURN VALUE

The value of .TRUE. is returned if the read is complete, and .FALSE. is returned if it is still pending.

SEE ALSO

AREAD(3F), ABLOCK(3F), astat(2), await(2), read(2)

NAME

chdir – change default directory

SYNOPSIS

```
EXTERNAL CHDIR
INTEGER CHDIR
CHARACTER*N DIRNAME
INTEGER CODE
CODE = CHDIR(DIRNAME)
```

DESCRIPTION

The default directory for creating and locating files will be changed to *DIRNAME*. Zero is returned if successful, or an error code is returned otherwise.

FILES

/usr/lib/libuF77.a

SEE ALSO

chdir(2), cd(1), PERROR(3F)

BUGS

Pathnames can be no longer than PATH_MAX characters as defined in <*limits.h*>.

Use of this function may cause **INQUIRE** by unit to fail.

NAME

chmod – change mode of a file

SYNOPSIS

```
EXTERNAL CHMOD
INTEGER CHMOD
CHARACTER*N NAME
CHARACTER*M MODE
INTEGER CODE
CODE = CHMOD(NAME, MODE)
```

DESCRIPTION

This function changes the filesystem *MODE* of file *NAME*. *MODE* can be any specification recognized by *chmod(1)*. *NAME* must be a single pathname.

The normal returned value is 0. Any other value will be a system error number.

FILES

/usr/lib/libuF77.a
/bin/chmod

SEE ALSO

chmod(1)

BUGS

Pathnames can be no longer than PATH_MAX characters as defined in <*limits.h*>.

NAME

cputim, systim, fputim – timing functions

SYNOPSIS

```
EXTERNAL CPUTIM
REAL CPUTIM
REAL OLDTIM
REAL TIME
TIME = CPUTIM(OLDTIM)

EXTERNAL SYSTIM
REAL SYSTIM
REAL OLDTIM
REAL TIME
TIME = SYSTIM(OLDTIM)

EXTERNAL FPUTIM
DOUBLE PRECISION FPUTIM
DOUBLE PRECISION OLDTIM
DOUBLE PRECISION TIME
TIME = FPUTIM(OLDTIM)

INTRINSIC SECNDS
REAL SECNDS
REAL OLDTIM
REAL TIME
TIME = SECNDS(OLDTIM)
```

DESCRIPTION

The real valued function *CPUTIM* returns the elapsed CPU time used by this process minus the value of its argument *OLDTIM*. The time is expressed in seconds and is accurate to 1/100th of a second.

The real valued function *SYSTIM* returns the elapsed system time used by this process minus the the value of its argument *OLDTIM*. The time is expressed in seconds and is accurate to 1/100th of a second.

The double precision valued function *FPUTIM* returns the elapsed floating point processor time used by this process minus the value of its argument *OLDTIM*. The time is expressed in seconds and is accurate to 62.5 nanoseconds.

The real valued function *SECNDS* returns the elapsed time since midnight minus the value of its argument *OLDTIM*. The time is expressed in seconds and is accurate to 1/100th of a second.

NOTES

If the argument to *CPUTIM*, *SYSTIM*, *FPUTIM*, or *SECNDS* is zero, then the value returned is simply the time used by this process or the time since midnight. When the argument value to any of these functions is non-zero, the function can be used as a split timer.

Notice that the argument to *FPUTIM* is a double precision value. If a constant is used, it must be specified as double precision or an error will occur at runtime. For example, if zero is desired, the constant *0.0D0* should be used.

NAME

drand, rand, irand – return random values

SYNOPSIS

```
INTEGER IFLAG

EXTERNAL DRAND
DOUBLE PRECISION DRAND
DOUBLE PRECISION X
X = DRAND(IFLAG)

EXTERNAL RAND
REAL RAND
REAL Y
Y = RAND(IFLAG)

EXTERNAL IRAND
INTEGER IRAND
INTEGER I
I = IRAND(IFLAG)
```

DESCRIPTION

The double precision function *DRAND* returns pseudo-random double precision values in the closed interval [0.0, 1.0].

The real function *RAND* returns pseudo-random real values in the closed interval [0.0, 1.0].

The integer function *IRAND* returns pseudo-random integer values in the closed interval [0, 2147483647].

If the integer argument *IFLAG* has the value 1, the generator is restarted and the first random value is returned. If *IFLAG* has any other non-zero value, it is used as a new seed for the random number generator and the call returns the first value generated from the new seed. If *IFLAG* has the value zero, the next pseudo-random value in the current sequence is returned without reseeding.

NOTES

This family of routines is supplied only for compatibility with BSD. Either the routines in *RAND (3F)* or *DRAND48 (3F)* are to be preferred.

To use these routines, you must explicitly include */usr/lib/fortran/drand.o* on your compilation or load line.

FILES

/usr/lib/fortran/drand.o

SEE ALSO

RAND(3F), DRAND48(3F)

NAME

drand48, erand48, lrand48, nrand48, mrand48, jrand48, srand48, seed48, lcong48 – generate uniformly distributed pseudo-random numbers

SYNOPSIS

```
INTEGER SEED
INTEGER XSEED(3)
INTEGER OXSEED(3)
INTEGER PARAMS(7)

DOUBLE PRECISION X
INTEGER I

EXTERNAL DRAND48
DOUBLE PRECISION DRAND48
X = DRAND48()

EXTERNAL ERAND48
DOUBLE PRECISION ERAND48
X = ERAND48(XSEED)

EXTERNAL LRAND48
INTEGER LRAND48
I = LRAND48()

EXTERNAL NRAND48
INTEGER NRAND48
I = NRAND48(XSEED)

EXTERNAL MRAND48
INTEGER MRAND48
I = MRAND48()

EXTERNAL JRAND48
INTEGER JRAND48
I = JRAND48(XSEED)

EXTERNAL SRAND48
CALL SRAND48(SEED)

EXTERNAL SEED48
CALL SEED48(XSEED, OXSEED)

EXTERNAL LCONG48
CALL LCONG48(PARAMS)
```

DESCRIPTION

This family of functions generates pseudo-random numbers using the well-known linear congruential algorithm and 48-bit integer arithmetic.

Functions *DRAND48* and *ERAND48* return non-negative double precision floating-point values uniformly distributed over the half-open interval [0.0, 1.0).

Functions *LRAND48* and *NRAND48* return non-negative integers uniformly distributed over the interval $[0, 2^{31})$.

Functions *MRAND48* and *JRAND48* return signed integers uniformly distributed over the interval $[-2^{31}, 2^{31})$.

Functions *SRAND48*, *SEED48*, and *LCONG48* are initialization entry points, one of which should be invoked before either *DRAND48*, *LRAND48*, or *MRAND48* is called. (Although it is not recommended practice, constant default initializer values will be supplied automatically if *DRAND48*, *LRAND48*, or *MRAND48* is called without a prior call to an initialization entry point.) Functions *ERAND48*, *NRAND48*, and

JRAND48 do not require an initialization entry point to be called first.

All the routines work by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula

$$X_{n+1} = (aX_n + c) \bmod m \quad n \geq 0$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless *LCONG48* has been invoked, the multiplier value *a* and the addend value *c* are given by

a = '5DEECE66D'X = '273673163155'O
c = 'B'X = '13'O.

The value returned by any of the functions *DRAND48*, *ERAND48*, *LRAND48*, *NRAND48*, *MRAND48*, or *JRAND48* is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (leftmost) bits of X_i and transformed into the returned value.

The functions *DRAND48*, *LRAND48*, and *MRAND48* store the last 48-bit X_i generated in an internal buffer and must be initialized prior to being invoked. The functions *ERAND48*, *NRAND48*, and *JRAND48* require the calling program to provide storage for the successive X_i values in the array specified as an argument when the functions are invoked. These routines do not have to be initialized; the calling program must place the desired initial value of X_i into the array and pass it as an argument. By using different arguments, functions *ERAND48*, *NRAND48*, and *JRAND48* allow separate modules of a large program to generate several *independent* streams of pseudo-random numbers; that is, the sequence of numbers in each stream will *not* depend upon how many times the routines have been called to generate numbers for the other streams.

The initializer function *SRAND48* sets the high-order 32 bits of X_i to the 32 bits contained in its argument. The low-order 16 bits of X_i are set to the arbitrary value $330E_{16}$.

The initializer procedure *SEED48* sets the value of X_i to the 48-bit value specified in the argument array *XSEED*. In addition, the previous value of X_i is copied into the second argument array *OXSEED*. This returned array, which can just be ignored if not needed, is useful if a program is to be restarted from a given point in the random number sequence at some future time—use the old values in argument *OXSEED* to reinitialize the generator when the program is restarted.

The initialization function *LCONG48* allows the user to specify the initial X_i, the multiplier value *a*, and the addend value *c*. Argument array *PARAMS* elements 1, 2, and 3 specify X_i, elements 4, 5, and 6 specify the multiplier *a*, and element 7 specifies the 16-bit addend *c*. After *LCONG48* has been called, a subsequent call to either *SRAND48* or *SEED48* will restore the "standard" multiplier and addend values *a* and *c*.

The functions and procedures *ERAND48*, *NRAND48*, *JRAND48*, *SEED48*, and *LCONG48* all take arrays of 3 or 7 integers as arguments. In each array element, only the least significant 16 bits is actually used to set parameter values; the most significant 16 bits are ignored. Thus, the 48-bit long seed value value X_i is formed after a call to *ERAND48* (for example) by concatenating the low 16 bits of *XSEED(1)*, *XSEED(2)*, and *XSEED(3)* from left to right.

SEE ALSO

RAND(3F)
drand48(3C)

NAME

etime, dtime – return elapsed execution time

SYNOPSIS

```
EXTERNAL ETIME
REAL ETIME
REAL TARRAY(2)
REAL TSECS
TSECS = ETIME(TARRAY)

EXTERNAL DTIME
REAL DTIME
REAL TARRAY(2)
REAL DTSECS
DTSECS = DTIME(TARRAY)
```

DESCRIPTION

These two routines return elapsed runtime in seconds for the calling process. *DTIME* returns the elapsed time since the last call to *DTIME* or the start of execution on the first call.

The argument array returns user time in the first element and system time in the second element. The function value is the sum of user and system time.

The resolution of all timing is 1/100th of a second.

FILES

/usr/lib/libuF77.a

SEE ALSO

times(2)

NAME

exception – arithmetic exception control routines

SYNOPSIS

```
LOGICAL MASK(6)

EXTERNAL SET_FLOATING_MASK
CALL SET_FLOATING_MASK(MASK)

EXTERNAL READ_FLOATING_MASK
CALL READ_FLOATING_MASK(MASK)

EXTERNAL SET_INPUT_IS_NAN
CALL SET_INPUT_IS_NAN

EXTERNAL CLEAR_INPUT_IS_NAN
CALL CLEAR_INPUT_IS_NAN

EXTERNAL SET_DIVIDE_BY_ZERO
CALL SET_DIVIDE_BY_ZERO

EXTERNAL CLEAR_DIVIDE_BY_ZERO
CALL CLEAR_DIVIDE_BY_ZERO

EXTERNAL SET_INVALID_OPERAND
CALL SET_INVALID_OPERAND

EXTERNAL CLEAR_INVALID_OPERAND
CALL CLEAR_INVALID_OPERAND

EXTERNAL SET_ARITHMETIC_OVERFLOW
CALL SET_ARITHMETIC_OVERFLOW

EXTERNAL CLEAR_ARITHMETIC_OVERFLOW
CALL CLEAR_ARITHMETIC_OVERFLOW

EXTERNAL SET_ARITHMETIC_UNDERFLOW
CALL SET_ARITHMETIC_UNDERFLOW

EXTERNAL CLEAR_ARITHMETIC_UNDERFLOW
CALL CLEAR_ARITHMETIC_UNDERFLOW

EXTERNAL SET_ARITHMETIC_INEXACT
CALL SET_ARITHMETIC_INEXACT

EXTERNAL CLEAR_ARITHMETIC_INEXACT
CALL CLEAR_ARITHMETIC_INEXACT

EXTERNAL SET_ALL_MASKS
CALL SET_ALL_MASKS

EXTERNAL CLEAR_ALL_MASKS
CALL CLEAR_ALL_MASKS
```

DESCRIPTION

Stardent 1500/3000 Fortran programs normally execute with all floating point exceptions turned off; errors such as divide by zero, overflow, and underflow are propagated variously into NaN (not-a-number), INF (infinity), and zero according to the rules of IEEE arithmetic. For debugging purposes, it may be desirable to change the handling of errors so that a program terminates immediately upon execution of an erroneous operation. There are six kinds of errors that can be managed;

Input operand is NaN.
Divide by zero.
Invalid input operand.

Overflow.
Underflow.
Inexact result.

For each kind of error, there is a corresponding element in a boolean mask vector and there is a corresponding trap. If the error occurs and the corresponding mask element is true, then the program will be aborted with a runtime trap. The routines in this suite control the handling of arithmetic errors by allowing manipulation of the mask vector.

A call to *READ_FLOATING_MASK* reads the current state of the mask vector and returns a logical value for each kind of error; *MASK(1)* is *.TRUE.* if an input NaN operand will cause a trap, *MASK(2)* is *.TRUE.* if a divide by zero will cause a trap, and so on. A call to *SET_FLOATING_MASK* changes the elements of the mask vector as specified by its logical argument vector; the trap for an input NaN operand is set if *MASK(1)* is *.TRUE.*, the trap for divide by zero is set if *MASK(2)* is *.TRUE.*, and so on.

The other routines listed above set or clear an individual mask element, as the name implies; none of them takes an argument. The routines *SET_ALL_MASKS* and *CLEAR_ALL_MASKS* set or clear all of the mask elements in a single operation.

NOTES

Some service routines may rely on the traps being in the normal (all clear) state for Fortran execution; changing the mask values may cause these routines to fail. For example, many random number routines rely on overflow being ignored.

NAME

exit – terminate process with status

SYNOPSIS

```
EXTERNAL EXIT
INTEGER STATUS
CALL EXIT(STATUS)
```

DESCRIPTION

EXIT flushes and closes all the process's files and notifies the parent process if it is executing a *WAIT*. The low-order 8 bits of *STATUS* are available to the parent process. (Therefore *STATUS* should be in the range 0 – 255)

This call never returns.

There may be cleanup actions before the final system exit.

FILES

/usr/lib/libuF77.a

SEE ALSO

exit(2), fork(2), FORK(3F), wait(2), WAIT(3F)

NAME

fdate – return date and time in an ASCII string

SYNOPSIS

```
EXTERNAL FDATE
CHARACTER*24 STRING
CALL FDATE(STRING)
```

DESCRIPTION

FDATE returns the current date and time as a 24 character string in the format described under *ctime(3)*. Neither newline nor NULL will be included.

FDATE can be called either as a function or as a subroutine. If called as a function, the calling routine must define its type and length. For example

```
EXTERNAL FDATE
CHARACTER*24 FDATE
CHARACTER*24 VALUE
VALUE = FDATE()
```

Notice that when *FDATE* is called as a function, no argument is supplied.

FILES

/usr/lib/libuF77.a

SEE ALSO

ctime(3), TIME(3F), ITIME(3F), IDATE(3F), LTIME(3F)

NAME

flmin, flmax, ffrac, dflmin, dflmax, dffrac, inmax – return extreme values

SYNOPSIS

```
INTEGER I
REAL R
DOUBLE PRECISION D

EXTERNAL FLMIN
REAL FLMIN
R= FLMIN()

EXTERNAL FLMAX
REAL FLMAX
R = FLMAX()

EXTERNAL FFRAC
REAL FFRAC
R = FFRAC()

EXTERNAL DFLMIN
DOUBLE PRECISION DFLMIN
D = DFLMIN()

EXTERNAL DFLMAX
DOUBLE PRECISION DFLMAX
D = DFLMAX()

.B EXTERNAL DFFRAC
DOUBLE PRECISION DFFRAC
D = DDFRAC()

EXTERNAL INMAX
INTEGER INMAX
I = INMAX()
```

DESCRIPTION

Functions *FLMIN* and *FLMAX* return the minimum and maximum positive floating point values respectively. Functions *DFLMIN* and *DFLMAX* return the minimum and maximum positive double precision floating point values. Function *INMAX* returns the maximum positive integer value.

The functions *FFRAC* and *DFFRAC* return the fractional accuracy of single and double precision floating point numbers respectively. This is the difference between 1.0 and the smallest real number greater than 1.0.

These functions can be used by programs which must scale algorithms to the numerical range of the processor.

FILES

/usr/lib/libuF77.a

NAME

flush – flush output to a logical unit

SYNOPSIS

```
SUBROUTINE FLUSH
INTEGER LUNIT
CALL FLUSH(LUNIT)
```

DESCRIPTION

FLUSH causes the contents of the buffer for logical unit *LUNIT* to be flushed to the associated file. This is most useful for logical units 0 and 6 when they are both associated with the control terminal.

FILES

/usr/lib/libuF77.a

SEE ALSO

FCLOSE(3S)

NAME

fork – create a copy of this process

SYNOPSIS

```
EXTERNAL FORK
INTEGER FORK
INTEGER CHILD
CHILD = FORK()
```

DESCRIPTION

FORK creates a copy of the calling process. The only distinction between the two processes is that the value returned to one of them (referred to as the "parent" process) will be the process id of the copy. The copy is usually referred to as the "child" process. The value returned to the "child" process will be zero.

All logical units open for writing are flushed before the fork to avoid duplication of the contents of I/O buffers in the external file(s).

If the returned value is negative, it indicates an error and will be the negation of the system error code. See PERROR(3F).

A corresponding *EXEC* routine has not been provided because there is no satisfactory way to retain open logical units across the exec. However, the usual function of *FORK/EXEC* can be performed using *SYSTEM*(3F).

FILES

/usr/lib/libuF77.a

SEE ALSO

fork(2), WAIT(3F), KILL(3F), SYSTEM(3F), PERROR(3F)

NAME

fseek, ftell – reposition a file on a logical unit

SYNOPSIS

```
EXTERNAL FSEEK
INTEGER FSEEK
INTEGER LUNIT
INTEGER OFFSET
INTEGER FROM
INTEGER SUCC
SUCC = FSEEK(LUNIT, OFFSET, FROM)

EXTERNAL FTELL
INTEGER FTELL
INTEGER LUNIT
INTEGER WHERE
WHERE = FTELL(LUNIT)
```

DESCRIPTION

FSEEK causes the file associated with logical unit *LUNIT* to be repositioned. *LUNIT* must refer to an open logical unit. *OFFSET* is an offset in bytes relative to the position specified by *FROM*. Valid values for *FROM* are:

0 meaning 'beginning of the file'
1 meaning 'the current position'
2 meaning 'the end of the file'

The value returned by *FSEEK* will be 0 if successful, a system error code otherwise. (See PERROR(3F))

FTELL returns the current position of the file associated with the specified logical unit. The value is an offset, in bytes, from the beginning of the file. If the value returned is negative, it indicates an error and will be the negation of the system error code. (See PERROR(3F))

FILES

/usr/lib/libuF77.a

SEE ALSO

fseek(3S), PERROR(3F)

NAME

getarg – return Fortran command-line argument

SYNOPSIS

```
EXTERNAL GETARG
CHARACTER*M GETARG
CHARACTER*N C
CHARACTER*N VALUE
INTEGER I
VALUE = GETARG(I, C)
```

DESCRIPTION

GETARG returns the *I-th* command-line argument of the current process. Thus, if a program were invoked via

```
foo arg1 arg2 arg3
```

GETARG(2, C) would return the string "arg2" in the character variable *C*.

SEE ALSO

getopt(3C)

NAME

getc, fgetc – get a character from a logical unit

SYNOPSIS

```
EXTERNAL GETC
INTEGER GETC
CHARACTER*N CHAR
INTEGER GOT
GOT = GETC(CHAR)

EXTERNAL FGETC
INTEGER FGETC
INTEGER LUNI
CHARACTER*N CHAR
INTEGER FGOT
FGOT = FGETC(LUNIT, CHAR)
```

DESCRIPTION

These routines return the next character from a file associated with a Fortran logical unit, bypassing normal Fortran I/O. *GETC* reads from logical unit 5, normally connected to the control terminal input. The character read is returned in argument CHAR.

The value of each function is a system status code. Zero indicates no error occurred on the read; –1 indicates end of file was detected. A positive value will be either a Unix system error code or a Fortran I/O error code. See PERROR(3F).

FILES

/usr/lib/libuF77.a

SEE ALSO

getc(3S), intro(2), PERROR(3F)

NAME

getcwd – get pathname of current working directory

SYNOPSIS

```
EXTERNAL GETCWD
INTEGER GETCWD
CHARACTER*N DIRNAME
INTEGER PATH
PATH = GETCWD(DIRNAME)
```

DESCRIPTION

The pathname of the default directory for creating and locating files will be returned in *DIRNAME*. The value of the function will be zero if successful; an error code otherwise.

FILES

/usr/lib/libuF77.a

SEE ALSO

CHDIR(3F), PERROR(3F)

BUGS

Pathnames can be no longer than PATH_MAX characters as defined in <*limits.h*>.

NAME

getenv – return Fortran environment variable

SYNOPSIS

```
EXTERNAL GETENV
CHARACTER*M C
CHARACTER*N NAME
CALL GETENV(NAME, C)
```

DESCRIPTION

GETENV returns the character-string value of the environment variable represented by its first argument into the character variable of its second argument. If no such environment variable exists, all blanks will be returned.

SEE ALSO

getenv(3C), environ(5)

NAME

gethostid – get host processor identification number

SYNOPSIS

```
EXTERNAL GETHOSTID
INTEGER GETHOSTID
INTEGER ARR(4)
CALL GETHOSTID(ARR)
```

DESCRIPTION

The *GETHOSTID* subroutine fills the four element integer array *ARR* with the unique part of each processor's PROM ID number. If there are fewer than 4 processors, the missing processor numbers are zero.

FILES

/usr/lib/libuF77.a

NAME

getlog – get user's login name

SYNOPSIS

```
EXTERNAL GETLOG
CHARACTER*M NAME
CALL GETLOG(NAME)
```

DESCRIPTION

GETLOG returns the user's login name in the argument *NAME* or all blanks if the process is running detached from a terminal.

FILES

/usr/lib/libuF77.a

SEE ALSO

getlogin(3)

NAME

getpid – get process id

SYNOPSIS

```
EXTERNAL GETPID
INTEGER GETPID
INTEGER PID
PID = GETPID()
```

DESCRIPTION

GETPID returns the process ID number of the current process.

FILES

/usr/lib/libuF77.a

SEE ALSO

getpid(2)

NAME

getuid, getgid – get user or group ID of the caller

SYNOPSIS

```
EXTERNAL GETUID
INTEGER GETUID
INTEGER UID
UID = GETUID()

EXTERNAL GETGID
INTEGER GETGID
INTEGER GID
GID = GETGID()
```

DESCRIPTION

These functions return the real user or group ID of the user of the process.

FILES

/usr/lib/libuF77.a

SEE ALSO

getuid(2)

NAME

hostnm – get name of current host

SYNOPSIS

```
EXTERNAL HOSTNM
INTEGER HOSTNM
INTEGER HN
CHARACTER*N NAME
HN = HOSTNM(NAME)
```

DESCRIPTION

This function puts the name of the current host into character string *NAME*. The return value should be 0; any other value indicates an error.

FILES

/usr/lib/libuF77.a

SEE ALSO

gethostname(2)

NAME

iargc – return the number of command line arguments

SYNOPSIS

```
EXTERNAL IARGC
INTEGER IARGC
INTEGER I
I = IARGC()
```

DESCRIPTION

The *IARGC* function returns the number of command line arguments passed to the program. Thus, if a program were invoked via

```
foo arg1 arg2 arg3
```

IARGC( ) would return 3.

SEE ALSO

GETARG(3F)

NAME

idate, itime – return date or time in numerical form

SYNOPSIS

```
EXTERNAL IDATE
INTEGER ARRAY IRRAY(3)
CALL IDATE(IRRAY)

EXTERNAL ITIME
CALL ITIME(IRRAY)
```

DESCRIPTION

IDATE returns the current date in *IRRAY*. The order is: day, month, year. Month will be in the range 1-12. Year will be ≥ 1970.

ITIME returns the current time in *IRRAY*. The order is: hour, minute, second.

NOTES

To use the *IDATE* routine, you must explicitly include */usr/lib/fortran/idate.o* on your compilation or load line as this BSD routine conflicts with another routine normally supplied by Stardent 1500/3000 Fortran.

FILES

/usr/lib/libuF77.a /usr/lib/fortran/idate.o

SEE ALSO

CTIME(3F), FDATE(3F)

NAME

kill – send a signal to a process

SYNOPSIS

```
EXTERNAL KILL
INTEGER KILL
INTEGER SUCC
INTEGER PID
INTEGER SIGNUM
SUCC = KILL(PID, SIGNUM)
```

DESCRIPTION

KILL will send signal SIGNUM to process PID. *PID* must be the process id of one of the user's processes. *SIGNUM* must be a valid signal number (see sigvec(2)). The returned value will be 0 if successful; an error code otherwise.

FILES

/usr/lib/libuF77.a

SEE ALSO

kill(2), sigvec(2), SIGNAL(3F), FORK(3F), PERROR(3F)

NAME

link – make a link to an existing file

SYNOPSIS

```
EXTERNAL LINK
INTEGER LINK
CHARACTER*N NAME1, NAME2
INTEGER SUCC
SUCC = LINK(NAME1, NAME2)

INTEGER LINK
INTEGER SYMLNK
CHARACTER*N NAME1, NAME2
INTEGER SUCC
SUCC = SYMLNK(NAME1, NAME2)
```

DESCRIPTION

LINK creates a link between file *NAME1* and file *NAME2*. *NAME1* must be the pathname of an existing file. *NAME2* is a pathname to be linked to file *NAME1*. *NAME2* must not already exist. The returned value will be 0 if successful; a system error code otherwise.

SYMLNK creates a symbolic link to *NAME1*.

FILES

/usr/lib/libuF77.a

SEE ALSO

link(2), symlnk(2), PERROR(3F), UNLINK(3F)

BUGS

Pathnames can be no longer than PATH_MAX characters as defined in <*limits.h*>.

NAME

loc – return the address of an object

SYNOPSIS

```
EXTERNAL LOC
INTEGER LOC
INTEGER WHERE
CHARACTER*N ARG
WHERE = LOC(ARG)
```

DESCRIPTION

The returned value will be the address of *ARG*. *ARG* may be of any data type.

FILES

/usr/lib/libuF77.a

NAME

long, short – integer object conversion

SYNOPSIS

```
EXTERNAL LONG
INTEGER LONG
INTEGER*2 INT2
INTEGER LI
LI = LONG(INT2)

EXTERNAL SHORT
INTEGER SHORT
INTEGER*4 INT4
INTEGER*2 SI
SI = SHORT(INT4)
```

DESCRIPTION

These functions provide conversion between short and long integer objects. *LONG* is useful when constants are used in calls to library routines and the code is to be compiled with "-i2". *SHORT* is useful in similar context when an otherwise long object must be passed as a short integer.

FILES

/usr/lib/libuF77.a

NAME

malloc, free, falloc – dynamic memory allocator

SYNOPSIS

```
EXTERNAL MALLOC
INTEGER SIZE, ADDR
CALL MALLOC (SIZE, ADDR)

EXTERNAL FREE
INTEGER ADDR
CALL FREE (ADDR)

EXTERNAL FALLOC
INTEGER NELEM, ELSIZE, CLEAN, ADDR, OFFSET
DIMENSION BASEVEC(1)
CALL FALLOC (NELEM, ELSIZE, CLEAN, BASEVEC, ADDR, OFFSET)
```

DESCRIPTION

MALLOC, *FALLOC*, and *FREE* provide a general-purpose memory allocation package. *MALLOC* returns in *ADDR* the address of a block of at least *SIZE* bytes beginning on an even-byte boundary.

FALLOC allocates space for an array of *NELEM* elements of size *ELSIZE* and returns the address of the block in *ADDR*. It zeros the block if *CLEAN* has the value 1. It returns in *OFFSET* an index such that the storage may be addressed as *BASEVEC(OFFSET+1) ... BASEVEC(OFFSET+NELEM)*. *FALLOC* gets extra bytes so that after address arithmetic, all the objects so addressed are within the block. *BASEVEC* should be declared to be a type whose size is the same as the value of *ELSIZE*.

The argument to *FREE* is the address of a block previously allocated by *MALLOC* or *FALLOC*; this space is made available for further allocation. To free blocks allocated by *FALLOC*, use *ADDR* in calls to *FREE*; do not use *BASEVEC(OFFSET+1)*.

Needless to say, grave disorder will result if the space assigned by *MALLOC* or *FALLOC* is overrun or if some random number is handed to *FREE*.

DIAGNOSTICS

MALLOC and *FALLOC* set *ADDR* to 0 if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block.

The following example shows how to obtain memory and use it within a subprogram:

```
    CALL FALLOC ( N, 4, 0, WORK, ADDR, OFFSET )
    DO 10 I = 1, N
    WORK(OFFSET+I) = ...
10  CONTINUE
```

FILES

/usr/lib/libuF77.a

SEE ALSO

malloc(3)

NAME

mclock – return Fortran time accounting

SYNOPSIS

```
EXTERNAL MCLOCK
INTEGER MCLOCK
INTEGER I
I = MCLOCK()
```

DESCRIPTION

MCLOCK returns time accounting information about the current process and its child processes. The value returned is the sum of the current process's user time and the user and system times of all child processes.

SEE ALSO

times(2), clock(3C), SYSTEM(3F)

NAME

perror, gerror, ierrno – get system error messages

SYNOPSIS

```
EXTERNAL PERROR
CHARACTER*N STRING
CALL PERROR(STRING)

EXTERNAL GERROR
CHARACTER*M GERROR
CHARACTER*J ANSWER
CALL GERROR(STRING)
ANSWER = GERROR(STRING)

EXTERNAL IERRNO
INTEGER IERRNO
INTEGER ERR
ERR = IERRNO()
```

DESCRIPTION

PERROR will write a message to Fortran logical unit 0 appropriate to the last detected system error. *STRING* will be written preceding the standard error message.

GERROR returns the system error message in character variable *STRING*. *GERROR* may be called either as a subroutine or as a function.

IERRNO will return the error number of the last detected system error. This number is updated only when an error actually occurs. Most routines and I/O statements that might generate such errors return an error code after the call; that value is a more reliable indicator of what caused the error condition.

FILES

/usr/lib/libuF77.a

BUGS

The length of the string returned by *GERROR* is determined by the calling program.

NAME

putc, fputc – write a character to a fortran logical unit

SYNOPSIS

```
EXTERNAL PUTC
INTEGER PUTC
INTEGER LUNIT
CHARACTER*1 CHAR
INTEGER SUCC
SUCC = PUTC(LUNIT, CHAR)

INTEGER PUTC
INTEGER FPUTC
INTEGER LUNIT
CHARACTER*1 CHAR
INTEGER SUCC
SUCC = FPUTC(LIUNT, CHAR)
```

DESCRIPTION

These functions write a character to the file associated with a Fortran logical unit bypassing normal Fortran I/O. *PUTC* writes to logical unit 6, normally connected to the control terminal output.

The value of each function will be zero unless some error occurred; a system error code otherwise. See PERROR(3F).

FILES

/usr/lib/libuF77.a

SEE ALSO

putc(3S), intro(2), PERROR(3F)

NAME

qsort – quick sort

SYNOPSIS

```
SUBROUTINE QSORT
EXTERNAL COMPAR
INTEGER*2 COMPAR
INTEGER ARRAY, LEN, ISIZE
CALL QSORT(ARRAY, LEN, ISIZE, COMPAR)
```

DESCRIPTION

One dimensional *ARRAY* contains the elements to be sorted. *LEN* is the number of elements in the array. *ISIZE* is the size of an element:

4 for **integer** and **real**
8 for **double precision** or **complex**
16 for **double complex**
(length of character object) for **character** arrays

COMPAR is the name of a user supplied INTEGER*2 function that will determine the sorting order. This function will be called with 2 arguments that will be elements of *ARRAY*. The function must return :

negative if arg 1 is considered to precede arg 2
zero if arg 1 is equivalent to arg 2
positive if arg 1 is considered to follow arg 2

On return, the elements of *ARRAY* will be sorted.

FILES

/usr/lib/libuF77.a

SEE ALSO

qsort(3)

NAME

rand, irand, srand – random number generator

SYNOPSIS

```
EXTERNAL RAND
DOUBLE PRECISION RAND
DOUBLE PRECISION X
X = RAND()

EXTERNAL IRAND
INTEGER IRAND
INTEGER I
I = IRAND()

EXTERNAL SRAND
INTEGER SEED
CALL SRAND(SEED)
```

DESCRIPTION

The integer function *IRAND* generates successive pseudo-random integers in the closed interval [0, 32,767].

The double precision function *RAND* generates successive pseudo-random double precision values in the half-open interval [0, 1.0).

The procedure *SRAND* uses its integer valued argument to re-initialize the seed used by the functions *IRAND* and *RAND*.

NOTES

These functions do not generate very high quality random numbers. *DRAND48*(3F) provides a much better, though more elaborate, random-number generator.

SEE ALSO

rand(3C), DRAND48(3F)

NAME

rename – rename a file

SYNOPSIS

```
EXTERNAL RENAME
INTEGER RENAME
INTEGER SUCC
CHARACTER*N FROM
CHARACTER*M TO
SUCC = RENAME(FROM, TO)
```

DESCRIPTION

FROM must be the pathname of an existing file. *TO* will become the new pathname for the file. If *TO* exists, then both *FROM* and *TO* must be the same type of file and must reside on the same filesystem. If *TO* exists, it will be removed first.

The returned value will be 0 if successful; a system error code otherwise.

FILES

/usr/lib/libuF77.a

SEE ALSO

rename(2), PERROR(3F)

BUGS

Pathnames can be no longer than PATH_MAX characters as defined in <*limits.h*>.

NAME

rindex, lnblnk – tell about character objects

SYNOPSIS

```
EXTERNAL RINDEX
INTEGER RINDEX
CHARACTER*N STRING
CHARACTER*M SUBSTR
INTEGER RSTRIN
RSTRIN = RINDEX(STRING, SUBSTR)

EXTERNAL LNBLNK
INTEGER LNBLNK
CHARACTER*N STRING
CHARACTER*M SUBSTR
INTEGER LASTCHAR
LASTCHAR = LNBLNK(STRING)
```

DESCRIPTION

RINDEX returns the index of the last occurrence of the substring *SUBSTR* in *STRING* or zero if it does not occur.

LNBLNK returns the index of the last non-blank character in *STRING*. This is useful because all Fortran character objects are fixed length and are often blank padded.

FILES

/usr/lib/libuF77.a

NAME

signal – specify Fortran action on receipt of a system signal

SYNOPSIS

```
SUBROUTINE SIGNAL
EXTERNAL INTFC
INTEGER I, INTFC
INTEGER SIGNAL
CALL SIGNAL(I,INTFC)
```

DESCRIPTION

The argument *I* specifies the signal to be caught. *SIGNAL* allows a process to specify a function to be invoked upon receipt of a specific signal. The interrupt processing function, *INTFC*, does not take an argument. The first argument specifies which fault or exception. The second argument specifies the function to be invoked.

SEE ALSO

kill(2), signal(2)

NAME

sleep – suspend execution for an interval

SYNOPSIS

```
SUBROUTINE SLEEP
INTEGER ITIME
CALL SLEEP(ITIME)
```

DESCRIPTION

SLEEP causes the calling process to be suspended for *ITIME* seconds. The actual time can be up to 1 second less than *ITIME* due to granularity in system timekeeping.

FILES

/usr/lib/libuF77.a

SEE ALSO

sleep(3)

NAME

stat, lstat, fstat – get file status

SYNOPSIS

```
EXTERNAL STAT
EXTERNAL LSTAT
CHARACTER*N NAME
INTEGER STATB(12)
CALL STAT(NAME, STATB)

EXTERNAL LSTAT
INTEGER LSTAT
CHARACTER*N NAME
INTEGER STATB(12)
INTEGER IERR
IERR = LSTAT(NAME, STATB)

EXTERNAL FSTAT
INTEGER FSTAT
INTEGER UNTI
CHARACTER*N NAME
INTEGER IERR
IERR = FSTAT(UNIT, STATB)
```

DESCRIPTION

These routines return detailed information about a file. *STAT* and *LSTAT* return information about file *NAME*; *FSTAT* returns information about the file associated with Fortran logical unit *LUNIT*. The order and meaning of the information returned in array *STATB* is as described for the structure *stat* under *stat(2)*. The "spare" values are not included.

The value of either function will be zero if successful; an error code otherwise.

FILES

/usr/lib/libuF77.a

SEE ALSO

stat(2), ACCESS(3F), PERROR(3F), TIME(3F)

BUGS

Pathnames can be no longer than PATH_MAX characters as defined in <*limits.h*>.

NAME

system – issue a shell command from Fortran

SYNOPSIS

```
SUBROUTINE SYSTEM
CHARACTER*N STRING
CALL SYSTEM(STRING)
```

DESCRIPTION

SYSTEM causes its character argument to be given to *sh(1)* as input as if the string had been typed at a terminal. The current process waits until the shell has completed.

SEE ALSO

exec(2), system(3S)
sh(1) in the *Commands Reference Manual.*

NAME

time, ctime, ltime, gmtime – return system time

SYNOPSIS

```
EXTERNAL TIME
INTEGER TIME
INTEGER T
T = TIME()

EXTERNAL CTIME
CHARACTER*N CTIME
INTEGER STIME
CHARACTER*24 RTIME
RTIME = CTIME(STIME)

EXTERNAL LTIME
INTEGER STIME
CHARACTER TARRAY(9)
CALL LTIME(STIME, TARRAY)

EXTERNAL GMTIME
INTEGER STIME
CHARACTER TARRAY(9)
CALL GMTIME(STIME, TARRAY)
```

DESCRIPTION

TIME returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds. This is the value of the UNIX system clock.

CTIME converts a system time to a 24 character ASCII string. The format is described under *ctime(3)*. No "newline" or NULL will be included.

LTIME and *GMTIME* dissect a UNIX time into month, day, and so on, either for the local time zone or as GMT. The order and meaning of each element returned in *TARRAY* is described under *ctime(3)*.

NOTES

To use the *TIME* routine, you must explicitly include */usr/lib/fortran/time.o* on your compilation or load line as this BSD routine conflicts with another routine normally supplied by Stardent 1500/3000 Fortran.

FILES

/usr/lib/libuF77.a /usr/lib/fortran/time.o

SEE ALSO

ctime(3), ITIME(3F), IDATE(3F), FDATE(3F)

NAME

topen,tclose,tread,twrite,trewin,tskipf,tstate – Fortran tape I/O

SYNOPSIS

```
EXTERNAL TOPEN
INTEGER TOPEN, TLU, RETVAL
CHARACTER*M DEVNAM
LOGICAL LABEL
RETVAL = TOPEN(TLU, DEVNAM, LABEL)

EXTERNAL TCLOSE
INTEGER TCLOSE, TLU, RETVAL
INTEGER TLU
RETVAL = TCLOSE(TLU)

EXTERNAL TREAD
INTEGER TREAD, TLU, RETVAL
CHARACTER*M BUFFER
RETVAL = TREAD(TLU, BUFFER)

EXTERNAL TWRITE
INTEGER TWRITE, TLU, RETVAL
CHARACTER*M BUFFER
RETVAL = TWRITE(TLU, BUFFER)

EXTERNAL TREWIN
INTEGER TREWIN, TLU, RETVAL
RETVAL = TREWIN(TLU)

EXTERNAL TSKIPF
INTEGER TSKIPF, TLU, NFILES, NRECS, RETVAL
RETVAL = TSKIPF(TLU, NFILES,NRECS)

EXTERNAL TSTATE
INTEGER TSTATE, TLU, FNUM, RNUM, TCSR,
LOGICAL ERRF, EOFF, EOTF
RETVAL = TSTATE(TLU, FNUM, RNUM, ERRF, EOFF, EOTF, TCSR)
```

DESCRIPTION

These functions provide a simple interface between Fortran and magnetic tape devices. A "tape logical unit", *TLU*, is "TOPEN"ed in much the same way as a normal Fortran logical unit is "open"ed. All other operations are performed via the *TLU*. The *TLU* has no relationship at all to any normal Fortran logical unit.

TOPEN associates a device name with a *TLU*. *TLU* must be in the range 0 to 3. The logical argument *LABEL* should indicate whether the tape includes a tape label. This is used by *TREWIN* below. *TOPEN* does not move the tape. The normal returned value is 0. If the value of the function is negative, an error has occured. See *PERROR*(3F) for details.

TCLOSE closes the tape device channel and removes its association with *TLU*. The normal returned value is 0. A negative value indicates an error.

TREAD reads the next physical record from tape to *BUFFER*. *BUFFER* **must** be of type **character**. The size of *BUFFER* when using 1/4" cartridge tape must be a multiple of 512 bytes. The return value will be the number of bytes read plus any necessary blanks needed to round to a 512 byte multiple. For use with a 9 track tape,

BUFFER should be large enough to hold the largest physical record to be read. The actual number of bytes read will be returned as the value of the function. If the value is 0, the end-of-file has been detected. A negative value indicates an error.

TWRITE writes a physical record to tape from *BUFFER*. The physical record length will be the size of *BUFFER* when using a 9 track tape. The number of bytes written will be returned. *BUFFER* must be a mutiple of 512 when using a 1/4" cartridge tape. The number of bytes written plus any necessary blank padding to round to a 512 byte multiple will be returned. *BUFFER* **must** be of type **character**. A return value of 0 or negative indicates an error.

TREWIN rewinds the tape associated with *TLU* to the beginning of the first data file. If the tape is a labelled tape (see *TOPEN* above) then the label is skipped over after rewinding. The normal returned value is 0. A negative value indicates an error.

TSKIPF allows the user to skip over files and/or records. First, *NFILES* end-of-file marks are skipped. If the current file is at EOF, this counts as one file to skip. (Note: This is the way to reset the EOF status for a *TLU*.) Next, *NRECS* physical records are skipped over. The normal returned value is 0. A negative value indicates an error.

Finally, *TSTATE* allows the user to determine the logical state of the tape I/O channel and to see the tape drive control status register. The values of *FNUM* and *RNUM* will be returned and indicate the current file and record number. The logical values *ERRF*, *EOFF*, and *EOTF* indicate an error has occurred, the current file is at EOF, or the tape has reached logical end-of-tape. End-of-tape (EOT) is indicated by an empty file, often referred to as a double EOF mark. It is not allowed to read past EOT although it is allowed to write. The value of *TCSR* will reflect the tape drive control status register. See *ht*(4) for details.

FILES

/usr/lib/libuF77.a

SEE ALSO

ht(4), PERROR(3F), rewind(1), mtio(4)

NOTES

The **QIC-11** and **QIC-24** 1/4 inch streaming tape specifications do not allow the updating of individual blocks on the tape. This is designed to protect the host from attempting to do a write in the middle of reading data or a read in the middle of writing data. Another of the limitations of this device is that the only way to append to the tape is to attempt a TREAD past the last block on the tape and then TWRITE. Trying to TREAD from this tape position is an error. Immediately after rewinding the tape to the beginning (TREWIN), it is possible to either TREAD or TWRITE.

NAME

ttynam, isatty – find name of a terminal port

SYNOPSIS

```
EXTERNAL TTYNAM
CHARACTER* TTYNAM
INTEGER LUNIT
CHARACTER*TNAME
TNAME = TTYNAM(LUNIT)

EXTERNAL ISATTY
LOGICAL ISATTY
INTEGER LUNIT
LOGICAL ISASSOC
ISASSOC = ISATTY(LUNIT)
```

DESCRIPTION

TTYNAM returns a blank padded path name of the terminal device associated with logical unit *LUNIT*.

ISATTY returns **.true.** if *LUNIT* is associated with a terminal device, **.false.** otherwise.

FILES

/dev/*
/usr/lib/libuF77.a

DIAGNOSTICS

TTYNAM returns an empty string (all blanks) if *LUNIT* is not associated with a terminal device in directory */dev*.

NAME

unlink – remove a directory entry

SYNOPSIS

```
EXTERNAL UNLINK
INTEGER UNLINK
CHARACTER*N NAME
INTEGER SUCC
SUCC = UNLINK(NAME)
```

DESCRIPTION

UNLINK causes the directory entry specified by pathname *NAME* to be removed. If this was the last link to the file, the contents of the file are lost. The returned value will be zero if successful; a system error code otherwise.

FILES

/usr/lib/libuF77.a

SEE ALSO

unlink(2), LINK(3F), filsys(5), PERROR(3F)

BUGS

Pathnames can be no longer than PATH_MAX characters as defined in <*limits.h*>.

ERROR MESSAGES

APPENDIX A

A.1 I/O Error Messages

This appendix contains a list of error messages that are provided to you during the compilation or the execution of Fortran programs.

A.1.1 General Information on the I/O Error Messages

When an I/O error occurs, the following information is reported:

- The Fortran I/O Library Error code (number)
- A description of the error code (message)
- A summary of the apparent I/O state
  - A line number and source file name to identify where the incorrect operation was being performed
  - A unit number and a file type
  - The possible operation or call that was being attempted during the error
  - The version number of the library and last date of a source change
  - The last system error (although not relevant!) that occurred

The error messages have this property:

$0 < Number$

where *Number* is the code number that is greater than 0 (zero) and specifies an index to the table of messages. The text of the error messages is held in this table. There are some reserved messages. For instance,

- 0: a message that says messages are garbled.
- 1: a message announcing general purpose allocation failures.
- 2: a message announcing a second entrance to the library.

A.1.2 Fortran I/O Error Message Library

This section contains the error messages that are generated from the Fortran I/O library. Some messages are self-explanatory, but some need further explanations. Whenever possible, we provide you hints and advice (in *italicized* form) on how to anticipate possible resolutions for errors in your code.

0 Error message garbled--internal library error

1 Memory allocation failure--allocation routine returned NULL

2 Second I/O statement attempted while first statement still active

3 BACKSPACE must refer to connected unit

You probably forget to include the unit arithmetic expression (0 or positive). Remember that BACKSPACE is allowed only for sequential files. Refer to the BACKSPACE statement in the Fortran Reference Manual in Chapter 2, Fortran Statements for detailed information on the BACKSPACE statement.

4 Positioning required for BACKSPACE does not work on this file

Your file is probably anything other than a sequential file. BACKSPACE works only with sequential files.

5 Cannot BACKSPACE over a list-directed record

Again, the BACKSPACE statement works only with sequential files. You cannot execute a direct access on records with BACKSPACE.

6 VMS style indexed files are not implemented

7 Library call fseek() failed during BACKSPACE statement

Did you make an appropriate library call? Is your path correct? Did you make a proper connection to the appropriate unit?

8 Associated variable updated by BACKSPACE is not an integer type

Your associated variable for BACKSPACE is probably anything other than an integer type. Check your BACKSPACE variable.

9 Library call fread() failed during BACKSPACE statement

Did you make an appropriate library call? Is your path correct? Did you specify the appropriate unit, ios, or label parameter?

10 BACKSPACE failed on this file for unknown reasons

Verify the syntax on your BACKSPACE statement line, and try to execute the program again.

11 Unit number cannot be less than zero nor greater than 2,147,483,637

Your unit number is probably a negative number or any other number rather than a positive number between zero and 2,147,483,637, inclusively. Check the unit number again to make sure it is a valid number.

12 Formatted file required and file attached to unit is not

Generally speaking, the file attached to a unit should be a formatted file. Your file attached to the unit is not a formatted file. You should convert the file to a formatted one first before attaching it to the unit. Your can use the FORMAT statement to perform the conversion. Refer to the FORMAT statement in the Fortran Reference Manual in Chapter 2, Fortran Statements for details on how to convert your file.

13 Sequential file required and file attached to unit is not

You must have a sequential file to perform this operation. Make sure you have ACCESS=SEQUENTIAL on the OPEN statement.

14*

15 Unformatted file required and file attached to unit is not

To perform this operation you must have an unformatted file attached to the specifying unit. Verify your OPEN statement parameters.

16 Positioning necessary for unformatted I/O will not work on this file

You probably tried to perform sequential unformatted I/O on a direct access file (i.e. a direct random access device). If you use the BACKSPACE statement to make the file pointer move to a certain position, remember that this only works in sequential files.

17 CLOSE statement STATUS keyword does not have a legal value

The STATUS keyword should be either KEEP or DELETE. Refer to the CLOSE statement in the Fortran Reference Manual in Chapter 2, Fortran Statements for detailed information on the STATUS keyword.

18 Scratch files cannot be kept after they are closed

Scratch files cannot be saved after they are closed, and if you want to save them, you have to specify them as 'NEW' on the OPEN statement.

* This error message code number is currently unused.

19 Library call fclose() failed during CLOSE statement

Did you make an appropriate library call? Is your path correct? Verify the syntax of the CLOSE statement. Refer to the CLOSE statement in the Fortran Reference Manual in Chapter 2, Fortran Statements for detailed information on the CLOSE statement.

20 Unlink() call failure during CLOSE probably left scratch file linked

Did you make an appropriate library call? Is your path correct? Verify the syntax of the CLOSE statement.

21 Illegal internal code; suspect compiler/library error or overwritten memory

22 REAL*16 data type and arithmetic are not supported

*Use REAL*8 (double precision) or COMPLEX*16 instead.*

23 DELETE must refer to a connected unit

You must specify an appropriate unit before attempting to delete a file.

24 DELETE applies only to relative and indexed files

Refer to the OPEN statement in the Fortran Reference Manual in Chapter 2, Fortran Statements for additional information on the DELETE parameter (DELETE is one of those parameters of DISPOSE).

25 Record number for DELETE must be positive

26 DELETE statements are not implemented

27 ENDFILE statement to open unit failed

Is your path correct? Did you specify an appropriate unit number on the ENDFILE statement? Refer to the ENDFILE statement in the Fortran Reference Manual in Chapter 2, Fortran Statements for further details on the ENDFILE statement.

28 ENDFILE may not be applied to indexed files

You can use the ENDFILE statement only with sequential files or devices.

29 Compatibility problem; VMS does not allow ENDFILE on direct access files

30 String containing format stopped before end of format found

31 Only blanks may precede opening '(' in format supplied as string

32 P format must be followed by comma, ')', F, E, D, or G format item

33 Repeat count may not precede /, $, :, '...', T, TL, TR, S, SP, SS, BN, BZ, Q format items

34 A comma is missing after a format item

35 Only optional + or - followed by decimal digits allowed in P format item

36 P format item must be preceded by a number

37 Only BN and BZ format items may begin with B

38 T, TL, or TR format code must be followed by a number

39 An H format code must be preceded by a number

40 I, O, or Z format code with a decimal point must be followed by a number

41 F and D formats must have no number or w.d specification

42 E and G formats must have no number, w.d, or w.dEe

43 Compiled format is garbled; suspect compiler/library error or overwritten memory

44 Repeat counts must be greater than zero

45 Format field specification value must be greater than zero

46 Closing parenthesis seen for second time while looking for data transmission item

47 Format stack overflow; use fewer nested parentheses in your format

48 Backspacing is not allowed on a TTY-like file

49 Library call fseek() failed during attempt to read characters from file

Did you make an appropriate library call? Is your path correct? Check to see if the named file exists.

50 End of record encountered unexpectedly during read

The length of the record you attempted to read is probably smaller than the one you specified on the READ statement. Make sure that you do not attempt to read more than what was written. Verify your RECL parameter.

51 Library call fseek() failed during attempt to write characters to file

Did you make an appropriate library call? Is your path correct? Does the specified file exist?

52 End of file encountered while attempting to read from file

Normally, a sequential written file is terminated with an end-of-file flag. You probably attempted to perform a direct access read on a sequential written file.

53 EXTENSION: Blank supplied to A format during read when end of record encountered

54 Attempt to write beyond length of internal record is illegal

55 Attempt to read beyond last record of internal file is illegal

56 Attempt to write beyond last record of internal file is illegal

57 FORTRAN string is too short to accept value assigned; value will be truncated

58 FORnnn.DAT file could not be opened; possible internal library failure

Did you make an appropriate library call? Is your path correct? Does the named file exist?

59 Allocation of internal unit descriptors failed in OPEN during attempt to connect unit

Did you specify an appropriate unit number? Did you have the correct path? Does the named file exist?

60 OPEN statement ACCESS keyword does not have a legal value

The ACCESS keyword should be either DIRECT, ASYNC or SEQUENTIAL. Refer to the OPEN statement in the Fortran Reference Manual in Chapter 2, Fortran Statements for detailed information on the ACCESS keyword.

61 OPEN statement FORM keyword does not have a legal value

The FORM keyword should be either FORMATTED or UNFORMATTED. Refer to the OPEN statement in the Fortran Reference Manual in Chapter 2, Fortran Statements for detailed information on the FORM keyword.

62 OPEN statement ORGANIZATION keyword does not have a legal value

The ORGANIZATION keyword should either SEQUENTIAL, INDEXED or RELATIVE. Refer to the OPEN statement in the Fortran Reference Manual in Chapter 2, Fortran Statements for detailed information on the ORGANIZATION keyword.

63 OPEN statement RECORDTYPE keyword does not have a legal value

The RECORDTYPE keyword should be either FIXED, VARIABLE, SEGMENTED, STREAM, STREAM_CR, or STREAM_LF. Refer to the OPEN statement in the Fortran Reference Manual in Chapter 2, Fortran Statements for detailed information on the RECORDTYPE keyword.

64 OPEN statement BLANK keyword only allowed on formatted files

65 OPEN statement BLANK keyword does not have a legal value

The BLANK keyword should be either NULL or ZERO. Refer to the OPEN statement in the Fortran Reference Manual in Chapter 2, Fortran Statements for detailed information on the BLANK keyword.

66 OPEN statement CARRIAGECONTROL keyword does not have a legal value

The CARRIAGECONTROL keyword should be either FORTRAN, LIST, or NONE. Refer to the OPEN statement in the Fortran Reference Manual in Chapter 2, Fortran Statements for detailed information on the CARRIAGECONTROL keyword.

67 OPEN statement DISPOSE keyword does not have a legal value

The DISPOSE keyword should be either DELETE, KEEP, PRINT, PRINT/DELETE, SAVE, SUBMIT, or SUBMIT/DELETE. Refer to the OPEN statement in the Fortran Reference Manual in Chapter 2, Fortran Statements for detailed information on the DISPOSE keyword.

68 OPEN statement STATUS keyword does not have a legal value

The STATUS keyword should be either OLD, NEW, SCRATCH, or UNKNOWN. Refer to the OPEN statement in the Fortran Reference Manual in Chapter 2, Fortran Statements for detailed information on the STATUS keyword.

69 File to be opened does not exist

Does the file you want to open already exist? Check the name of the file again.

70 A filename cannot be specified when file status is SCRATCH

71 File status was not set in preparation for opening; possible library error

72 In an attempt to change a file with read status to write status, fseek() failed

Did you make an appropriate library call? Is your path correct? Does the named file exist?

73 OPEN statement record length must be greater than zero

74 OPEN statement record length must equal zero

75 Direct access is not allowed on indexed files

76 In an attempt to change a file with read status to write status, freopen() failed

Did you make an appropriate library call? Is your path correct? Does the named file exist?

77 File and unit properties specified for OPEN are inconsistent

Verify the parameters you specify on the OPEN statement. Refer to the OPEN statement in the Fortran Reference Manual in Chapter 2, Fortran Statements for additional information on the UNIT and FILE values.

78 APPEND was specified, but cannot move to the end of this file. Are you trying to APPEND stdout or stderr?

79 Library call fseek() failed during attempt to move to end of file. Are you trying to move to the end of stdout or stderr?

80 Encountered end of record unexpectedly while reading under A format

81 When reading F, E, D, or G format, data item must be real or complex type

82 Unexpected character in input while reading under F, E, D, or G format

Data item must be real or complex type when reading F, E, D, or G format.

83 Unexpected decimal point in input while reading under F, E, D, or G format

Data item must be real or complex type when reading F, E, D, or G format.

84 Unexpected sign in exponent while reading under F, E, D, or G format

Data item must be real or complex type when reading F, E, D, or G format.

85 Encountered end of file unexpectedly while reading under F format

86 In Tn format item, the value of n must be greater than zero

87 Illegal internal format code; suspect compiler/library error or overwritten memory

88 Encountered end of record unexpectedly while reading Hollerith format item

89 Reads into Hollerith format items are illegal in Fortran 77

90 When reading under I format, data item must be integer or logical type

91 An extra sign or a sign after digits appeared while reading integer value

92 Character that is not a digit in the radix used by format item appeared

93 VMS style early field termination is not legal in Fortran 77

94 Unexpected end of record found; short field termination not allowed for L formats

95 Encountered end of file unexpectedly while reading under L format

96 T, t, F, or f must immediately follow period when reading under L format

97 First non-blank must be T, t, F, f, or period when reading under L format

98 When reading under L format, data item must be integer or logical type

99 Only character data may be read into character data items

100 Unexpected character began data item

101 Only one comma allowed in complex value

102 Closing parenthesis found before imaginary part of complex value

103 Comma is required to separate parts of complex value

104 Only white space may precede closing parenthesis of complex value

105 Character data may not be read into non-character data items

106 Unexpected end of file found while reading data value

107 Unit must have fixed length records

108 Library call fseek() failed positioning file for read

Is your path correct? Does the named file exist?

109 Read attempted beyond length of record

The length of the record you attempted to read is probably smaller than the one you specified on the READ statement. Adjust the record length you assigned on the READ statement.

110 During transfer of data, fread() was unable to read the amount of data requested

Is your path correct? Is the data you are attempting to read exhausted? Does the named file exist?

111 Cannot read record length from variable length record

112 Unexpected eof encountered while reading length of variable length record

113 A REWIND statement on a non-existant file dictates that the file is opened, connected, and then rewound. This attempt to open the file failed

Is the file attempted to be opened a direct access file? If it is a direct access file, you cannot perform this operation. Refer to the REWIND statement in the Fortran Reference Manual in Chapter 2, Fortran Statements for further details on the REWIND statement.

114 REWIND statement may only be applied to sequential files

115 Positioning required for REWIND does not work on this file

Again, the REWIND statement works only on sequential files.

116 In an attempt to change a file with write status to read status, freopen() failed

Is your path correct? Does the named file exist?

117 In an attempt to change a file with write status to read status, fseek() failed

Is your path correct? Does the named file exist?

118 Unit was specified as readonly

119 In an attempt to truncate a file, library call ftruncate() failed.

Is your path correct? Does the named file exist?

120 Library call fseek() failed positioning file for write

Is your path correct? Does the named file exist?

121 Write attempted of more bytes than length of record allows

The record length you specify on the WRITE statement is larger than the record length specified in the file. Reduce the record length specified on the WRITE statement before attempting to execute the program again.

122 Library call fwrite() failed when moving data to file

Did you make an appropriate library call? Is your path correct? Does the named file exist?

123 Library call fseek() failed while moving to end of record after write

124 Library call fwrite() failed while writing control information

125 When writing under I format, data item must be integer or logical type

126 When writing Iw.d format, d must be no more than w

127 When writing under L format, data item must be integer or logical type

128 Output record too short for list directed output value

129 When writing Zw.d format, d must be no more than w

130 Scale factor must be in range <-d+1, d+1> when writing E or D format

131 Pointer to file name is null; probably compiler error

132 Encountered end of record unexpectedly while reading under Z format

133 Only hexadecimal digits allowed under Z format

134 Value read under Z format has more nonzero bits than data item can hold

Only hexadecimal digits allowed under Z format.

135 Encountered end of record unexpectedly while reading under O format

136 Only octal digits allowed under O format

137 Value read under O format has more nonzero bits than data item can hold

Only octal digits allowed under O format.

138 End of file not caught by END= clause turned into error.

139 Second character in namelist input must be $ or &

140 Variable name on input record not found in namelist group

141 Namelist group name on input record not found

142 Cannot use substring or subscript on non-string or non-array in namelist

143 Cannot use substring if namelist element is not of type character

144 A character array cannot have a substring without a subscript in namelist input

145 The namelist input substring value is larger than the string length

146 Trying to assign to an index beyond array bounds in namelist input

147 In namelist input, a comma has been seen when reading a one dimensional array

148 The namelist array subscript is not within the array bounds

149 The count of subscript specifiers is greater than the number of array dimensions in namelist input

150 Illegal right paren in namelist input

151 No nested parenthesis in namelist input

152 Left parenthesis seen between element name and assignment operator

153 Unrecognized or illegal character in namelist input

154 Illegal character seen before the assignment operator in namelist input

155 Warning: Illegal character in namelist $end - Ignored

156 Warning: End symbol mismatch in namelist input - Ignored

157 Warning: VMS does not allow the use of the ENDFILE statement on files opened for direct access

It is practical to insert an endfile record at the end of all sequential files so that when a program subsequently reads these files, it can detect the end of file easily. However, since files opened for direct access contain special records or counts, it is illegal to insert the endfile statement in files to be accessed directly.

158 No ENDFILE statement allowed on standard input, standard output, or standard error

159 Direct access to standard units is not allowed

160 Direct file required and file attached to unit is not direct

161 It is illegal to overfill a record in a file connected for Direct Access

162 It is illegal to OPEN an existing file with a STATUS = 'NEW'

The named file you specified on the OPEN statement already existed. If you want to create a new file, select a new name for the file.

163 File not OPENed for asynchronous I/O

You must open a file for asynchronous I/O operation. Verify that you have 'ACCESS=ASYNC', and the UNIT number is the one specified in the OPEN statement. Is your path correct? Does the named file exist?

164*

* This error message code number is currently unused.

165 The requested number of bytes to be transferred in asynchronous I/O must be a multiple of 4096-byte

166 The user supplied buffer for an asynchronous read must be aligned on a four-byte boundary

167 Direct access to files without 'seek' capability is not allowed

168 ENDFILE statement not allowed on asynchronous I/O files

169 Encountered EOF while attempting asynchronous read

The file to be asynchronously read cannot contain an end-of-file flag. Remove this flag before attempting any asynchronous I/O.

170 Asynchronous read failed

Did you make the appropriate library call? Is your path correct? Does the named file exist?

171 Asynchronous wait failed

Did you specify ACCESS=ASYNC on the OPEN statement? Refer to Fortran Reference Manual in Chapter 8, Stardent 1500/3000 Fortran Optimization Facilities in the section titled Asynchronous I/O for further information on asynchronous wait.

172 Asynchronous Block Number must be positive integer

173 Q format is not allowed on standard input

174 Internal library error; error cleanup stack has grown too big

175 OPEN statement on an already OPENed and connected file is trying to change a file specifier

176 Double quotes are not allowed in a format as a string delimiter

Fortran does not allow double quotes in a string delimiter. Change them to single quotes.

177 Fopen failed in OPEN while trying to access a device for reading

Did you specify a UNIT number for the file? Does the named file exist? Check your OPEN statement. Refer to the OPEN statement in the Fortran Reference Manual in Chapter 2, Fortran Statements.

178 Failure in OPEN while trying to open a Read_Only file for asynchronous read

Do you have access permission for this file? Is your path correct?

179 Failure in OPEN while trying to open a file for asynchronous access

Do you have access permission for this file? Is your path correct?

180 Failure in OPEN while trying to open a Read_Only file

Do you have access permission for this file? Is your path correct?

181 Failure in OPEN while trying to access a file

Do you have access permission for this file? Is your path correct?

182 It is illegal to set RECL on a file not opened for DIRECT access

RECL should be set only if you specify a direct access in the OPEN statement. Refer to Fortran Reference Manual in Chapter 2, Fortran Statements if you need further information concerning RECL.

183 TRACE: Cannot open the executable for reading to initiate the stack backtrace

Is your path correct? Does the named file exist?

184 TRACE: Error while trying to read executable for stack backtrace

Is your path correct? Does the named file exist?

185 TRACE: The 'magic number' for this executable is wrong for the Stardent 1500/3000

186 TRACE: I am unable to allocate space for the string table while attempting a stack backtrace. Is the current directory full?

187 TRACE: I am unable to allocate space for the symbol table while attempting a stack backtrace. Is the current directory full?

188 TRACE: I am unable to read the string table while attempting a stack backtrace

Is your path correct? Check your path.

189 TRACE: I am unable to read the symbol table while attempting a stack backtrace

Is your path correct? Check your path.

190 When reading under Q format, data item must be integer or logical type

Your data varibles are probably anything other than integer or logical type. Check them, and reassign them if they are not either integer or logical.

191 Buffers larger than 128,000,000 bytes can not be allocated

192 One or more of the environment variables UNFORMATTED_IO, UNFORMATTED_INPUT or UNFORMATTED_OUTPUT is set to something other than BSD or VMS

Check your spelling; you might unintentionally type something other than BSD or VMS. Refer to Fortran Reference Manual in Chapter 10, User Commands for detailed information on these environment variables.

DATA LAYOUT AND CALLING CONVENTIONS

APPENDIX B

This appendix describes two topics: the *data layout* and the *language calling conventions*. In the *data layout* section, all data types supported by the Stardent 1500/3000 are described and illustrated. Each data type including its format parameters and conditions are also explained in details. In the *language calling conventions* section, various passing conventions are described as well as the return mechanism for a variety of function calls (especially useful if you want to interface between different languages such as Assembler, C, and Fortran). Finally, the section also includes a convention for caller and callee save registers, and the register sets are divided accordingly.

B.1 Data Layout

Stardent 1500/3000 Fortran has eleven data types:

- integer, also called INTEGER*4
- short integer, also called INTEGER*2
- BYTE
- real, also called REAL*4
- double precision, also called REAL*8
- complex, also called COMPLEX*8
- double complex, also called COMPLEX*16
- logical, also called LOGICAL*4
- short logical, also called LOGICAL*2
- byte logical, also called LOGICAL*1
- character

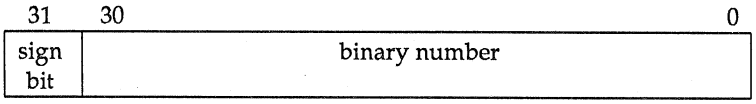
When stored in memory, each has the format described in the following sections.

NOTE
In the floating point formats, when dealing with numbers at or close to the limits of the range, it is possible to exceed the range during ASCII to binary conversion and vice versa. This is caused by rounding errors.

B.1.1 Integer Format

Integer data is always an exact representation of a short integer of value positive, negative, or 0. The integer format, **INTEGER*4**, occupies one 32-bit word and has a range of -2^{31} to $+2^{31}-1$ or –2 147 483 648 to +2 147 483 647.

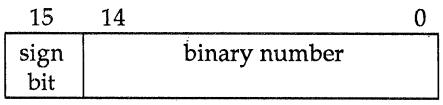
Table B-1. Integer Data Format (INTEGER*4)



B.1.2 Short Integer Format

A short integer format, **INTEGER*2**, occupies half of one 32-bit word and has a range of -2^{15} to $+2^{15}-1$ or -32768 to 32767.

Table B-2. Short Integer Format (INTEGER*2)



B.1.3 Real Format

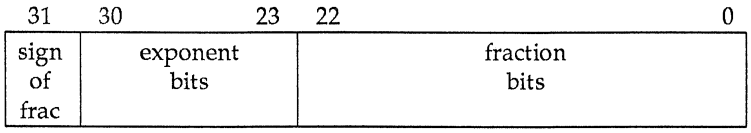
Real data is a processor approximation to a real number having a positive, negative, or zero value. In Stardent 1500/3000 Fortran, real format corresponds to IEEE Single Format. This format is capable of representing numbers in the range of – inf to zero to + inf, as well as **NaN**, which stands for "Not a Number". The real format, **REAL*4**, occupies one 32-bit word in memory and has an approximate range of 1.175495E-38 to 3.402823E+38.

The real format has 1-bit sign, an 8-bit exponent, and a 23-bit fraction. Significance is approximately seven decimal digits.

- The sign bit is 0 for plus, 1 for minus.
- The exponent field contains 127 plus the actual exponent (power of 2) of the number. Exponent fields containing all 0's and all 1's are "reserved." Special interpretations given to the numeric representation are:

- If the exponent is 0 and the fraction 0, the number is interpreted as a signed 0.
- If the exponent is 0 and the fraction not 0, the number is assumed to be "denormalized." Floating point numbers are usually stored in a "normalized" form with a binary point to the left of the fraction field and an implied leading 1 to the left of the binary point.
- If the exponent is all 1s and the fraction is 0, the number is regarded as a signed infinity. If the exponent is all 1s and the fraction is not 0, then the interpretation is "not-a-number" (NaN).

Table B-3. Real Format



B.1.4 Double Precision Format

Double precision data is a processor approximation to a real number having a positive, negative or zero value. The Stardent 1500/3000 Fortran double precision format corresponds to the IEEE Double Format. The double precision format, **REAL*8** or **DOUBLE PRECISION**, occupies two consecutive 32-bit words in memory, and has an approximate range of

$-1.79769313486231 * 10^{308}$ to $-2.22507385850721 * 10^{-308}$
0
$+2.22507385850721 * 10^{-308}$ to $+1.79769313486231 * 10^{308}$

The double precision format has an 11-bit exponent and a 52-bit fraction. Significance is approximately 16 decimal digits. The sign bit is 0 for plus, 1 for minus. The exponent field contains 1023 plus the actual exponent (power of 2) of the number. Exponent fields containing all 0's and all 1's are reserved.

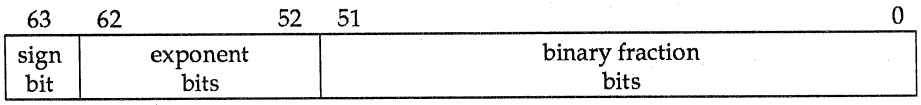
- If the exponent is 0 and the fraction 0, the number is interpreted as a signed 0.
- If the exponent is 0 and the fraction not 0, the number is assumed to be "denormalized." Floating point numbers are usually stored in a "normalized" form with a binary point to the left of the fraction field and an implied leading 1 to the

leff of the binary point.

- If the exponent is all 1s and the fraction is 0, the number is regarded as a signed infinity.
- If the exponent is all 1s and the fraction is not 0, then the interpretation is "not-a-number" (NaN).

In Table B-4 below, the position at which the implied binary decimal point is placed is just to the left of bit position 51.

Table B-4. Double Precision Format

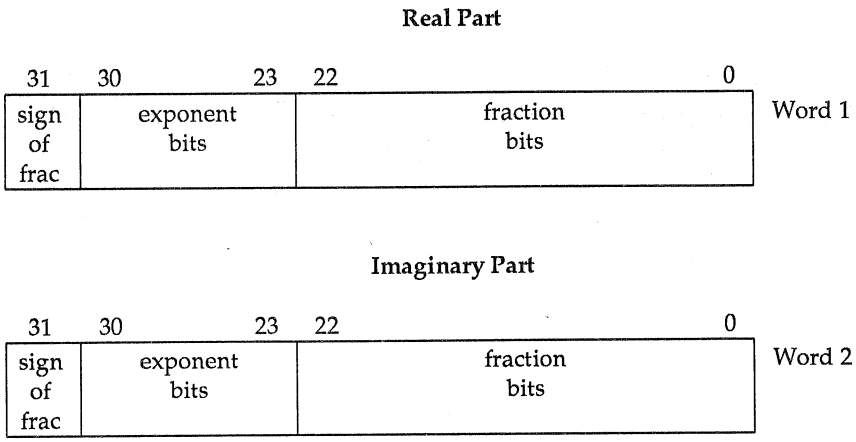


B.1.5 Complex Format

Complex data is a processor approximation to the value of a complex number. The complex format, **COMPLEX*8**, occupies two consecutive 32-bit words in memory. Both the real and imaginary parts have an approximate range of $1.2*10^{-39}$ to $3.4*10^{38}$.

Both the real and the imaginary parts have 23-bit fractions and 8-bit exponents; both have the same significance as a real number. The sign of the exponent is determined in the same manner as that of a real number.

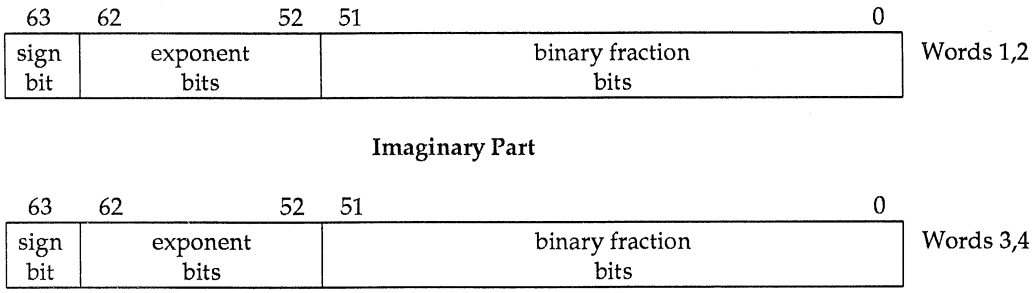
Table B-5. Complex Format



B.1.6 Double Complex Format

Double Complex data is a processor approximation to the value of a complex number. The double complex format, **COMPLEX*16** or **DOUBLE COMPLEX**, occupies four consecutive 32-bit words in memory. Both the real and imaginary parts have an approximate range of $2.2*10^{-308}$ to $1.8*10^{308}$. Both the real and the imaginary parts have 52-bit fractions and 11-bit exponents; both have the same significance as a double precision number. The sign of the exponent is determined in the same manner as that of a double precision number.

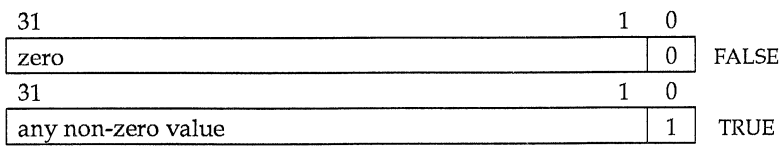
Table B-6. Double Complex Format



B.1.7 Logical Format

Logical data is a representation of *true* or *false*, with 0 representing *false*, and any non-zero value representing *true*. The logical format, **LOGICAL*4**, occupies one 32-bit word in memory.

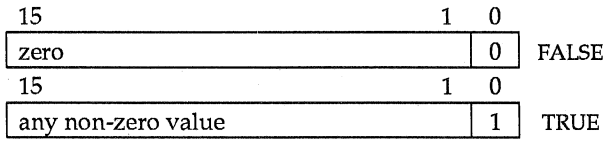
Table B-7. Logical Data Format



B.1.8 Short Logical Format

Logical data is a representation of *true* or *false*, with 0 representing *false*, and any non-zero value representing *true*. The short logical format, **LOGICAL*2**, occupies half of one 32-bit word in memory.

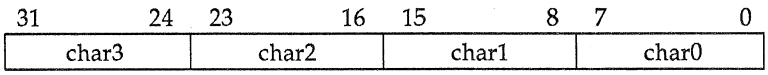
Table B-8. Short Logical Data Format



B.1.9 Character Format

Character data is a character string taken from the ASCII character set. ASCII characters occupy one byte (eight bits) of a 32-bit word, and are packed four to a word in memory. Character strings are stored in memory as a sequence of ASCII codes, 1 per byte.

Table B-9. Character Data Format



One Word

B.2 Language Calling Conventions

This section introduces a convention for passing scalar integer values, scalar floating point values, and vectors as arguments to functions for the Stardent 1500/3000 system. Arguments to functions are either passed in their respective registers or placed on the stack. Aggregates types that fall into special categories and pass certain conditions may be passed in registers. If they are not passed in registers, they are placed on the stack like any other argument.

Stack space for arguments are reserved and allocated by the caller. Therefore, if a callee does not require any stack storage due to absence of calls and absence of any local variable requiring storage, it need not allocate any stack space. Regardless of how an argument is passed, a stack area is reserved as its permanent residence. Therefore, an argument passed in a register also has a stack argument area reserved for it.

In addition, a return mechanism for functions that return values is covered. Scalar return values are returned in registers. Aggregate return values (structures and unions) may be returned in registers if they fall into special categories. Otherwise, the address of the aggregate is passed as an extra argument, and the

function return value is stored directly into the aggregate itself. Finally, a convention for caller and callee save registers and the register sets are categorized and described accordingly.

B.2.1 Stack Allocation

All arguments have a stack area reserved for them. Regardless of how an argument is passed to a routine (via register or stack) the *caller* allocates a stack frame for it. The amount of stack frame allocated for the arguments depends on the number and type of arguments. A minimum of 48 bytes of argument stack is allocated regardless of the number of arguments. Thus, a call with no arguments still causes the caller to allocate 48 bytes of argument area. These 48 bytes are allocated and reserved as locations for the arguments passed in registers. Only scalar arguments in registers have their locations presented in these 48 bytes, and contrastly, vector arguments do not have any memory location reserved for them by the caller. If there are any arguments not passed in registers, they are allocated a stack location beyond the minimum 48 bytes. Eventually, they are positioned on the stack argument area according to their position in the argument list.

B.2.2 Scalar Arguments

Scalar arguments are passed either in the corresponding registers or on the stack. The following describes the scalar conventions, its stack allocation, and its argument placement.

B.2.2.1 Scalar Floating Arguments

The first two floating arguments to a function are passed in the two MIPS floating registers, $f12 and $f14. Both single precision values and double precision values are passed in the same registers. Double precision values reside in the even and odd register pair. For instance, the first double precision argument is passed in registers $f12 and $f13; $f12 contains the low-order four bytes and $f13 holds the high-order four bytes of the double precision values. These register values are not preserved across a call; therefore, they are caller save registers.

The two floating arguments have their stack locations within the minimum 48 bytes of stack location allocated by the caller. The first 16 bytes of this minimum stack is reserved for these two floating arguments. The first 8 bytes is for the argument passed in $f12, and the second 8 bytes are for the argument passed in $f14.

Those floating arguments that are not passed in registers reside on the stack argument area. They reside in locations after the minumum 48 bytes of allocated argument area. Their position on the stack depends on their position in the argument list. Each argument is aligned either on a word boundary or on a 8-byte boundary, depending on its type.

B.2.2.2 Scalar Integer Arguments

Scalar integer arguments are passed in either the MIPS integer registers or via the stack. The first four integer arguments to a function are passed in the four integer argument registers, $a0, $a1, $a2, and $a3. Stack locations for these four arguments in registers are within the minimum 48 bytes of allocated stack argument area. The first register argument has its reserved area at 32 bytes off of the 48-byte minimum stack. The second follows it at 36, the third at 40, and the fourth at 44. Those integer arguments that are not passed in registers are placed in the stack argument area, in locations that are after the 48 bytes of minimum storage. Their position on the stack depends on their position in the argument list. Each of these integer related arguments is placed on a word aligned boundary.

EXAMPLE

The example given below is intended to illustrate the caller's code and the stack and register contents at the point of the function invocation. Consider the following routine.

```
INTEGER*4 I, J, K, L, M;
REAL*4 A, B, C, D, E;
.
.
.
SUBROUTINE FOOBAR(I,A,B,J,K,C,D,E,L,M)
```

NOTE
0 and 8 off the stack are reserved for the floating arguments, A and B, and 32, 36, 40, and 44 are reserved for the integer arguments I, J, K, and L.

Then, it

(1) causes the caller to allocate 72 bytes on its stack.

(2) passes I, J, K, and L in integer registers $a0, $a1, $a2, and $a3.

(3) passes A and B in MIPS floating registers $f12 and $f14.

(4) stores C at a location 48 from the stack pointer.

(5) stores D at a location 52 from the stack pointer.

(6) stores E at a location 56 from the stack pointer.

(7) stores L at a location 60 from the stack pointer.

(8) stores M at a location 64 from the stack pointer.

B.2.3 Structures and Unions

Structures or unions or aggregate types are either passed in registers or on the stack. Structures or unions or aggregate types (henceforth known as aggregates) may be passed in registers if they fall into particular classifications and follow certain conditions. Otherwise, they are placed on the stack like any other ordinary scalar variable. The following section describes the classifications for aggregates.

B.2.3.1 Classification of Aggregates

Aggregates are classified into three main classes:

- Integer Aggregates — all members are integer related types.
- Floating Aggregates — all members are floating related types.
- Mixed Aggregates — all members are both integer and floating related types.

B.2.3.1.1 Integer Aggregates. Aggregates, all of whose members are integer related types, are passed in the four or fewer integer argument registers if the following conditions are satisfied.

- Size of aggregate must not be greater than 16 bytes.
- Alignment of aggregate must not be greater than 4 bytes.
- None of the aggregate members are in another aggregate.
- If the structure is not the first argument, the aggregate must fit completely in the remaining 3 or fewer argument registers even if it is less than 16 bytes in total size.

An aggregate that falls into this category can be passed in the argument registers. The following expression is used to

determine the number of registers required to pass the entire aggregate.

numRegisters = sizeof(structure)/alignment(structure)

If the aggregate does not fall into this category, then the aggregate is placed on the stack obeying the scalar integer argument passing scheme. The remaining argument registers, if any, are now free for later arguments.

Note: *Because structures and unions are not necessarily passed in registers, there are cases where confusion can occur. For instance, when a union whose size is not greater than an integer is being passed to a routine, the receiving routine must have believed that its argument is an integer (which in fact, it is not!). If the union does not obey all the rules of structure passing in registers, as described above, the union is passed on the stack, whereas the routine that receives it assumes that it is in a register; obviously, the generated code contains an error. Thus, in general, unions or structures passed as arguments must also be received as unions or structures. In addition, they should also be of the same type.*

B.2.3.1.2 Floating Aggregates. Aggregates in which all members are floating related types may be passed in the floating argument registers if the following conditions are satisfied.

- Size of aggregate must not be greater than 16 bytes.
- Alignment of aggregate must not be greater than 8 bytes.
- Aggregate must be homogeneous in nature, either all floats or all doubles.
- An aggregate member may not be another aggregate.
- The aggregate must fit completely in the remaining argument registers even if it is less than 16 bytes in total size.

Aggregates that fall into this category are passed in the floating registers. The number of floating registers required is determined by the following expression.

numRegisters = sizeof(structure)/alignment(structure)

If the aggregate does not fall into this category, then it is passed on the stack, obeying the scalar floating point argument scheme.

B.2.3.1.3 Mixed Aggregates. Such aggregates are always passed on the stack.

B.2.4 Fortran Character Variables

Fortran character variables are passed in a descriptor. A descriptor is constructed for the Fortran character variable (henceforth, known as string), and the address of the descriptor is then passed as an ordinary integer quantity. A descriptor is a structure with three four-byte members where the first member of the structure is the address of the string, the second is the length of the string, and the third contains a descriptive string "sTrG".

B.2.5 Caller's Stack Frame

The following is an illustration of a caller's stack frame just before the call to the function is made.

```
        .
        .
        local variables...
high:   outgoing argument n
        outgoing argument n-1
        .
        .
        .
        outgoing integer argument 4
        outgoing integer argument 3
        outgoing integer argument 2
        outgoing integer argument 1
        empty 8 bytes, reserved.
        empty 8 bytes, reserved.
        outgoing floating argument 2
sp -->  outgoing floating argument 1
```

B.2.6 Vector Arguments

There are four vector registers reserved for vector arguments. They are bank 0 block 2, bank1 block 2, bank 2 block 2, and bank3 block2. The first vector argument is placed in bank 0 block 2, and the rest is in sequence to the rest of the vector registers.

In addition to these vector registers, integer registers $a0 and $a1 contain information which may be helpful to the vector routine. Integer register $a0 is the vector length associated with vector argument and return registers. Integer register $a1 contains a bit pattern indicating which vector registers need to be saved and restored by the callee. A value of 1 in the bit-field indicates that the vector register needs to be saved and restored. The layout for the registers are as follows:

WARNING
A function invocation with more than four vector arguments is an error. There is no support for passing more than four vector arguments to a function. Unpredictable results may appear when more than four vector arguments are passed.

```
$v0.0 - bit 0 (least significant bit)
$v0.1 - bit 1
.
.
.
$v0.7 - bit 7
$v1.0 - bit 8
.
.
.
$v3.7 - bit 31 (most significant bit)
```

These vector registers are preserved across a call. Thus, they are callee save. The integer registers $a0 and $a1 obey the normal scalar calling convention, and therefore, are caller save.

B.2.7 Returning Mechanism

Ardent's return mechanism is through registers as much as possible. Aggregates are the only case that a register return is not possible. In such case, a special mechanism is establised to ensure the maximum efficiency. In the following section scalar, aggregate, and vector return mechanisms are described.

B.2.7.1 Scalar Returns

Function returns are either returned in their corresponding registers or via a direct memory store. There are four integer scalar registers reserved for returning up to four integer values. Functions that return structures, that fall into the integer structure classification described above, are also returned in these four integer registers. The four integer registers reserved are $v0, $v1, $a0, and $a1.

Floating values are returned in the MIPS floating registers $f0 and $f2. Complex returned values are returned with $f0 containing the real part and $f2 containing the imaginary part of the complex. Double precision returned values are returned in the corresponding register with the low-order 4 bytes in $f0, and the high-order 4 bytes in $f1.

Fortran character functions return a descriptor by value. The address of the character variable is returned in the integer register $v0, and the length is returned in $v1.

B.2.7.2 Aggregate Returns

Integer aggregate functions return an aggregate in integer registers $v0, $v1, $a0, and $a1 if they fall into the same classification as the aggregate arguments. Floating aggregate functions return in floating registers $f0 and $f2 if they also follow the same restrictions as the aggregate arguments.

Those aggregates that are not returned in registers are returned directly in the aggregate's memory location. The address of the receiving aggregate is passed as an extra argument to the function being invoked. This address is passed in the integer register $v1. The function in turn directly stores into the address of the aggregate that return values.

B.2.7.3 Vector Returns

Vector functions return values in vector registers bank 2 block 1, and bank 3 block 1. Complex vector functions return the real part in the first vector return register, and the imaginary part in the second return register. Both return registers are chosen such that the amount of bank conflicts may be kept to minimum in most math libraries.

B.2.8 Register Conventions

Ardent's sets of integer scalars, floating scalars, and vector register sets are categorized into two classes of registers: the *caller save* and the *callee save.*

Caller save registers are those that are not preserved across a call. They need to be stored and restored around a call if they are used again. On the other hand, *callee save* registers are those that are preserved across a call. They are stored and restored by the callee if the callee wants to use them.

B.2.8.1 Integer Registers

The 32 integer registers are divided up in the following:

- $0 — always contains a zero.
- $at — an assembler temporary.

- $v0 <-> $t7 — caller save registers (includes argument and return registers).
- $s0 <-> $s7 — callee save registers.
- $t8, $t9 — caller save registers.
- The rest of the integer registers are reserved for special uses.

B.2.8.2 Floating Registers

The 16 MIPS floating registers are divided in the following:

- $f0 <-> $f18 — caller save registers.
- $f20 <-> $f30 — callee save registers.

B.2.8.2.1 VCU Scalar Floating Point Registers. Table B-10 gives the names and functions of the VCU scalar floating point registers.

Table B-10. VCU Scalar Floating Point Registers

Register	Function
F0–F11	Scratch registers (caller saves)
F12–F15	Register variables (callee saves)
F16–F19	Scratch registers (caller saves)
F20–F23	Register variables (callee saves)
F24–F27	Scratch registers (caller saves)
F28–F31	Register variables (callee saves)

The "F" in the register names in Table B-10 means the register is holding a single precision floating point quantity. It may be replaced with one of the following built-in names: **D** or **d**, if the register is holding a double-precision floating point quantity; or **I** or **i**, if the register is holding an integer quantity.

In addition, there is one accumulator register indicated by the built-in name **A** or **a**.

B.2.8.3 Vector Registers

The 28 vector registers are divided in the following:

- $v0.1, $v1.1, $v2.1, and $v3.1 are caller save registers.
- The rest of the vector registers are callee save.

B.2.8.4 Special Registers

In addition, there are some other special registers. Those are accumulator registers, The FPU status register, and the FPU Mask register. Each has the following convention:

- The accumulator registers are all caller save.
- The FPU status register is caller save.
- The FPU Mask register are divided into two parts, the first half of the 8 vector cells which reserved as the mask registers are callee save, and the rest are caller save. The 8 FPU Mask registers reside on bank 1 block 0 in cells 24 to 31.
- The vector length is callee save. If a routine changes the system's vector length, it should restore the vector length just before it returns.

B.2.9 Summary of Call/Return Conventions

The call/return process consists of four phases:

- The calling program:
  - Saves those of integer registers $v1–$t7, and the *caller save* floating point registers $f0–$f18 that are needed after the call.
  - Puts the arguments in registers $a0–$a3.
  - Puts arguments that won't fit in registers on the stack.
  - Branches and links to the beginning of the subroutine.
- The called program:
  - Decrements the stack pointer to allocate the new stack frame.

NOTE
Refer to the *Programmer's Guide* in Chapter 10, *Language Interfacing* for further detailed examples on passing conventions.

- If the called program in turn calls other programs, the return address is saved on the stack.
- Saves those of integer registers $s0–$s7, and the *callee save* floating point registers $f20–$f30 that are used.
- Begins execution.

- To return to the caller, the called program:
  - Puts the return value into register $v0 if it is an integer or a pointer, and otherwise into register $f0 if it is a float or a double.
  - Restores the return address to the link register.
  - Restores any of the integer registers $s0–$s7, and the *callee save* floating point registers $f20–Sf30 that were saved.
  - Increments the stack pointer to the previous value.
  - Returns to the return address.

- After the return, the calling program:
  - Restores any of the integer registers $v1–$t7, and the *caller save* floating point registers $f0–$f18 that were saved before the call was made.

If a function makes no calls or does not change integer registers $s0–$s7 and the *callee save* floating point registers $f20–$f30, there is no need for the function to touch the stack (except, perhaps, to store its arguments there). In this case, the code is simplified because there is no need to get a new stack frame nor to do any stores or reloads of the stack. The link address remains in register $ra throughout the called function's execution.

INDEX

A

B

C

D

E

F

G

H

I