March, 1989
Order Number: 311020-004

iPSC®/2

FORTRAN LANGUAGE REFERENCE MANUAL

intel Corporation

Disclaimer

GREEN HILLS SOFTWARE, INC MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Further, Green Hills Software, Inc. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Green Hills Software, Inc. to notify any person of such revisions or changes.

REV.	REVISION HISTORY	DATE
-001	Original issue	12/87
-002	Revision	03/88
-003	Revision	11/88
-004	Revision	03/89

RESTRICTED RIGHTS

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the rights in Technical Data and Computer Software clause at 52.227-7013. Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

Table of Contents

CHAPTER 1

Overview

1.1. How to Use This Manual

Each Green Hills compiler is specified by three components: the Language, the Target, and the Host. The Language, in this case Fortran, is the computer language that the compiler translates. The Target, in this case the 80386, is the machine on which your program will run. The Host is the computer system on which the compiler runs. The organization of this manual is given below.

Overview

The Overview describes the structure of the documentation for Fortran-386.

The Fortran Language

The Fortran Language section specifies the Fortran language features and extensions that are supported. It also explains restrictions and indicates how closely Fortran-386 matches other Fortran compilers.

80386 Target

The 80386 Target chapter describes the target processor and operating system environment in which your program will operate. It describes calling conventions, register allocation and memory allocation strategies. It describes restrictions imposed on the compiler by the target system. It also tells how to modify the output of the compiler to be compatible with different target environments.

Optimization

The Optimization chapter gives detailed information about the optimizations used by Fortran-386 to improve program performance. It also gives you general ideas as to how to get the best performance out of your program.

Porting Programs to Fortran-386

This chapter tells you about difficulties that you may encounter in moving a program developed with another compiler to Fortran-386. It gives specific examples of difficulties that may be encountered and how to resolve them.

Compile Time Options

This chapter describes how to adjust the output of Fortran-386 to accommodate your needs by using the many variations that have been implemented.

1.2. Related Documentation

You will need documents in addition to this manual in order to use Fortran-386. These documents will tell you how to install Fortran-386 and how to compile, link, execute, and debug Fortran-386 programs. You may also need documentation describing your 80386

assembler and linker, and the 80386 architecture.

CHAPTER 2

FORTRAN Language Introduction

FORTRAN is the oldest and most proven high-level computer programming language in existence today. The origins of FORTRAN date back to the mid 1950's. Up to that time, most programs were written in complicated machine languages that required considerable coding and debugging time. FORTRAN, the first high level language, was designed to reduce the communications gap between people and computers using a language structure that people could read and understand easily. A FORTRAN compiler is a program that translates the written FORTRAN programs into the specific machine language that a computer can read.

FORTRAN was developed primarily to solve problems that involve the manipulation of numerical data. The language syntax was designed to accommodate standard algebraic notation. Using FORTRAN, programmers can employ the computer to translate existing mathematical formulas for processing. The name FORTRAN was derived from this principle of FORmula TRANslation.

Through the late 1950's and early 1960's, FORTRAN evolved as the primary scientific programming language. In 1966, the American National Standards Institute (ANSI) approved the first FORTRAN language standard providing a new level of FORTRAN program portability. Programs written in ANSI standard FORTRAN-66 could be run on a variety of different computers with little or no program modification.

Through the late 1960's and early 1970's, computers became more sophisticated and the demands on software became more complicated. FORTRAN applications expanded to include more general program tasks such as sophisticated input/output, text manipulation, and structured programming facilities. In 1978, the American National Standards Institute gave final approval to an extended FORTRAN standard that came to be known as FORTRAN-77. Today, FORTRAN is still the most widely used programming language among mathematicians, scientists, and engineers.

2.1. FORTRAN Standards

Fortran-386 implements the ANSI FORTRAN-77 (Full Language) Standard, ANSI X3.9-1978 and the Military Standard Fortran, as described in the document "FORTRAN, DOD Supplement to American National Standard X3.9-1978, MIL-STD-1753". It is also compatible with the Berkeley 4.3BSD f77 compiler and the VAX/VMS Fortran V4.6 compiler.

2.2. Validation

Fortran-386 is tested by running the FORTRAN Compiler Validation System Version 2.0 (1978) from the U.S. Office of Software Development and the U.S. Department of Commerce, National Technical Information Service.

2.3. Form of Presentation

This manual defines the form and interpretation of language elements specific to Fortran-386. It does not attempt to teach general programming concepts. Familiarity with a beginning programming language, such as BASIC, is ample background for taking on Fortran-386 using this manual.

Each of the following chapters covers one general category of language concepts, such as data types, expressions, program structure, specification statements, control statements, or intrinsic functions. Concepts, such as variables and procedures, are defined specifically in terms of the Fortran-386 language. Explanations are elementary in nature, but concise to accommodate more experienced programmers. This manual may be used to learn the Fortran-386 programming language, or it may be used as a Fortran-386 reference manual.

CHAPTER 3

Fortran Syntax

A computer programming language, like a natural human language, is based on rules of syntax. Syntax is the predefined order or arrangement of language elements that produces meaning.

This manual uses the following typographical conventions within examples to highlight various entities that constitute the language's syntactic structure.

- Words in UPPERCASE LETTERS indicate language keywords.
- Words in lowercase letters indicate syntactic items, such as "symbolic-name" or "number", and literal names within examples, such as the variable names "var1" and "var2".
- Items enclosed in square brackets [ ] are optional.
- A horizontal ellipsis ... indicates that you can repeat the preceding optional item any number of times. A vertical ellipsis indicates an ambiguous continuation of the preceding items.

3.1. The Character Set

The fundamental element of any programming language is the character. Fortran-386 uses the standard ASCII character set, which consists of uppercase and lowercase letters (A through Z), digits (0 to 9), and a group of special characters that have a specific interpretation in Fortran-386. You can collectively refer to letters and digits as alphanumeric characters.

There are 16 special characters in Fortran-386 as shown in the table below.

FORTRAN-386 Special Characters			
Character	Name	Character	Name
'	apostrophe	=	equals sign
*	asterisk	(	left parenthesis
	blank	–	minus sign
:	colon	%	percent sign
,	comma	+	plus sign
.	decimal point	)	right parenthesis
&	ampersand	_	underscore
$	dollar sign	/	slash

Fortran-386 also recognizes the ASCII control characters that signify carriage returns, tabs, new line feeds, and form feeds.

All characters conform to a collating order. The collating order defines a hierarchy among the characters for the purpose of sorting character strings. Fortran-386 can compare two character strings a character at a time according to the ASCII collating sequence. Under ASCII conventions, all characters correspond to numeric values that determine the hierarchy. Please refer to the "ASCII and Hexadecimal Conversions" chapter for a listing of the collating sequence.

3.2. Symbolic Names and Keywords

Fortran-386 identifiers include symbolic names and keywords. A symbolic name is a sequence of alphanumeric characters. The dollar sign and underscore characters are valid in a symbolic name. However, the first character in a symbolic name must be a letter (A through Z or a through z). Symbolic names identify program entities, such as constants, variables, arrays, and program units.

A keyword is a sequence of letters that identifies a particular statement or serves as a separator within a statement. For example, COMMON, IMPLICIT, DATA, IF, and THEN are Fortran-386 keywords that have a specific purpose in the appropriate statement. Note that certain keywords satisfy the description of a symbolic name. Keywords have a specific purpose only if used within the specific context defined in Fortran-386.

3.2.1. Scope of Symbolic Names

Symbolic names vary in scope, depending upon what a given symbolic name identifies. Different program entities can be divided into two classes pertaining to scope: global and local.

A global entity has a scope that covers the entire executable program. A symbolic name that identifies a global entity cannot identify a second global entity in the same executable program.

A local entity has a scope that covers a single program unit. A symbolic name that identifies a local entity cannot identify a second local entity in the same program unit. A symbolic name that identifies a global entity in a program unit cannot identify a local entity within the same program unit, except for common block and external function names.

The following program entities are classified as global to a Fortran-386 executable program:

- block data subprogram
- common block
- external function
- main program
- subroutine

The following program entities are classified as local to a single Fortran-386 program unit:

- array
- constant
- dummy procedure
- statement function
- variable

There are two local program entities that have a scope somewhat smaller than a program unit: a dummy argument for a statement function statement and an implied DO variable in a DATA statement.

A symbolic name that identifies a dummy argument for a statement function statement has a scope of that statement. See the "Statement Functions" section for more information on statement function statements.

A symbolic name that identifies the DO variable for an implied DO in a DATA statement has a scope of the implied DO list. See "The DATA Statement" section for more information on DATA statements.

3.3. Statements and Lines

All Fortran-386 syntactical items, such as keywords symbolic names, statement labels, constants, operators, and special characters are used to form Fortran-386 statements. The statements you write using the Fortran-386 language are translated into computer instructions. Statements are classified as executable or nonexecutable.

An executable statement is any statement that specifies some processing action, such as the PRINT or CONTINUE statement. Executable statements execute sequentially in the order that they are placed in a program unit. Execution of an executable program begins with the first statement in the main program. Control statements, such as GOTO and CALL, transfer the execution sequence to a different point in the program. Statements that transfer the execution sequence are considered executable statements.

Nonexecutable statements define and classify program units, specify entry points in subprograms, specify editing information, and specify initial values and execution characteristics for data.

Each Fortran-386 statement is written on one or more program lines. A program line is a sequence of character positions. You can refer to character positions as columns. The columns are numbered consecutively beginning with 1, and proceeding from left to right. There are three kinds of program lines in Fortran-386: initial lines, continuation lines, and comment lines. Each type serves a different purpose within a program.

There are two ways to format a program line in Fortran-386: the free-field format and the column-based format. This section describes both formats in detail.

3.4. Free-field Line Format

The free-field format is indicated by the use of a tab character before column 72 of the initial line of a statement. If the initial line of a statement is in free-field format then all continuation lines will also be in free-field format. To indicate that a continuation line and all subsequent continuations lines are in the free-field format, use an ampersand, &, in column 1 to indicate that the line is a continuation of the previous line.

The free-field format enables you to enter program lines at a standard CRT type terminal without regard to specific column-oriented line fields. In free-field format, the length of a program line is unlimited. However, if your terminal has a width of 80 character positions, you might want to limit your lines to 80 characters to prevent the lines from running off the screen and out of view. On an 80 character terminal, you can use the entire 80 columns as an initial line for statement text with or without a statement label. The initial line is the first line used to hold a statement. If the statement requires more than the 80 columns in the initial line, you can place the remainder of the statement in continuation lines.

Example:

10<tab>FORMAT('This long statement contains a tab so it is in free-field format')

3.5. Column-based Program Format

The column-based format originated in the early days of FORTRAN when programs were stored on punch cards. Program lines consist of precisely defined fields. Each field can only contain a specific kind of information. Support of the column-based format enables the Fortran-386 compiler to process FORTRAN programs written before the free-field format was available. Many experienced FORTRAN programmers still prefer the structure and organization column-based formatting lends to a program. Fortran-386 interprets a line in column based format unless it contains a tab character in columns 1 through

72, or an ampersand character, &, in column 1 of a continuation line.

3.5.1. Program Line Fields

In the column-based format, the 80 columns in a program line are divided into four fields. Each field is reserved for a specific kind of information. Columns 1 through 5 are reserved for the statement label. You can use statement labels to identify and reference specific statements in a program. Column 6 is reserved for a continuation mark. The continuation mark indicates that the line is a continuation of the preceding line. Columns 7 through 72 are reserved for the actual statement text. Columns 73 through 80 can contain identifying information or notations. The compiler ignores all characters in this field. In the days of punch card storage, columns 73 through 80 contained a sequence number used to identify the position of an individual punch card within a stack of cards. The following example shows a line in column-based format. Columns 1 through 5 of the first line contain a line number, 10000. Column 6, the continuation column, contains a blank, indicating the initial line of a statement. Columns 7 through 72 contain the initial line of a FORMAT statement. Columns 73 through 80 contain a card sequence number which is ignored by Fortran-386.

Example:

```
10000 FORMAT('This line contains no tab, it is in column-based format') 01234567
```

3.5.2. Initial and Continuation Lines

A statement can span several program lines. The initial line is the first line used to hold a statement. If a statement exceeds column 72 of the initial line, you can place the remainder of the statement in continuation lines. You can use a maximum of nineteen continuation lines to hold a statement. Therefore, a statement can span a maximum of twenty program lines, or 1320 character positions.

You must distinguish continuation lines from initial lines in a program. Any character in column 6 except a blank or the digit 0 serves as a continuation mark, indicating that the line is a continuation line.

3.6. Comment Lines

Comment lines provide a method of program notation or documentation. Even high level language programs can appear somewhat confusing and ambiguous when you try to read them. The comment line enables you to place notes and explanations within a program to help other programmer's understand your programming intentions and to help you remember at a later date what the program does.

You must distinguish comment lines from the other kinds of lines in a program. If the letter C or an asterisk appears in column 1, Columns 2 through 80 may contain any characters in the Fortran-386 character set including all blanks. Comment lines can appear anywhere within a program unit and do not affect the executable program in any manner.

3.7. Statement Labels

You can use statement labels to identify and reference individual statements in a program. You can label any statement, but only labeled executable statements can be referenced with the statement label. For example, the GOTO statement transfers execution to an executable statement identified with a statement label. The GOTO statement requires the statement label as a parameter for reference.

A statement label is a sequence of one to five digits. At least one of the digits must be nonzero. You can place statement labels anywhere in columns 1 through 5 of the

statement's initial line. Continuation lines cannot contain the statement label.

Statement labels have the scope of a program unit. Every statement label in a program unit must be unique. Blank characters and leading zeros are insignificant for distinguishing different statement labels. For example, the following statement labels are the same in a Fortran-386 program.

Column	1	2	3	4	5
	7	7	7		
	0	0	7	7	7
	0	7	7	7	
			7	7	7

3.8. Statement Order

Different configurations and groupings of statements form program units. However, there are certain rules that apply to the order of statements and comments within a program unit.

- Comment lines can appear anywhere before the END statement.
- The PROGRAM statement can appear only as the first statement of a main program. The FUNCTION, SUBROUTINE, and BLOCK DATA statements can appear only as the first statement in a subprogram.
- FORMAT and ENTRY statements can appear anywhere before the END statement.
- PARAMETER statements can appear anywhere before DATA statements, statement function statements, and executable statements.
- IMPLICIT statements must appear before all other specification statements except PARAMETER statements and FORMAT statements.
- All other specification statements (COMMON, DIMENSION, EQUIVALENCE, EXTERNAL, INTRINSIC, SAVE) must appear before any DATA statements.
- DATA statements can appear anywhere following the specification statements.
- All statement function statements must appear before any executable statements.
- All executable statements must appear before the END statement.

- The END statement must be the last statement in a program unit.

The table below summarizes the rules for ordering statements in a Fortran-386 program unit. The horizontal lines separate the kinds of statement that you cannot mix in a program unit. The vertical lines delineate the kinds of statement that you can mix.

<table>
<tr><td rowspan="5">Comment Lines</td><td colspan="3">PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA Statement</td></tr>
<tr><td rowspan="4">FORMAT and ENTRY Statements</td><td rowspan="2">Parameter Statements</td><td>Implicit Statements</td></tr>
<tr><td>Other Specification Statements</td></tr>
<tr><td rowspan="2">DATA Statements</td><td>Statement Function Statements</td></tr>
<tr><td>Executable Statements</td></tr>
<tr><td colspan="4">END Statement</td></tr>
</table>

CHAPTER 4

Data Types

All computer programs, regardless of their degree of complexity, create, manipulate, or modify pieces of information that pertain to a given problem or task. The different values related to a problem or computing task are collectively referred to as data.

Data can consist of numeric values such as the circumference of the earth measured in kilometers or a worker's gross salary. A numeric value is represented by a series of digits. Data can also consist of textual information such as a client's name and address or abbreviations for chemical compounds in an equation. Text data is represented by strings of characters. You can use any character in the Fortran-386 character set as textual data.

All data in a Fortran-386 program falls into one of the following categories.

- integer
- real
- double precision
- complex
- logical
- character

Fortran-386 supports the use of Hollerith data. Hollerith data provided a text processing capability for earlier versions of the FORTRAN language and is considered an extension to the FORTRAN-77 standard. It is provided primarily for compatibility with older FORTRAN standards.

4.1. The Implicit Data Type Convention

Programmers frequently use a symbolic name in a program to identify a particular datum or data structure. Symbolic names used in this manner serve a dual purpose. First, the name identifies the datum or data structure within a program unit. Second, the name identifies a specific data type.

You can specify the data type for a symbolic name explicitly using the type statements. Refer to the "Specification Statements" chapter for more information. In the absence of an explicit data type specification, the first letter of the symbolic name determines the data type implicitly. Symbolic names default to either the integer or real data type.

If a symbolic name has the letter I, J, K, L, M, or N as a first letter, the symbolic name has an implicit integer data type. Any other letter of the alphabet implies a real data type. For example, the following symbolic names default to the integer data type in the absence of an explicit type specification: list, increment, multiple, kilo.

The following symbolic names default to the real data type in the absence of an explicit type specification: alpha, delta, total, variation.

You can use the IMPLICIT statement to change the implicit data type convention. Refer to the "IMPLICIT Statement" section for more information. An explicit data type specification using a type statement supercedes the implicit data type convention.

4.2. Integer Type

An integer is any whole positive number, any whole negative number, or zero (0). Zero is neither positive nor negative. Integers have no fractional part. Therefore, integers cannot contain decimal points.

You can write integers with a leading sign. If you omit the sign, the integer is considered to be positive. The integers 721 and + 721 represent the same value.

A constant is a value in your program that does not change. Refer to the "Constants" section for more information. The following are some valid integer constants: 1, 642, + 25, 9287, −41, −73268, 0.

A standard integer occupies four bytes of memory space and can represent values that range from −2147483648 up to + 2147483647. However, you can specify integers that occupy one or two bytes of memory. Refer to the "INTEGER Statement" section for additional information. The "80386 Target" chapter describes how Fortran-386 represents integers internally.

4.3. Real Type

In Fortran-386, a real number is any number that can express a fractional component, an exponent, or both. Real numbers can be either positive or negative. There are three forms for real numbers expressed as constants:

- basic real constant
- basic real constant with exponential notation
- integer constant with exponential notation

The basic real constant consists of an optional sign, an integer component, a decimal point, and a fractional component. Both the integer and fractional components are series of digits. The following are some examples of valid basic real constants: 1.5, + 82.7, .007, 375., −794.0, −.299999.

Exponential or scientific notation consists of the letter E followed by an optionally signed integer. The value of a basic real constant or an integer constant with exponential notation is the product of the constant that precedes the E (the mantissa) and the power of 10 indicated by the integer that follows the E (the exponent). The following examples show some basic real constants and integer constants with exponential notation.

5.82E2	=	582.0
314159.00E−5	=	3.14159
−.229E−3	=	−.000229
11E5	=	1100000
−5E−2	=	−.05
−100E+ 8	=	−10000000000

Standard real numbers occupy four bytes of memory space. Refer to the "80386 Target" chapter for information regarding the range, precision, and representation of the REAL data type. You can also specify real numbers that occupy eight bytes of memory. Refer to the "REAL Statement" section for more information.

4.3.1. Double Precision Type

Double precision real numbers provide additional significant digits of accuracy for real numbers. Use a double precision real number when the range and accuracy of a basic real number is inadequate for the application. There are two forms for a double precision real number.

- basic real constant with exponential notation
- integer constant with exponential notation

A double precision real number uses a slightly different exponential notation. Double precision exponential notation consists of the letter D followed by an optionally signed integer. The value of a basic real constant or an integer constant with exponential notation is the product of the constant that precedes the D (the mantissa) and the power of 10 indicated by the integer that follows the D (the exponent). The following examples show some basic real constants and integer constants with double precision exponential notation.

252.7D2	=	25270.0
.007D−1	=	.0007
883366.0D−4	=	88.3366
837612458378183D−8	=	8376124.58378183
−.0045D+3	=	−4.5
−69124820D−3	=	−69124.820

Double precision real numbers occupy eight bytes of memory space (two numeric storage units). Refer to the "80386 Target" chapter for information regarding the range, precision, and representation of the DOUBLE PRECISION data type. You can use the DOUBLE PRECISION type statement or a REAL*8 type statement to declare double precision numbers. Refer to the "DOUBLE PRECISION Statement" and the "REAL Statement" section for more information.

4.4. Complex Type

To find the roots of certain mathematical equations, we must sometimes consider the roots of negative numbers. Since the set of real numbers does not allow for such a possibility, mathematicians have defined an imaginary unit that uses the symbol i and is equal to the square root of negative one. Using the imaginary unit, we can write the square root of a negative number as the product of a real number and the number i. Numbers expressed in this manner are called pure imaginary numbers.

A complex number is any number that can be expressed in the form A+Bi. A and B are real numbers and i is equal to the square root of negative one. In Fortran-386, we can write a complex number as an ordered pair of integer or real constants separated with a comma and enclosed in parentheses. The first member of the pair is the real constant and the second member of the pair is the imaginary constant as shown in the following format specification.

(real-constant,imaginary-constant)

A complex number is always stored as a pair of real values. Either constant can be positive, negative, or zero. The following examples show some valid complex numbers expressed as constants.

(0,2)	=	0+2i or 2i
(7.5,2.2)	=	7.5+2.2i
(−11,−3)	=	−11−3i
(−.6E3,.55)	=	−600+.55i
(945E−2,−41E−1)	=	9.45−4.1i

A complex number occupies eight consecutive bytes of memory space (two consecutive numeric storage units) in a storage sequence. You can also specify complex values that occupy sixteen bytes of memory. Refer to the "COMPLEX statement" section for more information.

4.5. Logical Type

A logical datum can assume one of two values: true or false. The principle of binary logic is fundamental to the digital computer, having its origins in binary or Boolean arithmetic. In Fortran-386, logical data serves as a simple decision making criterion enabling a program to evaluate logical propositions. A proposition is a statement that evaluates to either true or false.

There are only two logical constants in Fortran-386: the words TRUE and FALSE, delimited on both sides with periods. You write the logical constants in a program as follows.

.TRUE. or .true.
.FALSE. or .false.

A standard logical datum occupies four bytes of memory space. However, you can specify logicals that occupy one or two bytes of memory. Refer to the "LOGICAL Statement" section for additional information. The "80386 Target" chapter describes how Fortran-386 represents logicals internally.

Use the LOGICAL statement to declare a symbolic name with the logical data type. Refer to the "LOGICAL Statement" section for more information.

4.6. Character Type

The character data type enables a program to process textual information. A character datum is a string of one or more characters delimited on both sides with apostrophes. Character data are often referred to as strings. Fortran-386 recognizes and differentiates between upper- and lowercase letters. The following examples show some valid string constants.

'Please enter your password.'
'The limit as T goes to 0 is ... '
'PERCENTAGE OF ERROR < 7%'

You can use any character in the Fortran-386 character set within a string including the blank character and the apostrophe. However, to represent the apostrophe within a string you must use two consecutive apostrophes, as shown in the following example.

'It''s TWELVE O''Clock!' (Displays as: It's TWELVE O'Clock!)

The length of a character string is the number of characters, including blank characters, that appear between the apostrophes. The apostrophes used as delimiters do not count in the string length. Two consecutive apostrophes within a string count as one character. The length of a character string must be greater than zero.

'Please enter your password.' (String length is 27)
'It''s TWELVE O''Clock!' (String length is 20)

Character data occupies one character storage unit in a storage sequence for each character in the string. A character storage unit in Fortran-386 is one byte of memory space.

Use the CHARACTER statement to declare a symbolic name with a character data type. Refer to the "CHARACTER Statement" section for more information.

4.6.1. Hollerith Type

Hollerith data provided a text data processing capability for earlier versions for the FORTRAN language. The FORTRAN-77 standard is the first version of the language to provide the character data type that is considered superior to the Hollerith form. Hollerith data is considered an extension to the FORTRAN-77 standard provided primarily for compatibility with the earlier FORTRAN standards.

Hollerith data, like character data, is a string of characters. You can use any character in the Fortran-386 character set within a Hollerith string including the blank character. The form for a Hollerith constant consists of a nonzero, unsigned, integer constant (n), the letter H, and a string of contiguous characters (c) as shown in the following format specification.

```
nHccc...c

(n characters after the H)
```

The following examples show some valid Hollerith constants.

```
16HToday's date is:
11HGRAND TOTAL
4HNaCl
```

You cannot declare a symbolic name with a Hollerith data type. You can identify Hollerith data, other than constants, with a symbolic name of type integer, real, or logical using a DATA statement or READ statement.

CHAPTER 5

Constants, Variables, Arrays, and Substrings

Fortran-386 provides different methods for handling the data that your program is designed to process. Different programming situations call for a different form of data representation. In certain situations, it is most appropriate to specify a data item explicitly. In other situations, it is most appropriate to specify a data item using a symbolic name that can take on new values as the program progresses. For programs that process large amounts of data, it can be most efficient to specify groups of related or similar data using one symbolic name.

To develop efficient problem solving algorithms using Fortran-386, you must understand the proper use of constants, variables, arrays, and substrings. This section describes the conceptual nature of these four entities. The descriptions refer you to the appropriate Fortran-386 statements used to implement these entities in a program.

5.1. Constants

A constant is an explicit, literal representation of a numeric, logical, or character value in a program. Constants remain unchanged during program execution. The "Data Types" chapter shows examples of constants for all the different data types.

Consider a simple program that calculates the area of a circle using the standard formula $A = \pi r^2$ where r is the radius of the circle, π is equal to 3.1415926, and A is the area. The value of π remains unchanged regardless of any other value in the calculation. Therefore, to translate the area formula to Fortran-386, you can use the real constant 3.1415926 for π. If you want additional accuracy in your area calculation, you can use π expanded to the extended precision constant 3.141592653589793D0.

You can identify a constant with a symbolic name using the PARAMETER statement. When you require a long constant, such as 3.141592653589793D0, at many different locations within a program, repetitious typing of digits can become space and time consuming. To simplify the situation, you can assign a symbolic name such as "pi" to the constant 3.141592653589793D0 using the PARAMETER statement. The program substitutes the constant value wherever the assigned name "pi" appears in the program. Refer to the "PARAMETER Statement" section for more information.

5.2. Variables

A variable consists of a symbolic name and an associated value. A variable can represent a numeric, logical, or character value in a program. Unlike the symbolic name for a constant, the symbolic name for a variable can assume many different values during the execution of a program.

Consider, again, the simple program that calculates the area of a circle using the standard formula $A = \pi r^2$ where r is the radius of the circle, π is equal to 3.1415926, and A is the area. The value of π remains unchanged and is represented with the real constant 3.1415926. However, the value of r varies for circles of different sizes. If you use a constant to represent r, the program would be limited to circles of one size. You would have to rewrite the program every time you wanted to calculate the area of a circle with a different radius. Instead, let the radius be a variable. The program can assign a new value

to the radius variable for each new area calculation.

A variable must have an assigned value before you can reference the variable name in the program. When a reference to a variable name executes, the program uses the value that is currently assigned to the variable at that point in the execution of the program.

Variables assume values through assignment statements. For example, a simple integer variable named int assumes the value 14 in the following arithmetic assignment statement.

```
int = 12 + 2
```

The variable int can assume values other than 14 during the execution of a program. Consider a second arithmetic assignment statement.

```
int = int + 1
```

Assuming that this assignment statement executes after the previous assignment statement in a program, the value of the variable int changes from 14 to 15. Refer to the "Assignment Statements" chapter for additional information.

You can specify an initial value for a variable in a program using the DATA statement. Refer to the "DATA Statements" chapter and the "BLOCK DATA" section for more information.

5.3. Arrays

An array is a sequence of variables that represent numeric, logical, or character values in a program. Each variable in the sequence is called an array element. Arrays can organize large amounts of data, particularly in complex programs, because they enable you to treat a group of related variables as a unit instead of as separate entities. Using arrays, you can reference a group of similar or related values with a single symbolic name.

Arrays consist of one or more dimensions. Dimensions enable you to organize data according to various criteria defined in the context of your program. For example, a program designed to analyze political opinion poll information might organize data according to three different criteria: the voter's age, precinct, and political party affiliation. Such a program could use a three dimensional array. Each element can have a data type comprised of several bytes.

5.3.1. Array Declarators and Subscripts

To declare an array in a program you must use an array declarator in a DIMENSION, COMMON, or type statement. An array declarator specifies a symbolic name to identify the array and a number of dimension declarators. The form for an array declarator is shown below. The abbreviation "dim" stands for dimension declarator.

```
symbolic-name (dim [,dim ]...)
```

Dimension declarators specify the number of elements in each array dimension that you declare. You set the number of elements with a lower- and upper-bound value. The lower- and upper-bound values are called dimension bounds. The form for a dimension declarator is as follows.

```
[lower-bound:] upper-bound
```

Dimension bounds can be arithmetic constant or variable expressions that evaluate to integers. The optional lower-bound value can be negative, zero, or positive. If you do not specify a lower-bound, a value of 1 is implied. The upper-bound value can be negative, zero, positive, or an asterisk indicating an assumed-size array declarator. The use of a variable expression as a dimension bound value constitutes an adjustable array declarator.

Adjustable and assumed-size array declarators are described later in this section.

The number of dimension declarators that you specify in an array declarator determines the number of dimensions for the array. An array in Fortran-386 can have a maximum of seven dimensions.

The size of an array is equal to the number of elements in the array. The number of elements is equal to the product of the dimension sizes specified in the array declarator.

The following example is an array declarator that declares an array with the symbolic name DISTANCES and one dimension declarator. DISTANCES is a one-dimensional array with a lower-bound of 10 and an upper-bound of 20. DISTANCES consists of 11 elements.

```
DISTANCES (10:20)
```

The next example is an array declarator that declares an array with the symbolic name AMPS and three dimension declarators. The dimension declarators do not include lower-bound specifications. Therefore, each dimension has an implied lower-bound of 1. AMPS is a three-dimensional array. The first two dimensions have an upper-bound of 8 and the third dimension has an upper-bound of 2. AMPS consists of 128 elements.

```
AMPS (8,8,2)
```

Each element in an array is a variable that can assume different values during program execution. A program can access the values in the array by referencing the element name in an expression. To reference an element name, you specify the name of the array followed by a subscript. A subscript consists of one or more arithmetic constant expressions enclosed in parentheses. Subscript expressions must evaluate to integers. An array element name has the following form. The abbreviation "sub-exp" stands for subscript expression.

```
symbolic-name (sub-exp [,sub-exp]...)
```

The number of subscript expressions that you specify in an element name reference must match the number of dimension declarators specified in the array declarator.

In certain situations, you can specify an array name without a subscript to reference the entire array. You can reference an array name without a subscript in the following Fortran-386 statements.

- type Statements
- COMMON Statement
- DATA Statement
- EQUIVALENCE Statement
- FUNCTION Statement
- SUBROUTINE Statement
- ENTRY Statement
- SAVE Statement
- Input/Output Statements

You can also use unsubscripted array names as actual arguments in a reference to a subroutine or external function.

5.3.2. One-dimensional Arrays

To demonstrate a one dimensional array, consider a simple program designed to calculate the average high meteorological temperature over a one week period. The program must store seven temperature readings, one for each day of the week, then calculate the average high. To store the temperatures, we declare an array using an array declarator. For this example, we declare a one-dimensional integer array with the symbolic name TEMPS.

To declare the integer data type we use the array declarator in an INTEGER type statement.

```
INTEGER TEMPS(7)
```

The number 7 enclosed in parentheses is the dimension declarator specifying an upper-bound of 7 elements. Notice that no lower-bound for the dimension is specified. Therefore, a lower-bound of 1 is implied. Refer to "Array Declarators and Subscripts" in this section for additional information.

Each element in TEMPS is a variable numbered 1 through 7 that can assume an assigned temperature value. You can assign initial values to array elements with the DATA statement. During program execution, you can assign values to array elements with assignment or input statements. The following figure represents the structure of array TEMPS with assigned values.

	SUN	MON	TUE	WED	THU	FRI	SAT
TEMPS	63	60	55	68	72	73	64
	(1)	(2)	(3)	(4)	(5)	(6)	(7)

The program can access any temperature value in the TEMPS array for a calculation by referencing the array element name. To reference an element name you specify the array name followed by a subscript. For example, TEMPS(3) refers to the third element in TEMPS, which has been assigned the value 55 degrees. The number 3 is a subscript expression. The number 3 including the parentheses constitutes the entire subscript. Remember, the number of subscript expressions that you specify in an element name reference must match the number of dimensions specified in the array declarator.

Each array element in TEMPS is a variable that can take on new values. Therefore, at the end of each week you can assign new temperature readings to the corresponding array elements and execute the program to calculate the average high for the new week.

5.3.3. Multi-dimensional Arrays

One-dimensional arrays organize data in linear form according to one criterion defined in the context of the program. In our hypothetical temperature program, the elements of array TEMPS correspond, one to one, with the days of the week. Multi-dimensional arrays enable you to organize data according to more than one criterion. For example, suppose we want to expand the temperature program to calculate the average body temperature for a medical patient over a one week period using three temperature readings taken each day instead of one. Now, we have two criteria upon which to organize the temperature values; the day of the week and the time of the day.

The new program must store twenty-one temperature readings, three for each day of the week, then calculate the average. To store the temperature values, we can declare a two-dimensional array using an array declarator. For this example, we declare the array with the symbolic name TEMPS2. The data type for TEMPS2, however, must be type REAL because body temperatures for medical patients must be accurate to a tenth of a degree. Therefore, to declare the REAL data type we use the array declarator in a REAL type statement.

```
REAL TEMPS2(7,3)
```

Notice that two dimension declarators are enclosed in parentheses following the symbolic name TEMPS2. The number 7 is the dimension declarator for the first dimension specifying an upper bound of seven elements. The number 3 is the dimension declarator for the second dimension specifying an upper bound of three elements. Both dimensions have an implied lower bound of 1.

You can assign initial values to the elements in TEMPS2 with the DATA statement. During program execution, you can assign values to array elements with assignment or input statements. The following figure represents the structure of array TEMPS2 with assigned values.

TEMPS2	SUN	MON	TUE	WED	THU	FRI	SAT	
6 A.M.	103.4	103.4	103.2	103.1	101.1	102.9	103.3	(1)
2 P.M.	103.0	102.3	100.4	99.7	99.1	98.8	98.7	(2)
10 P.M.	99.2	100.2	101.6	102.9	100.7	99.2	98.6	(3)
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	

Two-dimensional arrays organize data in grid form. The program can access any temperature value in the TEMPS2 array for a calculation by referencing the array element. To reference an element in a two-dimensional array, you must specify two subscript expressions after the array name. The subscript expressions serve as coordinates to locate values in the grid. For example, TEMPS2(4,2) refers to the fourth element in the first dimension and the second element in the second dimension. The reference points to the value 99.7 degrees taken at 2 P.M. on Wednesday.

5.3.4. Adjustable Array Declarators

All of the arrays in the previous examples are declared using constant array declarators. In a constant array declarator, each dimension bound is an integer constant or integer constant expression. In certain cases, you can specify dimension bounds using integer variables and integer variable expressions.

The use of a variable in a dimension bound specification constitutes an adjustable array declarator. Adjustable array declarators enable program units to pass various sized arrays as arguments. The adjustable array declarator serves as a dummy array in a subroutine or function procedure. The reference to the procedure contains the actual array size specifications. The adjustable array declarator must specify the same number of dimensions as the actual argument array.

The following example shows an adjustable array declarator named ADJ used in a DIMENSION statement. The DIMENSION statement is in a subroutine named TEST. Note that the adjustable array declarator name, ADJ, and the two variables used as dimension bound values, M and N, are dummy arguments for the TEST subroutine. M and N are the adjustable dimensions.

```
SUBROUTINE TEST(ADJ,M,N)
  .
  .
DIMENSION ADJ(M,N)
  .
  .
END
```

The following program unit example declares two actual arrays, ACT1 and ACT2, and contains two calls to the TEST subroutine defined above. Each call passes a different array to TEST for processing. The first CALL statement passes the array name ACT1 and the two dimension declarators 5 and 10 as actual arguments. The second CALL statement

passes the array name ACT2 and the dimension declarators 25 and 50 as actual arguments.

```
DIMENSION ACT1(5,10)
DIMENSION ACT2(25,50)
  .
  .
CALL TEST(ACT1,5,10)
  .
  .
CALL TEST(ACT2,25,50)
  .
  .
END
```

ACT1 and ACT2 are different sized arrays, but the adjustable array declarator, ADJ, enables the TEST subroutine to process both arrays one at a time.

5.3.5. Assumed-size Array Declarators

Assumed-sized array declarators, like adjustable array declarators, serve as dummy arrays in a function or subroutine procedure. An assumed-size array declarator uses an asterisk (*) as an upper dimension bound for the last dimension declared in the array. The actual upper dimension bound passes to the procedure from the procedure reference. Dimension bound values in an assumed-size array declarator other than the upper bound of the last dimension can be integer constant or variable expressions.

The following example shows an assumed-size array declarator named ASM used in a DIMENSION statement. The DIMENSION statement is in a function named CALC. In this example, the lower dimension bounds for both dimensions are integer constants. The upper bound for the first dimension is a variable, W. The upper bound for the second dimension is an asterisk. ASM serves as a dummy array within the CALC function. Note that the name ASM and the variable W used as an upper dimension bound for the first dimension are dummy arguments for the CALC function.

```
FUNCTION CALC(ASM,W)
  .
  .
DIMENSION ASM(1:W,1:*)
  .
  .
END
```

The following program unit example contains a reference to the CALC function defined above. The reference passes an actual array named ACT for processing. Note that the reference to CALC passes the array name ACT and the value 10 as actual arguments. The actual upper bound for the second dimension, 30, does not pass as an argument.

```
DIMENSION ACT(10,30)
  .
  .
VALUE = CALC(ACT,10)
  .
  .
END
```

The assumed-size array declarator, ASM, assumes the size of the actual array, ACT, passed to the CALC function in the function reference.

5.4. Substrings

A substring is a contiguous portion of the space that a character variable or character array element represents. Substring references enable you to manipulate segments of character strings in a program. A substring has a character data type.

Substring references have two forms: one for a character variable and one for a character array element. A substring reference for a character variable uses the following form.

variable-name([1st-expression]:[2nd-expression])

The character positions that a character variable represents are numbered from left to right, beginning with 1. The 1st-expression specifies the first or leftmost character of the substring that you want to reference. The 2nd-expression specifies the last or rightmost character of the substring that you want to reference. For example, the following substring reference specifies character positions three through seven in the character variable "materials".

materials(3:7)

The variable "materials" can take on a variety of values during program execution. However, the substring reference always specifies character positions three through seven regardless of the value of the variable at any given time. A substring reference for a character array element uses the following form:

array name (sub[, sub]...) ([1st-expression]:[2nd-expression])

Like a substring reference for a character variable, the character positions that a character array element represents are numbered from left to right, beginning with 1. The 1st-expression in the substring reference specifies the first or leftmost character of the substring that you want to reference. The 2nd-expression specifies the last or rightmost character that you want to reference. The abbreviation sub stands for subscript expression. You can specify any number of subscript expressions in a substring reference. For example, the following substring reference specifies character positions 5 through 10 of an element in a three dimensional character array named "products".

products (4,4,12) (5:10)

For both character variable and character array element references, the 1st-expression must be greater than or equal to 1 and less than or equal to the 2nd-expression. The 2nd-expression must be less than or equal to the length of the variable or array element. Expressions that do not evaluate to integers convert to integers.

If you omit the 1st-expression, a leftmost character position of 1 is implied. If you omit the 2nd-expression, a rightmost character position equal to the length of the variable or array element is implied. To omit both expressions implies a reference to all the character positions in the variable or array element. If you omit both expressions, you must still specify the colon enclosed in parentheses. The following examples show some different substring references.

```
versions(2:8)
items(5:)
fourth(3,9)(:11)
sysform(:)
```

CHAPTER 6

Expressions

An expression is a sequence of characters that specifies instructions for calculating a value. An expression can be a single basic data item, such as a constant or variable, or a combination of data items and operators. Operators are special characters that specify computations to perform using values specified in the expression. The values that an operator processes are called operands. All expressions represent, or evaluate to a single value. There are four kinds of expressions in Fortran-386:

- arithmetic
- character
- relational
- logical

6.1. Arithmetic Expressions

Arithmetic expressions represent numeric values. An arithmetic expression uses a special set of arithmetic operators, arithmetic operands, and parentheses to control the evaluation order of the operations specified in the expression. There are five arithmetic operators in Fortran-386 as shown in the following table.

Operator	Purpose
+	Addition
−	Subtraction or unary negation
*	Multiplication
/	Division
**	Exponentiation

The *, /, and ** operators work with two operands and are called binary operators. The + and − operators can work as binary operators or as unary operators that work on a single operand. The following table defines the interpretation of expressions using each of the arithmetic operators.

Interpretation of Arithmetic Operators	
Example	Interpretation
OP1 + OP2	Add OP1 and OP2.
+ OP1	Identify OP1 as positive.
OP1 − OP2	Subtract OP2 from OP1.
−OP1	Identify OP1 as negative.
OP1 * OP2	Multiply OP1 and OP2.
OP1 / OP2	Divide OP1 by OP2.
OP1 ** OP2	Raise OP1 to the power OP2.

There is a precedence among the arithmetic operators that determines the order in which operands are combined within an arithmetic expression that contains two or more

operators. The precedence among the arithmetic operators follows the standard rules of algebra and is shown in the following table, with the lowest number having the highest precedence.

Precedence Among Arithmetic Operators

Operator	Precedence
**	1
* and /	2
+ and –	3

When an expression contains two or more operators of equal precedence, such as * and /, Fortran-386 evaluates the operations algebraically from left to right. Exponentiation, however, is evaluated from right to left. For example, consider the following expression.

```
OP1 ** OP2 ** OP3
```

Fortran-386 first calculates OP2 raised to the power indicated by OP3, then calculates OP1 raised to the power indicated by the value that results from the first calculation.

You can use parentheses to control the evaluation order of the operations specified in an arithmetic expression. The portions of an expression you enclose in parentheses are evaluated first. The use of parentheses supercedes the precedence among arithmetic operators. The following examples demonstrate the evaluation order of operations within arithmetic expressions according to operator precedence and the use of parentheses.

```
25 + 15 – 10 + 12 = 42          (25 + 15) – (10 + 12) = 18
  ↑    ↑    ↑                     ↑     ↑     ↑
 1st  2nd  3rd                   1st   3rd   2nd

3  + 4  * 3  – 5 = 10           3  + 4  * (3 – 5) = –5
  ↑    ↑    ↑                     ↑     ↑     ↑
 2nd  1st  3rd                   3rd   2nd   1st

100 – 10 ** 2 * 3 = –200        (100 – 10 ** 2) * 3 = 0
   ↑    ↑    ↑                      ↑     ↑     ↑
  3rd  1st  2nd                    2nd   1st   3rd

11 + 3 * 2 – 10 / 2 = 12        ((11 + 3) * 2 – 10) / 2 = 9
  ↑   ↑   ↑    ↑                    ↑     ↑   ↑     ↑
 3rd 1st 4th  2nd                  1st   2nd 3rd   4th
```

In the preceding examples, all the operands are unsigned numeric constants. You can, however, use operands in a variety of forms within an arithmetic expression. An arithmetic operand can be any of the following entities.

- an unsigned numeric constant
- a symbolic name of an unsigned numeric constant
- a numeric variable reference
- a numeric array element reference
- an arithmetic function reference
- an arithmetic expression enclosed in parentheses

The data type of an arithmetic expression is determined by the data types of the operands specified in the expression. An arithmetic expression that contains operands of one data type evaluates to a value of the same data type. An arithmetic expression that contains operands of two or more different data types evaluates to a value of the highest

ranking type in the expression. Operands of low ranking data types automatically convert to the higher ranking types. The one exception to this strict hierarchy of data type conversions is that the combination of a REAL*8 (DOUBLE PRECISION) value with a COMPLEX*8 (COMPLEX) value causes both operands to be converted to COMPLEX*16 (DOUBLE COMPLEX). The following table defines the hierarchy of data types for conversion within expressions.

Arithmetic Data Type Conversion Hierarchy

Data Type	Rank
COMPLEX*16	1 (Highest)
COMPLEX*8 (COMPLEX)	2
REAL*8 (DOUBLE PRECISION)	3
REAL*4 (REAL)	4
INTEGER*4	5
INTEGER*2 (INTEGER)	6
INTEGER*1	7
LOGICAL	8 (Lowest)

According to the hierarchy for arithmetic data type conversion, an arithmetic expression that consists of real and complex operands evaluates to a complex value. An arithmetic expression that consists of INTEGER*2 and INTEGER*4 operands evaluates to an INTEGER*4 value, and so on. Here are some rules that apply to the conversion of data types within arithmetic expressions.

- In an expression that contains integer and real operands, the integer operands first receive a fractional component of 0 in the conversion to real. Then, Fortran-386 evaluates the expression using real arithmetic. Consider the expression (10/5)*3.14. Note that Fortran-386 first evaluates the integer division, (10/5), then converts the result of the division to real.
- To convert an operand of one REAL data type to a REAL data type with a higher precision, Fortran-386 uses the existing operand as the most significant portion of the higher precision data item and the least significant part of the data item becomes zero. Fortran-386 then evaluates the expression using the higher precision arithmetic.
- In an expression that contains complex and integer operands, the integer operands convert to real as described above. The converted real operand then serves as the real part of a complex number and the imaginary part becomes zero. Fortran-386 then evaluates the expression using complex arithmetic. The expression evaluates to a complex value.
- Any fractional component that results from a division of integers is truncated, not rounded. For example, the expression (1/2 + 1/2) is equal to 0, not 1. The expression (12/5) is equal to 2. The fractional components are truncated.

6.2. Character Expressions

Character expressions enable you to manipulate character strings. A character expression uses character operands and a special character operator. All character expressions evaluate to a single character string value.

The character operator consists of two slashes, //, and is called the concatenation operator. The concatenation operator simply joins two character operands together. For example, the following character expression evaluates to the character string values 'FOR-TRAN' and 'Fortran-386', respectively.

```
'FOR' // 'TRAN'
'FOR' // 'TRAN' // '-386'
```

In the preceding examples, all the operands are character constants. You can, however, use operands in a variety of forms within a character expression. A character operand can be any of the following entities.

- a character constant
- a symbolic name of a character constant
- a character variable reference
- a character array element reference
- a character substring reference
- a character expression enclosed in parentheses
- a character function reference

The use of parentheses does not affect the value of a character expression. For example, the following character expressions are equivalent.

```
'PRESS' // 'ANY KEY' // 'TO CONTINUE'
('PRESS' // 'ANY KEY') // 'TO CONTINUE'
'PRESS' // ('ANY KEY' // 'TO CONTINUE')
```

The length of a character expression equals the sum of the lengths of the individual operands. The length value includes any spaces that are parts of an operand. For example, the following expression has a length of 19.

```
'ENTER' // 'ÞYOURÞPASSWORD'
```

Parentheses are not parts of an operand and are not included in the length value.

6.3. Relational Expressions

A relational expression compares the values of two operands using a special set of relational operators. A single relational expression can compare only two arithmetic expressions or two character expressions. A relational expression cannot compare an arithmetic expression with a character expression. Relational expressions evaluate to a logical value, either true or false, depending on the comparison condition.

The relational operators test for certain relationships that exist between two operands. There are five relational operators in Fortran-386 as shown in the following table. Note that the periods that delimit each relational operator are required for use in an expression.

Relational Operators	
Operator	Meaning
.LT.	less than
.LE.	less than or equal to
.EQ.	equal to
.NE.	not equal to
.GT.	greater than
.GE.	greater than or equal to

In an arithmetic relational expression, Fortran-386 first evaluates the arithmetic operands, then compares the resulting values to determine if the relationship specified by the relational operator exists. Consider the following arithmetic expression.

```
100-50  .GT.  100/50
  ↑             ↑
  50            2
```

Fortran-386 first evaluates the arithmetic expressions on either side of the .GT. operator. Following this first evaluation, the relational expression states that 50 is greater than 2. Fortran-386 then evaluates the validity of the expression. Since 50 is greater than 2, the value of the relational expression is true.

If we change the operator in the preceding example to .LT. (less than), the value of the relational expression becomes false, because 50 is not less than 2. You can use parentheses within the arithmetic operands of a relational expression to alter the evaluation order.

In a character relational expression, Fortran-386 first evaluates the character operands, then compares the resulting values to determine if the relationship specified by the relational operator exists. Consider the following character expression.

```
'APP' // 'LE' .LT. 'AP' // 'RICOT'
```

Fortran-386 first evaluates the character expressions on either side of the .LT. operator. Following this first evaluation, the relational expression states that the string 'APPLE' is less than the string 'APRICOT'.

For character relational expressions, operands are evaluated according to the ASCII character collating sequence. The length of the character operands is not significant for comparison. If the two character operands have different lengths, the shorter operand is padded on the right with blank characters until the two strings are equal. Fortran-386 then compares the two strings, a character at a time according to the ASCII collating sequence. Under ASCII conventions, all characters correspond to numeric values that determine a hierarchy among the characters. Refer to the "ASCII and Hexadecimal Conversions" section for a listing of the collating sequence.

In the preceding example, the string 'APPLE' has a length of 5 and the string 'APRICOT' has a length of 7. Therefore, Fortran-386 pads 'APPLE' on the right with two blank characters to make the strings equal in length. Then, Fortran-386 compares the two strings a character at a time. The first two letters in both strings, AP, are equal. The third letter in each string, however, is different. 'APPLE' has a P and 'APRICOT' has an R. According to the collating sequence, P is less than R. Therefore, the string 'APPLE' is less than the string 'APRICOT' and the relational expression is true.

Fortran-386 can compare complex expressions only with the .EQ. and .NE. operators. Two complex values are considered equal only if their corresponding real and imaginary parts are both equal.

A relational operator can compare two numeric expressions of different data types. Fortran-386 converts the value of the expression with the lower ranked data type to the higher ranked data type before making the comparison. When a REAL*8 and a COMPLEX*8 value are compared, the values are first converted to the type COMPLEX*16 before making the comparison.

All the Fortran-386 relational operators have equal precedence. However, arithmetic and character operators have a higher precedence than relational operators. Therefore, Fortran-386 evaluates arithmetic and character operators before relational operators.

6.4. Logical Expressions

Logical expressions compare logical values (true and false) using a special set of logical operators. The operands in a logical expression evaluate to logical values. You can use parentheses to control the evaluation order of the operations specified in a logical expression. All logical expressions evaluate to a single logical value. There are five logical operators in Fortran-386 as shown in the following table.

Logical Operators	
Operator	Meaning
.NOT.	logical negation
.AND.	logical conjunction
.OR.	logical inclusive disjunction
.XOR.	logical exclusive disjunction
.EQV.	logical equivalence
.NEQV.	logical nonequivalence

The interpretation of expressions formed using each of the logical operators is shown in the following table. OP1 denotes an operand for a unary operator or an operand to the left of a binary operator. OP2 denotes an operand to the right of a binary operator.

Interpretation of Logical Operators	
Example	Interpretation
OP1 .AND. OP2	The expression is true only if both OP1 and OP2 are true.
OP1 .OR. OP2	The expression is true if either OP1 or OP2, or both, is true.
OP1 .EQV. OP2	The expression is true only if both OP1 and OP2 have the same logical value: either true or false.
OP1 .NEQV. OP2	The expression is true if OP1 is true and OP2 is false or if OP2 is true and OP1 is false. The expression is false if both operands have the same logical value.
.NOT. OP1	The expression is true only if OP1 is false.

There is a precedence among the logical operators that determines the order in which operands are compared within a logical expression that contains two or more operators. The following table shows the precedence among the logical operators, with the lowest number having the highest precedence.

Precedence Among Logical Operators	
Operator	Precedence
.NOT.	1
.AND.	2
.OR.	3
.EQV. and .NEQV.	4

When an expression contains two or more operators of equal precedence, such as .EQV. and .NEQV., Fortran-386 evaluates the operations algebraically from left to right. You can use parentheses to control the evaluation order of the operations specified in a

logical expression. The following examples demonstrate the evaluation order of operations within logical expressions according to operator precedence and the use of parentheses.

```
6*3+ 4 .LT. 25 .AND. 4 .LE. 8/2  (evaluates to true)
6*(3+ 4) .LT 25 .AND. 4 .LE. 8/2  (evaluates to false)
```

Two logical operators cannot appear consecutively within an expression unless the second operator is .NOT. The following example is a valid expression.

```
3.14159 .LE. 10 .AND. .NOT. 10/5 .EQ. 3  (evaluates to true)
```

In the preceding examples, all the operands are arithmetic relational expressions. You can, however, use operands in a variety of forms within a logical expression. A logical operand can be any of the following entities.

- an arithmetic relational expression
- a character relational expression
- a logical constant (.TRUE. or .FALSE.)
- a symbolic name of a logical constant
- a logical variable reference
- a logical array element reference
- a logical function reference

6.5. Fortran-386 Operator Precedence

There is a precedence among all four kinds of Fortran-386 operators that determines the order in which operands are combined in expressions that contain more that one kind of operator. The following table shows the precedence among the four kinds of operators, with the lowest number having the highest precedence.

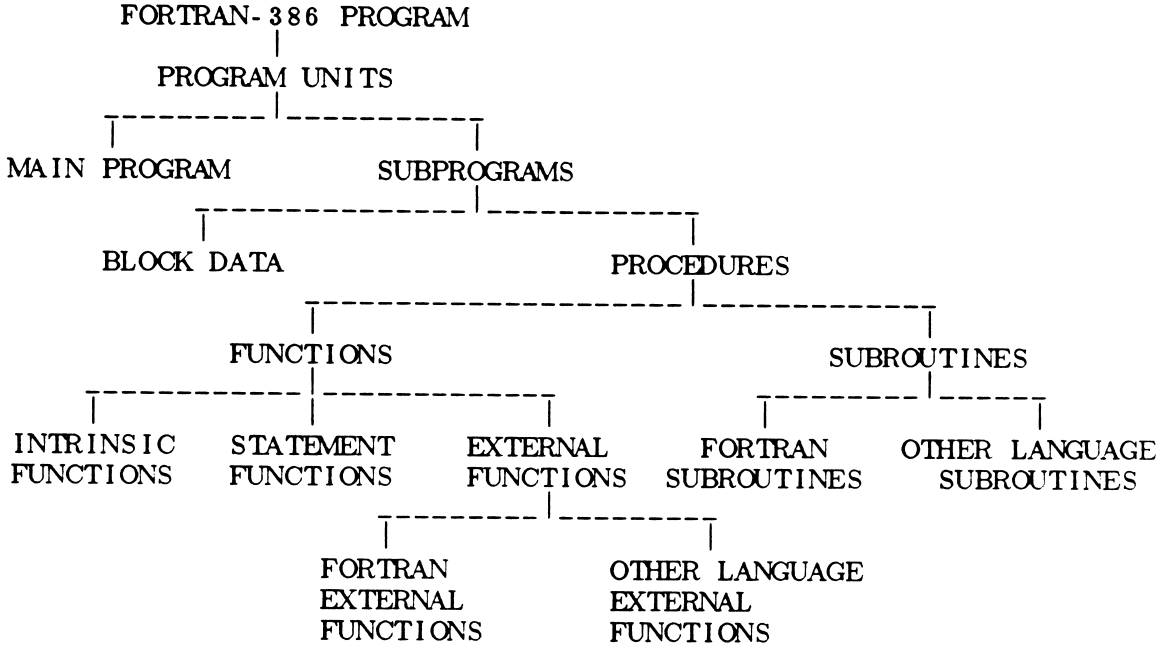
Fortran-386 Operator Precedence	
Operator	Precedence
Arithmetic	1
Character	2
Relational	3
Logical	4

Remember, you can use parentheses to control the evaluation order of the operations specified in an expression. The use of parentheses supercedes operator precedence.

CHAPTER 7

Fortran-386 Program Structure

An executable Fortran-386 program consists of one or more program units. A program unit is a logical, self-contained sequence of statements and optional comment lines that forms a discrete part of a larger program. There are two kinds of program unit: main programs and subprograms. This section explains the main program, the different classes of subprogram, and the relationships among program units that combine to form Fortran-386 executable programs. The diagram below illustrates Fortran-386 program structure.



Fortran-386 Program Structure

7.1. Main Program

The main program serves as the center or base of all processing activity in an executable program. The main program is the program unit that receives control to begin execution. During execution, the main program can invoke a variety of subprograms that perform different tasks. Control returns to the main program to terminate execution except when a STOP statement executes. See the END and STOP statements for more information on program termination.

An executable Fortran-386 program can consist of a main program with no subprograms. Each executable program can have only one program unit defined as the main program. Subprograms cannot call or reference the main program and the main program cannot call or reference itself.

Use the PROGRAM statement to define a program unit as a main program. The PROGRAM statement is not required, but if used, it must be the first statement of the main program. The PROGRAM statement specifies a symbolic name for the main program as shown in the following syntax specification.

PROGRAM symbolic-name

The main program can contain any of the Fortran-386 statements except BLOCK DATA, FUNCTION, SUBROUTINE, ENTRY and RETURN statements. A STOP or END statement terminates execution of the program. The SAVE statement has no effect within the main program.

7.2. Subprograms

The term subprogram covers two kinds of program unit: block data subprograms and procedures. Block data subprograms initialize variables and array elements, and cannot contain executable statements. Procedures contain executable statements that define a specific computing operation. Subprograms cannot call or reference the main program. An executable Fortran-386 program must contain only one main program but can contain any number of subprograms.

7.3. Block Data

A block data subprogram enables you to specify initial values for variables and array elements that are listed in named common blocks. Common blocks are areas of storage containing data that different program units can share. Any program unit that contains a definition of a given common block can use the data in that block. Use the COMMON statement to define common blocks.

Common blocks can be either named or unnamed. However, only variables and array elements in named common blocks can be initialized in a block data subprogram. You can initialize data from several named common blocks in one block data subprogram. Refer to the COMMON statement for more information on common storage area management.

You identify block data subprograms with the BLOCK DATA statement. The BLOCK DATA statement has the following form.

BLOCK DATA [symbolic-name]

The symbolic-name is a global reference for the subprogram. The name is optional, but if used it must be unique. There cannot be more than one unnamed block data subprogram in one executable program. The BLOCK DATA statement must be the first statement in a block data subprogram.

A block data subprogram can contain only certain specification statements: COMMON, DIMENSION, DATA, EQUIVALENCE, IMPLICIT, PARAMETER, SAVE, and type statements. You can also include comment lines. The last statement in a block data subprogram must be the END statement. An executable Fortran-386 program can contain any number of block data subprograms. Block data subprograms have the following form.

```
BLOCK DATA [symbolic-name]
.
. specification statements and comments
.

END
```

The following block data subprogram example defines three named common blocks for a hypothetical program that calculates weather information: time, location, and

conditions. The example specifies a data type for each variable in the named common blocks and declares initial values for some of the variables.

```
      BLOCK DATA weather
C Define named common areas
      COMMON /time/day,hour,min  /location/zone,altitude
      COMMON /conditions/temp,humidity,pressure,wind,rain
C Declare data types for the variables
      INTEGER day,hour,min,zone,altitude,temp,humidity,wind
      REAL pressure
      LOGICAL rain
C Declare initial values for some of the variables
      DATA day/31/, hour/24/, zone/7/, altitude/5285/
      DATA temp/65/, pressure/29.92/, rain/.true./
      END
```

Notice that you must specify all the variables in the named common blocks using specification statements even if you do not declare initial values for all the variables with the DATA statement.

7.4. Procedures

A procedure is a subprogram that performs a specific task within an executable program. Unlike block data subprograms, procedures contain the executable statements that define the purpose of the program. Procedures structure programs into a series of routines that can execute repeatedly, in any order, as required to accomplish a larger task. This process of breaking up a large programming task into a series of smaller, logically individual procedures is the fundamental principle of structured programming. An executable Fortran-386 program can contain any number of procedures.

A procedure receives execution control through a call or reference. Procedures can receive control from the main program or from another procedure. However, procedures cannot call or reference the main program. Procedures can share values through the use of arguments and common blocks.

There are two classes of procedure within Fortran-386: subroutines and functions. Subroutines and functions differ primarily in the method by which they are invoked during execution and in the specific result that they produce. Functions are further classified into three different categories: external functions, statement functions, and intrinsic functions. Subroutines and external functions are collectively referred to as external procedures.

7.4.1. Procedure Arguments

Arguments supply the values that a procedure requires to produce the desired result. A program unit, such as the main program, can invoke a procedure to perform a specific task using arguments to pass the values that the procedure needs to complete the task. Both subroutine and function procedures can change the values of the arguments during execution. Therefore, values that the procedure produces can return to the invoking program unit via the arguments. Arguments are sometimes called parameters.

Two kinds of argument must be distinguished: dummy or formal arguments and actual or calling arguments. Dummy arguments are used in the procedure definition to reserve a place and declare a data type for actual values that the procedure requires to produce the desired result. Actual arguments are used in the procedure call or reference and are substituted for the dummy arguments during the actual procedure execution.

Dummy arguments used in the procedure definition must correspond in number, order, and data type with the actual arguments in the procedure call or reference.

Depending on the kind of procedure, a dummy argument can be a variable name, an array name, a dummy procedure name, or an alternate return specifier.

A variable name that serves as a dummy argument can only be associated with an actual argument that is a variable, a constant, a symbolic name of a constant, a function reference, an array element, a substring, or an expression.

An array name that serves as a dummy argument can only be associated with an actual argument that is an array, array element, or array element substring of matching data type.

7.5. Subroutines

A subroutine is an external procedure that performs a specific task within an executable Fortran-386 program. An external procedure is a distinct program unit that is defined outside of the program unit that invokes it. You can write external procedures using a programming language other than Fortran-386, such as C or assembly language. Refer to the "Interfacing FORTRAN and C" chapter and the "80386 Target" chapter for more information.

Use the SUBROUTINE statement to define a program unit as a subroutine procedure. The SUBROUTINE statement must be the first statement in the subroutine. The SUBROUTINE statement specifies a symbolic name for the subroutine and a list of dummy arguments the subroutine requires to produce the desired result. Remember, dummy arguments reserve a place and specify a data type for the actual arguments supplied in the subroutine call. The following syntax specification shows the general format for a SUBROUTINE statement.

SUBROUTINE symbolic-name [(dummy [, dummy] ...)]

A dummy argument in a SUBROUTINE statement can be one of the following items.

- a variable name
- an array name
- a dummy procedure name
- an asterisk (alternate return specifier)

A dummy procedure name enables actual procedure names to be passed as arguments.

A subroutine can receive control of execution from the main program or from another procedure. A subroutine cannot invoke itself. Execution control transfers to a subroutine through the CALL statement. The CALL statement must specify the symbolic-name of the subroutine you want to invoke and a list of actual arguments that the subroutine requires to produce the desired result. Actual arguments specified in the CALL statement must correspond in number, order, and data type to the dummy arguments specified in the SUBROUTINE statement. Refer to the "CALL Statement" section for more information.

Actual arguments in a CALL statement can be one of the following items.

- an expression
- an array name
- an intrinsic function name
- an external procedure name
- a dummy procedure name
- an alternate return specifier using the statement label of an executable statement in the same program unit as the CALL statement

7.6. Functions

A function is a procedure that performs a specific task within an executable Fortran-386 program. Execution control transfers to a function through a reference. You cannot use the CALL statement to invoke a function. To reference a function means to use the function name within an expression. Functions can receive control of execution from the main program or from another procedure. A function cannot reference itself.

Unlike a subroutine, a function is specifically designed to return a single value to the program unit that contains the function reference. The function assigns this return value to the function name. Therefore, the return value becomes the value of the function. When the function name appears in an expression, the value of the function is used in the evaluation of the expression. The function name determines the data type for the return value.

There are three classes of functions: external, statement, and intrinsic. specify the symbolic-name and any actual arguments the function requires to produce the desired result. Use the following general format to reference a function.

symbolic-name [(actual [, actual]...)]

Actual arguments used in the function reference must correspond in number, order, and data type with the dummy arguments in the function definition. Actual arguments in the function reference can be any one of the following items.

- an expression
- an array name
- an intrinsic function name
- an external procedure name
- a dummy procedure name

7.6.1. External Functions

An external function, like a subroutine, is a distinct program unit that is defined outside of the program unit that invokes it. Remember, you can write external procedures using a programming language other than Fortran-386, such as C or assembly language. Refer to the "Interfacing FORTRAN and C" chapter and the "80386 Target" chapter for more information.

Use the FUNCTION statement to define a program unit as an external function procedure. The FUNCTION statement must be the first statement in the external function. The FUNCTION statement specifies a symbolic name for the external function, a data type for the value the function returns, and a list of dummy arguments the function requires to produce the desired result. Remember, dummy arguments reserve a place and specify a data type for the actual arguments supplied in the function reference. The following syntax specification shows the general format for a FUNCTION statement.

type FUNCTION symbolic-name [(dummy [, dummy] ...)]

A dummy argument in a FUNCTION statement can be one of the following items.

- a variable name
- an array name
- a dummy procedure name
- an asterisk (alternate return specifier)

7.6.2. Statement Functions

A statement function is a procedure that is completely defined in a single statement. Unlike an external function, you can only reference a statement function within the program unit that contains the statement function definition. A statement function consists of a symbolic-name to identify the function, a list of dummy arguments the function needs to produce the desired result, and an expression as shown in the following syntax specification.

symbolic-name ([dummy [, dummy] ...]) = expression

A statement function is structured much like an assignment statement. Therefore, the data type of the expression converts to the type of the symbolic-name, if necessary, according to the rules for conversion in assignment as described in the "Assignment Statements" chapter. The symbolic-name is the statement function name.

The dummy arguments reserve a place and declare a data type for actual values that the statement function requires to produce the desired result. Actual arguments are used in the statement function reference and are substituted for the dummy arguments during the actual procedure execution. The dummy arguments in a statement function are local to the statement function. Therefore, you can use the dummy argument names to represent other entities in the same program unit. You cannot use the statement function name to represent another entity within the same program unit.

Statement functions are referenced in an expression. Actual arguments specified in a statement function reference must correspond in number, order, and data type with the dummy arguments in the statement function definition.

Other functions can be referenced within the expression of a statement function. However, the functions that you reference in a statement function expression must be defined before that statement function in the same program unit. The definition of a statement function and all references to that statement function must be in the same program unit.

7.7. Alternate Return Specifiers

A CALL statement transfers execution control to a subroutine. A RETURN or END statement transfers execution control from the subroutine back to the program unit that contains the CALL statement. A normal return from a subroutine is when execution control transfers to the first executable statement following the CALL statement in that calling program unit. You can, however, specify alternate return points from subroutines using alternate return specifiers. Alternate return specifiers operate through the passing of arguments between a calling program unit and a subroutine.

An alternate return specifier consists of an asterisk (*) that you specify as a dummy argument to a subroutine in a SUBROUTINE or ENTRY statement. The corresponding actual argument used in a CALL statement must be a statement label. The statement label identifies an executable statement that serves as an alternate return point for the subroutine. You can specify any number of alternate return specifiers for a subroutine.

The RETURN statement has an optional integer expression used to select one alternate return specifier from a series of specifiers. The integer expression indicates which asterisk in the SUBROUTINE or ENTRY statement dummy argument list to use for the return. A valid integer expression must be greater than or equal to 1 and less than or equal to the number of asterisks specified in the SUBROUTINE or ENTRY statement dummy argument list. Remember, each asterisk in the dummy argument list corresponds to an actual argument supplied in the CALL statement that invokes the subroutine. The actual arguments must be statement labels that indicate the alternate return points.

For example, the following hypothetical subroutine contains three RETURN statements. The first RETURN statement, statement 110, specifies a normal return from the subroutine because there is no specified integer expression. The second and third RETURN statements, statements 120 and 130, have integer expressions indicating alternate returns.

Statement 100 is an arithmetic IF statement that determines which of the three RETURN statements will execute. The SUBROUTINE statement contains two alternate return asterisks that serve as dummy arguments to the subroutine named thrust.

```
      SUBROUTINE thrust (var1, *, *, var2)
        .
        .
        .
100     IF (dat/val) 110, 120, 130
110     RETURN
120     RETURN 1
130     RETURN 2
```

If the expression in the arithmetic IF statement evaluates to less than zero, the first RETURN statement, statement 110, executes. The first RETURN statement specifies a normal return. If the expression evaluates to zero, the second RETURN statement, statement 120, executes. Note that the second RETURN statement specifies the integer 1 indicating the first alternate return asterisk in the dummy argument list. If the expression evaluates to greater than zero, the third RETURN statement, statement 130, executes. Note that the third RETURN statement specifies the integer 2 indicating the second alternate return asterisk in the dummy argument list.

The following CALL statement calls the thrust subroutine. Note that the second and third actual arguments in the CALL statement are statement labels that correspond to the two dummy argument asterisks in the SUBROUTINE statement.

```
CALL thrust (2.86, *200, *300, 4.13)
```

If the second RETURN statement in the subroutine executes, execution control transfers to the statement identified with the label 200. This happens because the first dummy argument asterisk in the SUBROUTINE statement corresponds to the actual argument 200 in the CALL statement. If the third RETURN statement in the subroutine executes, execution control transfers to the statement identified with the label 300. This happens because the second dummy argument asterisk corresponds to the the actual argument 300. Statement 200 and 300 are alternate return points selected on the basis of the arithmetic IF statement.

Note that if the integer expression used in a RETURN statement is less than 1 or greater than the number of asterisks in the dummy argument list, a normal return from the subroutine is executed.

CHAPTER 8

The Fortran-386 Input/Output System

Fortran-386 provides a device-independent input/output (I/O) system for transferring data. The I/O system can transfer data from one location to another within a processor's memory. It can also transfer data to and from processor memory and any external device such as a console, printer, or a storage medium such as a disk, floppy disk, or magnetic tape.

This section describes the I/O system in general, as well as the related topics of records, files, I/O units, and data transfer. "The Input/Output Statements" chapter describes the syntax of each Fortran-386 I/O statement in detail.

8.1. Records

A record is any logically-related set of data items. There are three kinds of records:

- formatted
- unformatted
- endfile

A formatted record is a sequence of ASCII characters; an unformatted record is a sequence of items having any combination of data types. An endfile record is the last record in a file, and is written using the ENDFILE statement. Refer to the "File Positioning Statements" section for additional information.

The length of a formatted record is the number of characters it contains, and depends on the number of characters written to the record when it is created. The length of an unformatted record is specified when it is created. The length of either kind of record can be zero. The physical length of either kind of record is measured in bytes.

Formatted records can only be accessed with formatted I/O statements; unformatted records can only be accessed with unformatted I/O statements. Refer to the "Data Transfer" section for more information.

8.2. Files

A file is a sequence of records. The records in a file are either all formatted or all unformatted. A file cannot contain both kinds of records.

Each record in a file has a unique record-number, which is an integer that the I/O system assigns to the record when it is created.

There are two categories of files:

- external
- internal

An external file contains data that can be transferred between internal storage and any external device. Also, an external file can be permanently stored on some external storage medium, such as a floppy disk, hard disk, or magnetic tape.

An internal file is an area of internal storage. An internal file is implemented as a character variable, a character array, or an element of a character array.

The physical size of a file is the number of records in the file multiplied by the length of record, measured in bytes.

8.3. I/O Units

An I/O unit is a logical or generic designation for a file. That is, at any given time an I/O unit can designate one of several different files.

Internally, the Fortran-386 I/O system uses I/O units when transferring data or manipulating files, so the I/O statements described below generally refer to I/O units rather than names of files.

An I/O unit is designated by an unsigned integer in the range 0 to 99.

8.3.1. Connection

Connection is the property of an I/O unit that defines the relationship between the I/O unit and a file. An I/O unit is connected when it refers to a specific file; an I/O unit is disconnected when it does not refer to a specific file.

Connection is a symmetric property. That is, when an I/O unit is connected to a file, the file is also connected to the I/O unit. An I/O unit cannot be connected to more than one file at a time, nor can a file be connected to more than one I/O unit at a time.

All Fortran-386 I/O statements, except OPEN and CLOSE operate on I/O units that are connected to files on a one-to-one basis. No data transfer can take place on a file unless it is connected to an I/O unit. The I/O unit/file connection must be made when the file is opened, or by preconnection.

8.3.2. Preconnection

An I/O unit is preconnected if, by some means external to the program, it is connected to a specific file before the program begins to run. Each Fortran-386 program has three preconnected I/O units:

- Unit 0 - the default error output
- Unit 5 - the default console input
- Unit 6 - the default console output

8.4. Properties of Files

Each file has an associated set of properties, some of which are determined by the OPEN statement at the time of connection to an I/O unit.

8.4.1. Existence

Existence is a file property that describes the potential an executable program has to access the file. That is, at any given time the processor is aware of a set of files, some of which can be accessed by a program, and some of which cannot.

The files that a program can potentially access are said to exist for that program. The files that a program cannot access do not exist. For example, a file could be protected for security reasons, or a file could be in use by another program. In such a case, the file is inaccessible to the program, and therefore does not exist.

The property of existence only applies to a file in relation to a particular executable program. It does not apply to the file's physical existence within the environment of an implementation.

There are four possible combinations of connection and existence.

1. A file can exist and be connected (a disk file being written to).
2. A file can exist and not be connected (a disk file that is not yet open).
3. A file can be connected but not exist (a newly created disk file before the first record is written).
4. A file can both not exist and not be connected (a disk file that was erased).

8.4.2. Access Method

There are two methods for accessing a file:

- sequential access
- direct access

With sequential access, you access the records in the same order that they were created. With direct access, you can access the records in any order.

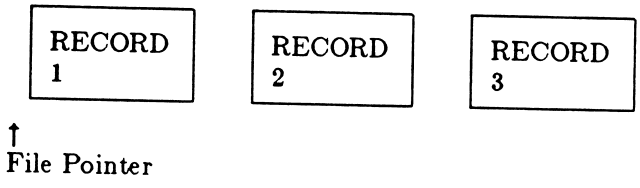
You can access an external file using either access method, but you can only access an internal file using sequential access.

If a file is connected for direct access, all the records must have the same length. A file connected for sequential access can have records with different lengths.

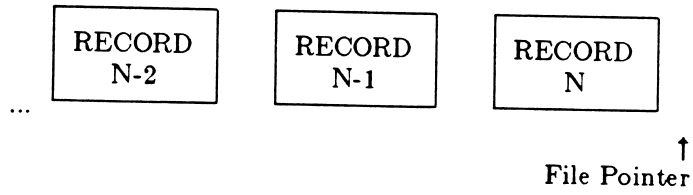
8.4.3. Position

When a file is connected to an I/O unit, the file has the property of position. At any given time, the file position is determined by a file pointer. The file pointer is not a physical entity; rather, it is an internal reference the I/O system uses to keep track of the file position.

The following diagrams illustrate the concept of file position.

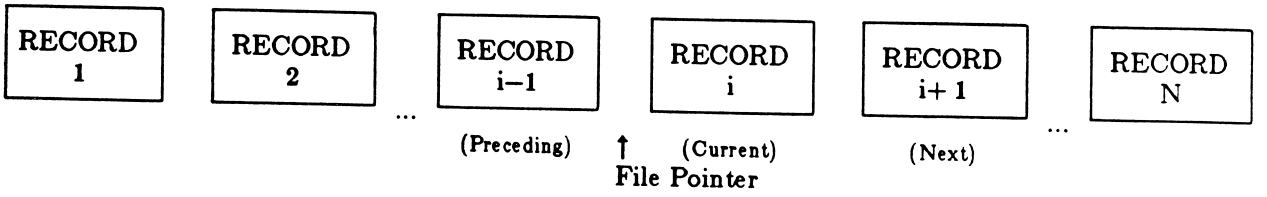


In the diagram above, the file is positioned at the initial point, which is just before the first record.



In the diagram above, the file is positioned at the terminal point, which is just after

the last record.



In the diagram above, the file is positioned at a specific record. This is the current record. If the file is not positioned at or within a record, there is no current record.

If a file contains N records, and the file pointer points to record i, where $1 \leq i \leq N$, then record i+1 is the next record, and record i–1 is the preceding record. If i=1, there is no preceding record. If i=N or if N=0, there is no next record.

8.5. Categories of I/O Statements

Fortran-386 provides three categories of I/O statements:

- data transfer statements
- file positioning statements
- auxiliary statements

Data transfer statements move data between internal (processor) storage and a file. Data transfer statements can reference both internal and external files.

File positioning statements affect the position of the file pointer relative to a specific file. File positioning statements cannot reference internal files.

Auxiliary I/O statements manipulate the connection of I/O units to external devices and media, and inquire about the characteristics of a particular connection. Auxiliary statements cannot reference internal files.

The "Input/Output Statements" chapter describes the syntax of each I/O statement in detail.

8.5.1. Data Transfer Statements

The data transfer statements are

- READ
- WRITE
- PRINT
- ENCODE
- DECODE

The READ statement inputs data from a specific I/O unit. The WRITE statement outputs data to a specific I/O unit. The PRINT statement outputs data to the default output I/O unit.

If a READ or WRITE statement contains a format specifier it is a formatted I/O statement. Otherwise, it is an unformatted I/O statement. Refer to the "Data Transfer" section for additional information.

The ENCODE and DECODE statements transfer data between internal locations only. DECODE translates data from external character form to internal binary representation. ENCODE translates data from internal binary representation to external character form. Both ENCODE and DECODE translate the data using a format specifier.

Using formatted READ and WRITE statements with internal files achieves the same results as ENCODE and DECODE. Fortran-386 supports ENCODE and DECODE only for

compatibility with older FORTRAN compilers.

8.5.2. File Positioning Statements

The file positioning statements are

- BACKSPACE
- REWIND
- ENDFILE

The BACKSPACE statement moves the file pointer to the start of the preceding record. The REWIND statement moves the file pointer to the initial point of a file. The ENDFILE statement marks the preceding record as the last record in a file.

All three file positioning statements require that the file be connected for sequential access.

8.5.3. Auxiliary I/O Statements

The auxiliary I/O statements are

- OPEN
- CLOSE
- INQUIRE

The OPEN statement can

- create a file and connect it to an I/O unit.
- create a preconnected file.
- connect an existing file to an I/O unit.
- change the characteristics of an existing I/O unit/file connection.

The CLOSE statement disconnects a file from an I/O unit.

The INQUIRE statement returns information about the characteristics of a named file, or the connection of a file to a particular I/O unit.

8.6. Data Transfer

Data transfer is the process of moving data between records and individual items specified in an I/O- list. An I/O- list is a list of data items whose values are transferred by a data transfer statement.

The I/O system performs the following steps each time it executes a data transfer statement:

1) determines the direction of the data transfer.
2) identifies the I/O unit.
3) establishes the format (if one is specified).
4) positions the file before the transfer.
5) performs the transfer between the file and the I/O- list.
6) positions the file after the transfer.
7) sets the I/O status specifier (if one is defined).

There are two categories of data transfer:

- formatted
- unformatted

Formatted data transfer requires formatted I/O statements; unformatted data transfer requires unformatted I/O statements.

8.7. Formatted Transfer

During formatted data transfer, the I/O system transfers data between a file and the I/O- list, and performs editing on the data. The current record and (optionally) other records are read from or written to.

8.7.1. Editing

The editing performed during the transfer is directed by a format specification. A format is a description of the arrangement or pattern that data has when it is read or written. The same data can be read or written with different formats to suit different situations.

A format specification is a list of items separated by commas and enclosed in parentheses. The list is called a format list, and contains items called edit descriptors. A format list can also contain other (nested) format lists.

There are two kinds of edit descriptors:

- repeatable
- nonrepeatable

Repeatable edit descriptors control the editing of character, logical, and numeric data. Each item in the I/O- list corresponds to a repeatable edit descriptor in the format list.

Each repeatable edit descriptor can be preceded by an integer constant called a repeat-factor, which tells the I/O system how many times to repeat the edit specified in the edit descriptor.

Nonrepeatable edit descriptors are not associated with specific data items in the I/O- list. Instead, they control such things as column position, spacing, sign control, blank control, and line termination.

8.7.2. Format Control

The interaction between the I/O- list and the format specification is a dynamic process called format control.

Format control always proceeds from left to right matching each item in the I/O- list with the next repeatable edit descriptor. Format control executes nonrepeatable edit descriptors as they are encountered.

If an edit descriptor has a repeat-factor, format control processes the I/O- list as if it contained the specified number of consecutive items.

If the format list ends before reaching the end of the I/O- list, format control reverts to the beginning of the last nested format list, if there is one. If there is none, format control reverts to the beginning of the format-specification and again passes through the I/O- list. Each time format control reverts, it accesses a new record.

8.7.3. List-directed Formatting

List-directed formatting is an alternative method of formatted data transfer. List-directed formatting is specified by using an asterisk (*) as the format specification in an I/O statement. A FORMAT statement is not required.

When the I/O system executes a list-directed READ, WRITE, or PRINT statement, it begins a new record, and formats each input or output value using the data type and field width of the corresponding I/O- list item to generate an equivalent edit descriptor.

The "Format Statement and Format Specification" chapter completely describes rules for writing valid format specifications, the actions of the various edit descriptors, and list-

directed formatting.

8.8. Unformatted Transfer

During unformatted data transfer, the I/O system transfers data between the current record of a file and the I/O- list. The I/O system does not perform any editing of the data, and only one record is read from or written to.

CHAPTER 9

Structural Statements

The Fortran-386 structural statements enable you to define the different kinds of program units that make up a Fortran-386 program. A program unit is a logical, self-contained sequence of statements and optional comment lines that forms a discrete part of the larger program. All Fortran-386 programs consist of one or more program units. There are five statements that you can use to structure a Fortran-386 program.

- PROGRAM Statement
- BLOCK DATA Statement
- SUBROUTINE Statement
- FUNCTION Statement
- ENTRY Statement

9.1. BLOCK DATA Statement

The BLOCK DATA statement identifies a program unit as a block data subprogram. A block data subprogram is an nonexecutable program unit that enables you to specify initial values for variables and array elements in named common blocks.

The BLOCK DATA statement can specify a symbolic name for the block data subprogram as shown in the following syntax specification.

Syntax:

BLOCK DATA [symbolic-name]

The symbolic name is optional. If you specify a symbolic name, it must not be the same as the symbolic name of the main program, an external procedure, a common block, or another block data subprogram within the same executable program. The symbolic name must not be the same as any local name used in the block data subprogram.

A block data subprogram can contain only certain specification statements: COMMON, DIMENSION, DATA, EQUIVALENCE, IMPLICIT, PARAMETER, SAVE, and type statements. You can also include comment lines. The last statement in a block data subprogram must be the END statement.

An executable Fortran-386 program can contain any number of block data subprograms. Refer to the "Block Data" section for additional information.

Examples:

```
BLOCK DATA
BLOCK DATA initial
```

9.2. ENTRY Statement

Use an ENTRY statement to specify a secondary entry point in a procedure. A secondary entry point enables execution of the procedure to begin with an executable statement other than the first executable statement in the procedure. You can place an ENTRY statement anywhere after the FUNCTION statement in an external function procedure or after the SUBROUTINE statement in a subroutine procedure. A procedure can contain more than one ENTRY statement.

The ENTRY statement specifies a symbolic name, to identify the entry point in the procedure, and a list of dummy arguments. The dummy arguments hold a place and specify a data type for the actual arguments specified in the procedure reference.

Syntax:

ENTRY symbolic-name [(dummy[, dummy] ...)]

You cannot use an ENTRY statement between a block IF statement and the corresponding END IF statement, or between a DO statement and the last statement of the DO loop.

Use the CALL statement to reference a secondary entry point in a subroutine. To reference a secondary entry point in a function, use the entry point name in an expression. The dummy argument list in an ENTRY statement need not match the dummy argument list in the SUBROUTINE or FUNCTION statement at the beginning of the procedure. However, the actual arguments specified in an entry point reference must correspond in number, order, and data type with the dummy arguments specified in the corresponding ENTRY statement.

Example:

```
ENTRY enter2 (pressure, volume)
```

9.3. FUNCTION Statement

Use the FUNCTION statement to define a program unit as an external function procedure. An external function, like a subroutine, is a distinct program unit defined outside of the program unit that invokes it.

The FUNCTION statement specifies a symbolic name for the external function, a data type for the value the function returns, and a list of dummy arguments. Dummy arguments reserve a place and specify a data type for the actual arguments supplied in the function reference.

Syntax:

type FUNCTION symbolic-name [(dummy[, dummy] ...)]

The type specification can be any one of the standard data types described in the "Data Types" chapter. The type specification determines the data type of the value that the function returns.

A dummy argument in a FUNCTION statement can be any one of the following items.

- a variable name
- an array name
- a dummy procedure name
- an asterisk (alternate return specifier)

The symbolic name of a function serves as a variable name to hold the value that the function returns. The symbolic name must be used as a variable name within the function procedure. The value of the variable name when a RETURN or END statement executes (the last value assigned to the function name) is the value of the function. An external function can use any of its dummy arguments to return values in addition to returning the value of the function.

An external function can receive control of execution from the main program or from another procedure. Execution control transfers to an external function through a reference in an expression. Refer to the "Functions" section for more information.

You can write external functions using a programming language other than Fortran-386, such as C or an assembly language. Refer to the "Interfacing FORTRAN and C" chapter for more information.

Examples:

```
INTEGER FUNCTION reset(arg1, arg2, arg3)
DOUBLE PRECISION FUNCTION molarity(atomwt, grams)
```

9.4. PROGRAM Statement

Use the PROGRAM statement to define a program unit as the main program. The main program is the program unit that receives control at the beginning of execution. During execution, the main program can invoke a variety of subprograms that perform different tasks. Control returns to the main program to terminate execution unless you use a STOP statement in a subprogram.

The PROGRAM statement specifies a symbolic name for the main program as shown in the following syntax specification.

Syntax:

PROGRAM symbolic-name

The PROGRAM statement is not required, but if used, it must be the first statement of the main program. Refer to the "Main Program" section for additional information.

In the example below, the PROGRAM statement specifies a main program with the symbolic name "main".

Example:

```
PROGRAM main
```

9.5. SUBROUTINE Statement

Use the SUBROUTINE statement to define a program unit as a subroutine procedure. A subroutine is an external procedure. This means that a subroutine is a distinct named program unit defined outside of the program unit that invokes it.

The SUBROUTINE statement specifies a symbolic name for the subroutine and a list of dummy arguments. Dummy arguments reserve a place and specify a data type for the actual arguments supplied in the subroutine call.

Syntax:

SUBROUTINE symbolic-name [(dummy[, dummy] ...)]

A dummy argument in a SUBROUTINE statement can be any one of the following items.

- a variable name
- an array name
- a dummy procedure name
- an asterisk (alternate return specifier)

A subroutine can receive control of execution from the main program or from another procedure. Execution control transfers to a subroutine through the CALL statement. A subroutine cannot invoke itself. Refer to the "Subroutines" section for additional information.

You can write subroutines using a programming language other than Fortran-386, such as C or an assembly language. Refer to the "Interfacing FORTRAN and C" chapter for more information.

Example:

```
SUBROUTINE calculations (arg1, arg2, arg3)
```

CHAPTER 10

Specification Statements

The Fortran-386 specification statements enable you to define data types, establish the interpretation and use of symbolic names, and control the management of storage. Specification statements are nonexecutable statements that appear most often at the beginning of a program unit, before the executable statements. There are eight kinds of specification statements in Fortran-386:

- type statements
- DIMENSION statement
- COMMON statement
- EQUIVALENCE statement
- IMPLICIT statement
- PARAMETER statement
- EXTERNAL statement
- INTRINSIC statement
- SAVE statement

Type statements specify a data type explicitly for symbolic names that represent constants, variables, arrays, external functions, and statement functions. Once a symbolic name appears in a type statement, the data type of that symbolic name is defined for the entire program unit. An explicit data type specification confirms or supercedes the implicit data type convention. You can also use type statements to specify the dimensions of an array. There are eight type statements in Fortran-386.

- INTEGER type statement
- REAL type statement
- DOUBLE PRECISION type statement
- COMPLEX type statement
- LOGICAL type statement
- CHARACTER type statement

Refer to the "Data Types" chapter for additional information on data types.

10.1. INTEGER Statement

Use the INTEGER statement to specify an integer data type for symbolic names that represent integer constants, variables, arrays, external functions, and statement functions.

Syntax:

INTEGER[*number] symbolic-name[, symbolic-name] ...

The optional asterisk and number that follows the keyword INTEGER specifies the number of bytes each integer value occupies in memory. In Fortran-386, you can specify integers that are 1, 2, or 4 bytes long. If you do not specify the asterisk and number, integers default to 4 bytes. A two byte integer can represent values that range from −65536 to 65535. A one byte integer can represent values that range from −128 to 127.

You can also use the -i2 compile time option to specify that the default size for integers (and logicals) is 2 bytes. Refer to "Compile Time Options" chapter for more information on compiler option switches.

In the first example below, by default, each variable and each element in the array occupies 4 bytes of storage. However if the -i2 compile time option is used each variable and each element in the array will occupy 2 bytes.

Examples:

```
INTEGER var1, var2, array(10)
INTEGER*4 var1, var2, array(10)
INTEGER*1 var1, var2, array(10)
INTEGER*2 var1, var2, array(10)
```

10.2. REAL Type Statement

Use the REAL statement to specify a real data type for symbolic names that represent real number constants, variables, arrays, external functions, and statement functions.

Syntax:

REAL[*number] symbolic-name[, symbolic-name] ...

The optional asterisk (*) and number that follows the keyword REAL specifies the number of bytes each real number value occupies in memory. In Fortran-386, you can specify real values that are 4 or 8 bytes long. Specifying 8 byte real values using a REAL*8 type statement is equivalent to using the DOUBLE PRECISION type statement.

If you do not specify the asterisk and number, real numbers default to 4 bytes. The range, precision, and representation of real numbers is given in the "80386 Target" chapter.

In the first two examples below, each variable and each element in the array occupy 4 bytes of storage. The first two examplex are equivalent. The third example specifies 8 byte real values.

Examples:

```
REAL var1, var2, array(10)
REAL*4 var1, var2, array(10)
REAL*8 var1, var2, array(10)
```

10.3. DOUBLE PRECISION Type Statement

Use the DOUBLE PRECISION statement to specify a double precision real data type for symbolic names that represent real number constants, variables, arrays, external functions, and statement functions. Using the DOUBLE PRECISION type statement is equivalent to specifying 8 byte real values using a REAL*8 type statement.

Syntax:

DOUBLE PRECISION symbolic-name[, symbolic-name] ...

In Fortran-386, a double precision real number occupies 8 bytes of storage. The range, precision, and representation of double precision real numbers is given in the "80386 Target" chapter.

In the following example the DOUBLE PRECISION type statement specifies two double precision variables and a ten element double precision array. Each variable and each element in the array occupies 8 bytes of storage.

Example:

```
DOUBLE PRECISION var1, var2, array(10)
```

10.4. COMPLEX Type Statement

Use the COMPLEX statement to specify a complex data type for symbolic names that represent complex constants, variables, arrays, external functions, and statement functions.

Syntax:

COMPLEX[*number] symbolic-name[, symbolic-name] ...

The optional asterisk (*) and number that follows the keyword COMPLEX specifies the number of bytes each complex value occupies in memory. In Fortran-386, you can specify complex values that are 8 or 16 bytes long. If you do not specify the asterisk and number, complex numbers default to 8 bytes.

In the first two examples below, each variable and each element in the array occupies 8 bytes of storage. The first two examples are equivalent.

Examples:

```
COMPLEX  var1, var2, array(10)
COMPLEX*8  var1, var2, array(10)
COMPLEX*16  var1, var2, array(10)
```

Refer to the "Complex Type" section for more information on complex numbers.

10.5. LOGICAL Type Statement

Use the LOGICAL statement to specify a logical data type for symbolic names that represent constants, variables, arrays, external functions, and statement functions.

Syntax:

LOGICAL[*number] symbolic-name[, symbolic-name] ...

The optional asterisk and number that follows the keyword LOGICAL specifies the number of bytes each logical value occupies in memory. In Fortran-386, you can specify integers that are 1, 2, or 4 bytes long. If you do not specify the asterisk and number, logicals default to four bytes.

You can also use the -i2 compile time option to specify that the default size for logicals (and integers) is 2 bytes. Refer to "Compile Time Options" chapter for more information on compiler option switches.

In the first example below, by default, each variable and each element in the array occupies 4 bytes of storage. However if the -i2 compile time option is used each variable and each element in the array will occupy 2 bytes.

Examples:

```
LOGICAL var1, var2, array(10)
LOGICAL*4 var1, var2, array(10)
LOGICAL*1 var1, var2, array(10)
LOGICAL*2 var1, var2, array(10)
```

10.6. CHARACTER Type Statement

Use the CHARACTER statement to specify a character data type for symbolic names that represent constants, variables, arrays, external functions, and statement functions.

Syntax:

CHARACTER[*number] symbolic-name[*number] [, symbolic-name[*number]] ...

The asterisk and number that follows the keyword CHARACTER specifies the length or number of characters for each character item that you list in the statement. Alternatively, you can specify the length of each character item individually with an asterisk and number that immediately follows each symbolic name. If you do not specify the length of the character items using either method, the length of the items default to a value of one.

In certain cases, you can specify a length with an asterisk enclosed in parentheses, (*). An asterisk enclosed in parentheses indicates that a dummy argument assumes the length specification from the corresponding actual argument or that a function name obtains its length specification from the function reference. If you use an asterisk enclosed in parentheses as a length specifier for the symbolic name of a character constant, the symbolic name assumes the actual length of the constant that it represents.

Each variable and array element declared in the first three examples below has a length of 1 character. The three examples are equivalent.

In the last example below, var1 has a length of 15 characters, var2 and each element in the array have a length of 5 characters.

Examples:

```
CHARACTER var1, var2, array(10)
CHARACTER*1 var1, var2, array(10)
CHARACTER var1*1, var2*1, array(10)*1
CHARACTER*5 var1*15, var2, array(10)
```

10.7. COMMON Statement

Use the COMMON statement to define common blocks. Common blocks are contiguous areas of storage containing data that different program units can share. When you declare common blocks of the same name in different program units, the blocks all share the same storage space when the program units are combined into an executable program.

A COMMON statement specifies symbolic names enclosed in slashes to identify each common block and lists of variable names, array names, and array declarators that represent the values in each common block that you specify.

Syntax:

COMMON [/symbolic-name/] common list [[,]/[symbolic-name]/ common-list] ...

All items that you place in a common list are contained in the common block identified by the symbolic-name that precedes the list. You must delimit all the items in a common list with commas. The slashes serve to delimit each symbolic-name/common list pair that you specify in the COMMON statement. You can also use commas as secondary delimiters.

If you do not specify a symbolic-name, all items in the corresponding common list are declared to be in an unnamed (blank) common block. A program can only have one unnamed common block. If you omit the first symbolic-name in a COMMON statement that specifies more than one common block, the slashes are optional. Otherwise, slashes are required, with or without a symbolic-name, to delimit one common block specification from another.

COMMON statements are frequently used in a block data subprogram. A block data subprogram enables you to specify initial values for variables and array elements that are listed in named common blocks. Refer to the "Block Data" section for additional information on block data subprograms.

Note that the unnamed block in the last example below contains var1, var2, and var3, as well as the array names array1 and array2.

Examples:

```
COMMON  /atomic/var1,var2,array
COMMON  /com1/var1,var2,array1/com2/var3,var4,array2
COMMON var1,var2/com1/var8,array3,/com2/var12,var13,//var3,array1,array2
```

10.8. DIMENSION Statement

Use a DIMENSION statement to declare arrays in a program unit. A DIMENSION statement can contain any number of array declarators. An array declarator specifies a symbolic name to identify the array and a number of dimension declarators. Each array declarator that you use in a DIMENSION statement specifies a different array. In the following syntax specification, the abbreviation dim stands for dimension declarator.

Syntax:

DIMENSION symbolic-name (dim[,dim]...) [symbolic-name (dim[,dim]...)] ...

Dimension declarators specify the number of elements in each array dimension that you declare. You set the number of elements with a lower- and upper-bound value. The lower- and upper-bound values are called dimension bounds. The form for a dimension declarator is as follows.

[lower-bound:]upper-bound

Dimension bounds can be arithmetic constant or variable expressions that evaluate to integers. The optional lower-bound value can be negative, zero, or positive. If you do not specify a lower-bound, a value of 1 is implied. The upper-bound value can be negative, zero, positive, or an asterisk indicating an assumed-size array declarator. The use of a variable expression as a dimension bound value constitutes an adjustable array declarator. Refer to the "Adjustable Array Declarators" and "Assumed-size array declarators" section for more information.

The number of dimension declarators that you specify in an array declarator determines the number of dimensions for the array. An array in Fortran-386 can have a maximum of seven dimensions.

The size of an array is equal to the number of elements in the array. The number of elements is equal to the product of the dimension sizes specified in the array declarator.

Examples:

The following DIMENSION statement uses one array declarator to specify an array named values. The values array is two-dimensional. The first dimension has a lower bound of −12 and an upper bound of 12. The second dimension has an implied lower bound of 1 and an upper bound of 5. values consists of 125 elements.

```
DIMENSION values (-12:12, 5)
```

The next DIMENSION statement uses three array declarators to specify arrays named arr1, arr2, and arr3. All three are one-dimensional arrays with a lower bound of 0 and an upper bound of 19. Each array consists of 20 elements.

```
DIMENSION arr1(0:19), arr2(0:19), arr3(0:19)
```

10.9. EQUIVALENCE Statement

The EQUIVALENCE statement enables two or more program entities to share the same memory space. You can use the EQUIVALENCE statement to conserve memory space or to specify two or more symbolic names for the same program entity.

Syntax:

EQUIVALENCE (item- list)[, (item-list)] ...

You can specify variable names, array names, array element names, or character substrings in an EQUIVALENCE statement item- list. All the program entities in one item- list share the same starting address in memory, even if the length of the items differ. Each item- list must contain at least two items.

You can specify program entities of different data types in an EQUIVALENCE statement. For example, you can specify an integer variable and a complex variable in one item- list. The result is that the integer variable shares storage with the real portion of the complex variable. No data type conversion takes place.

Examples:

In the following example, the EQUIVALENCE statement specifies that the double precision variable named DOUBVAR and the first two elements of an integer array named INTARR occupy the same storage units.

```
INTEGER*2 INTARR(5)
DOUBLE PRECISION DOUBVAR
EQUIVALENCE (INTARR(1),DOUBVAR)
```

In the next example, the EQUIVALENCE statement specifies that the first five characters of two character variables share the same storage units.

```
CHARACTER CODE*5, ZONE*12
EQUIVALENCE (CODE,ZONE)
```

10.10. EXTERNAL Statement

Use the EXTERNAL statement to specify external procedure names and dummy procedure names for use as actual arguments. If you want to use an external or dummy procedure name as an actual argument in a program unit, you must specify the name in an EXTERNAL statement.

Syntax:

EXTERNAL symbolic-name[, symbolic-name] ...

A symbolic-name cannot represent an intrinsic function. Use the INTRINSIC statement to specify intrinsic functions as actual arguments.

Each symbolic name that you specify can appear only once in all the EXTERNAL statements that a program unit contains.

In the example below, the EXTERNAL statement declares three symbolic names as representing external procedures. The three external procedure names can be used as actual arguments.

Example:

```
EXTERNAL alpha, bravo, delta
```

10.11. IMPLICIT Statement

In the absence of an explicit data type specification, symbolic names that begin with the letters I, J, K, L, M, or N have an implied integer data type. Symbolic names that begin with any other letter of the alphabet have an implied real (REAL*4) data type. Use the IMPLICIT statement to change or confirm the implicit data type convention.

Syntax:

IMPLICIT data-type (letter– list)[, (letter-list)] ...

The IMPLICIT statement assigns the specified data type to symbolic names that begin with the letters you list in the statement. The data-type can be any one of the standard Fortran-386 data types. A letter– list can be a list of single letters or ranges of letters. A range specification consists of the first and last letters in the range separated with a minus sign. You must specify the range letters in alphabetical order. You cannot use the same letter, singularly or in a range specification, more than once in the IMPLICIT statements for one program unit.

You can specify a length for a CHARACTER type data as shown in the following examples. The length value must be an unsigned integer constant or an integer constant expression enclosed in parentheses. If you do not specify a length, a value of one is implied.

A program unit can contain more than one IMPLICIT statement. However, IMPLICIT statements must precede all other specification statements in a program unit except PARAMETER statements.

Examples:

The following IMPLICIT statements confirm the implicit data type convention.

```
IMPLICIT INTEGER(I, J, K, L, M, N)
IMPLICIT REAL(A–H, O–Z)
```

The next IMPLICIT statement specifies that all symbolic names beginning with the letters A, B, C, D, or E have an implied data type of INTEGER*2. Note that the implicit data type convention remains in effect for the other letters of the alphabet.

```
IMPLICIT INTEGER*2(A, B, C, D, E)
```

The next IMPLICIT statements specify character data types. Symbolic names that begin with letters ranging from A to M and S to Z have a CHARACTER data type and a length of 10.

```
IMPLICIT CHARACTER*10(A–M, S–Z)
```

10.12. INTRINSIC Statement

Use the INTRINSIC statement to specify intrinsic function names for use as actual arguments. If you want to use an intrinsic function name as an actual argument in a program unit, you must specify the name in an INTRINSIC statement.

Syntax:

INTRINSIC symbolic-name[, symbolic-name] ...

The "Fortran-386 Intrinsic Functions" chapter describes each of the individual Fortran-386 intrinsic functions with examples.

In the example below, the INTRINSIC statement declares three symbolic names as representing intrinsic functions. The three intrinsic function names can be used as actual arguments.

Examples:

```
INTRINSIC exp, tan, sqrt
```

10.13. PARAMETER Statement

Use the PARAMETER statement to assign symbolic names to constants. Once you assign a symbolic name to a constant using the PARAMETER statement, you can use the symbolic name in place of the actual constant anywhere in the program unit.

Syntax:

PARAMETER (symbolic-name = expression [, symbolic-name = expression] ...)

The PARAMETER statement assigns the symbolic-name on the left of the equal sign to the value of the expression on the right of the equal sign.

Symbolic names that have an arithmetic data type can only be assigned to arithmetic constants. Symbolic names that have a logical or character data type can only be assigned to logical or character constants, respectively.

Examples:

The following PARAMETER statement assigns the symbolic name maxnum to the integer constant 65535. Note that the name maxnum has an implicit integer data type.

```
PARAMETER (maxnum = 65535)
```

The next PARAMETER statement assigns the symbolic name faraday to the real constant 9.6487E−2 and the symbolic name bohr to the real constant 5.2917E−9. Both names have implicit real data types.

```
PARAMETER (faraday = 9.6487E−2, bohr = 5.2917E−9)
```

10.14. SAVE Statement

Execution of a RETURN or END statement in a procedure causes all program entities in that procedure to become undefined except for the following:

- entities in a blank (unnamed) common block
- entities that are initially defined but do not become redefined or undefined in the procedure
- entities in a named common block that appear in the procedure and in at least one other program unit that references the procedure

Use the SAVE statement to retain the definition status of a program entity following the execution of a RETURN or END statement.

Syntax:

SAVE [symbolic-name [, symbolic-name] ...]

A symbolic-name in a SAVE statement can represent a variable, an array, or a named common block. The name of a common block must be enclosed in slashes. A SAVE statement that does not contain explicit name specifications implies the saving of all appropriate program entities in the program unit that contains that SAVE statement.

Entities specified in SAVE statements for one program unit do not become undefined when a RETURN or END statement executes in that program unit. However, if the entities are in a common block, they might become undefined in another program unit.

You cannot use the names of procedures, entities within a common block, or dummy arguments in a SAVE statement.

Examples:

The following SAVE statement specifies that three program entities retain their definition status following the execution of a RETURN or END statement in the program unit. Note that the name of a common block is enclosed in slashes.

```
SAVE var1, array1, /block/
```

The next SAVE statement specifies the saving of all appropriate program entities in the program unit that contains the SAVE statement.

```
SAVE
```

CHAPTER 11

The DATA Statement

The DATA statement assigns initial values to variables, arrays, array elements, and substrings. An initial value is a value assigned to a program entity at the beginning of program execution. An entity that is not assigned an initial value at the beginning of execution is recognized as undefined.

DATA statements are nonexecutable. You can use a DATA statement anywhere in a program unit after any specification statements that the program unit contains.

Syntax:

DATA name- list /constant-list/ [[,] name-list /constant-list/] ...

A name- list consists of one or more variable names, array names, array element names, substring names, and implied DO lists. Use a substring name in a DATA statement to initialize a portion of a character string. Use an implied-DO loop in a DATA statement to initialize a portion of an array. You must separate the names in the name- list with commas.

You cannot use the names of functions, entities in blank common, or dummy arguments in a DATA statement name- list. You can use the names of entities in named common blocks in a DATA statement name- list if the DATA statement is in a block data subprogram.

A constant- list consists of constants, symbolic names for constants, and constants preceded by a factor. A constant preceded by a factor specifies multiple, successive appearances of the constant in a constant- list. Use the following form to write a constant preceded by a factor.

factor * constant

The factor must be a nonzero, unsigned integer constant or the symbolic name of such a constant. The constant that follows the asterisk can be zero, a signed or unsigned constant, or a symbolic name. The following examples show two equivalent constant- lists.

```
/3*100/
/100, 100, 100/
```

Note that you must delimit each constant- list with slashes and you must separate the constants in the constant- list with commas. You can optionally delimit each name- list/constant-list pair with commas.

The DATA statement assigns the constant values in each constant- list to the entities specified in the preceding name- list. The DATA statement assigns the values consecutively, one by one, as they appear in each list. The number of constants in a constant- list must correspond exactly to the number of entities specified in the preceding name- list. If you specify an unsubscripted array name in a name- list, you must specify a constant for each element of the array in the constant- list.

The DATA statement in the following example declares initial values for two variables, an array, and a substring.

Example:

```
INTEGER array1(3)
REAL var1
LOGICAL var2
CHARACTER subst*5
DATA var1, var2 /1.02E3,.TRUE./ array1,subst /3*100,'total'/
```

11.1. Type Conversion in DATA Statements

An entity in a name- list and the corresponding constant in a constant-list must both have either numeric or character data types. When the name- list entity and the corresponding constant have numeric data types that differ, the data type of the constant converts to the type of the name- list entity. Conversion takes place according to the Fortran-386 rules for arithmetic conversion described in the "Assignment Statements" chapter.

When the length of a character entity in a name- list is greater than the corresponding character constant, the DATA statement initializes the extra characters in the entity with blanks. When the length of an entity in a name- list is less than the corresponding character constant, the DATA statement ignores the extra characters in the entity.

Examples:

In the following example, the real constant assigned to the integer variable, intvar, is converted to the integer 3. The integer constant 128 assigned to the real variable, realvar, is converted to the real number 128.0.

```
INTEGER intvar
REAL realvar
DATA intvar, realvar /3.14159,128/
```

In the next example, the DATA statement ignores the last character in the character constant Challenger because the character variable substr1 has a length of only 9 characters. The DATA statement initializes the last four characters in subst2 with blanks because the corresponding character constant, orbit, only has five characters.

```
CHARACTER*9 subst1, subst2
DATA subst1, subst2 /'Challenger','orbit'/
```

11.2. Implied-DO in DATA Statements

You can use an implied-DO loop in a DATA statement to initialize a portion of an array. The implied-DO loop repeats the initialization process in a DATA statement for the array elements that you specify.

Syntax:

(do list, variable = initial,limit [,increment])

The do list specifies the array elements that you want to initialize. The variable, referred to as the implied-DO variable, assumes each iteration value in the range specified using the initial and limit values. The variable must be an integer variable. The iteration count specified with the initial and limit values must proceed in a positive direction. The optional increment value specifies an iteration step. If you omit the increment value, a value of 1 is implied.

Examples:

The DATA statement in the following example uses an implied-DO loop to initialize elements 10 through 20 of a 50 element integer array. The statement initializes a total of 11 elements with the constant 100.

```
INTEGER array(50)
DATA (array(i), i = 10, 20)/11*100/
```

The DATA statement in the next example uses an implied-DO loop to initialize every other element from (1,1) through (100,199) in a 20,000 element real array. Note the use of the increment value, 2, for iteration control over the second array dimension. The statement initializes a total of 10,000 elements with the real value 3.14159.

```
REAL array(100,200)
DATA ((array(k,m), k=1, 100), m=1, 200,2)/10000*3.14159/
```

CHAPTER 12

Assignment Statements

Assignment statements assign values to variables, arrays, array elements, and substrings. The assignment statement evaluates an expression and assigns the result of the evaluation to the entity. Once an assignment statement assigns a value to an entity, that entity is recognized as defined. There are four kinds of assignment statements in Fortran-386:

- arithmetic
- logical
- character
- ASSIGN

12.1. Arithmetic Assignment Statements

Use an arithmetic assignment statement to assign the value of an arithmetic expression to an arithmetic variable or array element. Place the arithmetic expression on the right side of an equal sign and the name of the arithmetic entity on the left as shown in the following syntax specification.

Syntax:

symbolic-name = arithmetic-expression

Upon execution, an arithmetic assignment statement first evaluates the arithmetic-expression according to the rules for expression evaluation described in Section 5. Then, the statement assigns the result of the evaluation to the entity that the symbolic-name identifies.

If the entity on the left of the equal sign has the same data type as the expression on the right, the statement assigns the value directly. If the data types differ, the value of the expression converts to the data type of the entity on the left before the assignment takes place. Assignment statements use certain Fortran-386 intrinsic functions to perform the conversion of data types. The table below lists the generic intrinsic function names used for conversion in arithmetic assignment statements. Refer to the "Fortran-386 Intrinsic Functions" chapter for additional information.

Functions for Arithmetic Type Conversion	
Variable or Array Element Data Type	Generic Intrinsic Function Name
integer	INT(expression)
real	REAL(expression)
double precision	DBLE(expression)
complex	CMPLX(expression)

Examples:

The arithmetic assignment statement in the following example assigns the real constant 6.02E23 to the real variable avogadro.

```
REAL avogadro
avogadro = 6.02E23
```

The next arithmetic assignment statement evaluates an expression, converts the result of the expression to an integer, and assigns the integer value to the first element in the integer array element ionic(25).

```
INTEGER ionic(25)
ionic(1) = 28.43/14.32**(-3)
```

12.2. Logical Assignment Statements

Use a logical assignment statement to assign the value of a logical expression to a logical variable or array element. Place the logical expression on the right side of an equal sign and the name of the logical entity on the left, as shown in the following syntax specification.

Syntax:

symbolic-name = logical-expression

Upon execution, a logical assignment statement first evaluates the logical-expression. Then, the statement assigns the result of the evaluation to the entity that the symbolic-name identifies. Note that a logical expression must evaluate to a logical value, either true or false. Refer to the "Logical Expressions" section for additional information.

Examples:

The following logical assignment statement assigns the logical constant .false. to the logical variable switch.

```
switch = .false.
```

The next logical assignment statement first evaluates a logical expression, then assigns the result to the logical variable prnout.

```
prnout = var1/var7 .GT. 128 .AND. var2/var8 .LT. 128
```

12.3. Character Assignment Statements

Use a character assignment statement to assign the value of a character expression to a character variable, array element, or substring. Place the character expression on the right side of an equal sign and the name of the character entity on the left as shown in the following syntax specification.

Syntax:

symbolic-name = character-expression

If the length of the character-expression is less than the length of the character entity, the statement pads the expression on the right with blank characters.

If the length of the character-expression is greater than the length of the character entity, the statement truncates the expression from the right.

Assigning a value to a character substring does not affect any characters in the character

variable or array element that are outside the substring reference. Any character positions in a character entity that are outside the substring reference remain unchanged whether or not the positions were defined or undefined.

Examples:

The character assignment statement in the following example assigns the character constant Sirius to the character variable starname.

```
CHARACTER starname*12
starname = 'Sirius'
```

The character assignment statement in the next example evaluates a character expression, then assigns the result to the character array element compound(5).

```
CHARACTER*8 compound(25)
compound(5) = 'Na' // 'Cl'
```

The character assignment statement in the last example assigns the character constant mix67 to the substring var1(2:6). Note that character positions in the character variable var1 that are outside of the substring reference remain unchanged after the assignment takes place.

```
CHARACTER var1*7
var1(2:6) = 'mix67'
```

12.4. ASSIGN Statement

Use the ASSIGN statement to assign a statement label value to an integer variable. This enables the variable name to be used as a transfer specification in an assigned GOTO statement, or as a format specifier in a formatted I/O statement.

Syntax:

ASSIGN statement-label TO symbolic-name

The statement-label must identify an executable statement or a FORMAT statement. The executable statement or the FORMAT statement must be in the same program unit as the ASSIGN statement.

The symbolic-name must represent an integer variable. Once the variable becomes defined for reference as a statement label, it becomes undefined for use as an integer variable.

The ASSIGN statement must execute before any statements containing a reference to the assigned variable name. You cannot specify arithmetic operations involving a variable that represents a statement label.

Examples:

The following ASSIGN statement assigns the statement label 300 to the integer variable trans. The statement defines trans as a statement label variable.

```
INTEGER trans
ASSIGN 300 TO trans
```

You can redefine trans in another assignment statement. The following statement returns trans to its status as an integer variable. After this statement executes, you can no longer use trans in an assigned GOTO statement.

```
trans = 2149
```

CHAPTER 13

Control Statements

Normally, statements in a program execute sequentially, in the order that you write them. Control statements specify changes to the sequential flow of statement execution. You can use a control statement to transfer the flow of execution to a point within the same program unit or to a point in a different program unit.

Some control statements change the flow of execution depending on a condition that is determined at that point in the flow of execution. Other control statements transfer the flow of execution every time that particular control statement executes, regardless of any condition.

There are sixteen control statements in Fortran-386. Note that many of these are variations of the GOTO and IF statements.

- Unconditional GOTO Statement
- Computed GOTO Statement
- Assigned GOTO Statement
- Arithmetic IF Statement
- Logical IF Statement
- Block IF Statement
- ELSE IF Statement
- ELSE Statement
- END IF Statement
- DO Statement
- CONTINUE Statement
- STOP Statement
- PAUSE Statement
- END Statement
- CALL Statement
- RETURN Statement

13.1. Arithmetic IF Statement

Use an arithmetic IF statement to transfer control to one of three executable statements that you specify. The executable statements must be in same program unit as the arithmetic IF statement. The value of an arithmetic expression determines which of the three statements receives control.

Syntax:

IF (arithmetic-expression) 1st-label, 2nd-label, 3rd-label

The value of the arithmetic-expression determines which of the three statements receives control. If the arithmetic-expression evaluates to a number less than 0, control transfers to the executable statement that the 1st-label identifies. If the arithmetic-expression evaluates to a number equal to 0, control transfers to the executable statement that the 2nd-label identifies. If the arithmetic-expression evaluates to a number greater than 0, control transfers to the executable statement that the 3rd-label identifies.

All three labels are required, but they do not have to identify three different statements.

Examples:

The following arithmetic IF statement transfers control to an executable statement with the statement label 7514 if the integer variable num is greater than to 0. If num is equal to zero, control transfers to statement 3500. Control transfers to statement 2000 if num is less than 0.

```
IF (num) 2000, 3500, 7514
```

The following arithmetic IF statement transfers control to statement 120 if the two variables in the expression have different values. Control transfers to statement 150 if the two variables have the same value.

```
IF (total-value) 120, 150, 120
```

13.2. Assigned GOTO Statement

Use an assigned GOTO statement to transfer control to an executable statement identified with a variable. The value assigned to the variable must represent the statement label of the executable statement that is to receive control. The executable statement that the variable identifies must be in the same program unit as the assigned GOTO statement. You assign a statement label to an integer variable using the ASSIGN statement. Refer to the "ASSIGN Statement" section for more information.

Syntax:

GOTO variable-name[[,] (statement-label list)]

The variable-name must have an integer data type and must have an assigned statement label value before the GOTO statement executes. If a program unit contains more than one ASSIGN statement for the variable-name in an assigned GOTO statement, the most recently executed ASSIGN statement determines the value of the variable. You can transfer control to different executable statements using multiple ASSIGN statements. An assigned GOTO statement and any related ASSIGN statements must be in the same program unit.

The optional statement-label list contains all the statement labels to which the assigned GOTO statement can transfer control. If you specify the list, your program can compare an assigned value with the values in the list. If the program does not find the assigned value in the list, the program can interrupt execution and issue an error message. If you specify the list, you must include all valid statement labels.

Examples:

The assigned GOTO statement in the following example transfers control to an executable statement with the statement label 810.

```
ASSIGN 810 TO inert
GOTO inert
```

The assigned GOTO statement in the next example transfers control to an executable statement with the statement label 464. Note the use of the statement-label list. The program could be designed to report an error if the program cannot find the assigned value in the list. The assigned value 464 is in the list. Therefore, no error would be detected.

```
ASSIGN 464 TO kalend
GOTO kalend (312, 1012, 464, 2000)
```

13.3. Block IF Statement

A block IF statement indicates the beginning of a block IF construct. A block in a block IF construct is a sequence of Fortran-386 statements designed to execute under specified conditions. The decision to execute a statement block in a block IF construct depends on a logical expression.

A block IF construct utilizes the ELSE, ELSE IF, and END IF statements in addition to the block IF statement to control the flow of execution.

Syntax:

```
IF (logical-expression) THEN
  .
  .
  statement-block
  .
  .
END IF
```

If the value of the logical-expression is true, control transfers to the statement-block. A statement-block can be empty. If the value of the logical-expression is false, control transfers to the first executable statement following the END IF.

Note that each block IF statement that you specify must have a corresponding END IF statement. The END IF statement identifies the end of a block IF construct.

Example:

The following example shows a block IF construct. The block IF statement that identifies the beginning of the construct contains an expression that evaluates to true. Therefore, the specified statement-block executes.

```
smallnum = 100/2
largenum = 100*2
IF (smallnum .LT. largenum) THEN
  pi = 3.14159
  radius = smallnum
  area = pi * radius ** 2
END IF
```

13.4. CALL Statement

Use the CALL statement to transfer execution control to a subroutine. The CALL statement specifies the symbolic name of the subroutine that you want to invoke and a list of actual arguments to pass to the subroutine.

Syntax:

CALL symbolic-name [(actual[, actual] ...)]

Actual arguments specified in the CALL statement must correspond in number, order, and data type to the dummy arguments specified in the subroutine. Actual arguments can be any one of the following items.

- an expression
- an array name
- an intrinsic function
- an external procedure name
- a dummy procedure name
- an alternate return specifier using the statement label of an executable statement in the same program unit as the CALL statement

In the following example, the CALL statement calls a subroutine named graph and passes three actual arguments.

Example:

```
CALL graph (vertical, horizontal, number)
```

13.5. Computed GOTO Statement

Use a computed GOTO statement to transfer execution control to one of a specified list of executable statements. The executable statements that you specify must be in same program unit as the computed GOTO statement. The value of an arithmetic expression determines which statement in the specified list receives control.

Syntax:

GOTO (statement-label list)[,] arithmetic-expression

The statement-label list contains one or more statement labels that identify executable statements. Separate the labels in the list with commas. You can refer to the statement-label list as the transfer- list.

A computed GOTO statement converts the value of the arithmetic-expression to an integer, if necessary. The value of the arithmetic-expression specifies a number that corresponds to the position of a particular statement label in the statement-label list.

If the value of the arithmetic-expression is less than 1 or greater than the number of statement labels in the statement-label list, execution control transfers to the first executable statement that follows the computed GOTO statement in the program.

Examples:

The following computed GOTO statement selects one of four statement labels for the transfer of execution control depending on the value of the variable num.

```
GOTO (54, 166, 418, 500), num
```

The next computed GOTO statement selects one of six statement labels for the transfer of execution control depending on the value of the three variables: velocity, t1, and t2.

```
GOTO (250, 500, 1000, 2000, 2500, 3000) velocity/(t1-t2)
```

13.6. CONTINUE Statement

The CONTINUE statement simply transfers execution control to the next executable statement in the program. Use the CONTINUE statement as the terminal statement for a DO loop that would otherwise end incorrectly with a control statement, such as an arithmetic IF, ELSE IF, or RETURN statement.

Syntax:

CONTINUE

The following example shows a DO loop that would end incorrectly with an arithmetic IF statement. The CONTINUE statement enables the DO loop to end properly.

Example:

```
                DO 211 var1 = 1, 5
                .
                .
                .
                IF (tax/2 + 4.25) 300, 400, 500
211             CONTINUE
```

13.7. DO Statement

Use the DO statement to specify a block of statements for loop processing. Loop processing is the repetitious execution of a statement block a specified number of times.

The DO statement indicates the beginning of a DO statement block. You must specify a terminal statement using a statement label in the DO statement to indicate the end of the block. All statements between the DO statement and the terminal statement, inclusive, define the range of a DO loop.

A variable and three expressions in the DO statement control the iteration count for a DO loop. The iteration count is the number of times a DO statement block is designed to execute or loop.

Syntax:

DO statement-label [,] variable = 1st-exp, 2nd-exp, [3rd-exp]

The statement-label identifies the terminal statement. The terminal statement cannot be an unconditional GOTO, assigned GOTO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement.

The variable can be an integer, real, or double precision variable. You can refer to this variable as the control variable or the DO variable. The value of the variable at any given time during execution of the DO loop depends on the values of the three expressions.

1st-exp, 2nd-exp, and 3rd-exp can be integer, real, or double precision expressions. 1st-exp is the initial value of the DO variable. 2nd-exp is the limit value of the DO variable. The optional 3rd-exp is an increment value that specifies how much the DO variable is incremented after each execution of the DO loop. If you omit 3rd-exp, an increment value of 1 is implied.

The DO statement evaluates the expressions first and converts the values of the expressions to the same data type as the DO variable, if necessary. The value of the 1st-exp is assigned to the DO variable. Then, the iteration count is calculated using the intrinsic functions MAX and INT, as shown below.

MAX(INT((2nd-exp − 1st-exp + 3rd-exp) / 3rd-exp), 0)

If the iteration count is greater than zero, the statements within the range of the DO loop execute. Following the first execution of the DO loop, the value of the DO variable is incremented according to the specified increment value. The iteration count is computed again and tested to see if it is greater than zero. If the iteration count is greater than zero, the statements within the range of the DO loop execute again. If the iteration count is negative or equal to zero, execution of the DO loop stops. Execution of the program continues with the first executable statement that follows the terminal statement of the DO loop.

You can nest DO loops; however, the range of a nested (inner) DO loop must lie completely within the range of the host (outer) DO loop. Nested DO loops can use the same terminal statement.

If you use a DO statement within the statement block of a block IF, ELSEIF, or ELSE statement, the range of the corresponding DO loop must lie completely within that statement block. If you use a block IF statement within the range of a DO loop, the

corresponding END IF statement must lie within the range of the DO loop.

Examples:

The following DO statement specifies 10 iterations of a DO loop. Note that an increment value is not specified. Therefore, an increment value of 1 is implied. The statement identified with the statement label 130 is the terminal statement for the loop.

```
DO 130 var = 1, 10
```

The next DO statement specifies 50 iterations of a DO loop. Note the use of a specified increment value.

```
DO 2100 var1 = 1, 100, 2
```

The next DO statement specifies 8 iterations of a DO loop. Note the use of the negative increment value.

```
DO 623 var2 = 16, 1, -2
```

The last DO statement demonstrates the use of more complicated expressions.

```
DO 599 var3 = value1*2+ 1, value2-var4/2, incr/2
```

13.8. END Statement

Use the END statement to indicate the end of a program unit. The END statement must be the last statement in every program unit.

Syntax:

END

When execution control reaches the END statement in a main program, program execution terminates. When execution control reaches the END statement in a subprogram, control returns to the main program. An END statement in a subprogram works like a RETURN statement.

13.9. END IF Statement

An END IF statement indicates the end of a block IF construct.

Syntax:

```
.
.
END IF
```

Each block IF statement that you use to specify a block IF construct must have a corresponding END IF statement to indicate the end of the construct.

Examples:

```
IF (smallnum .GT. largenum) THEN
                        STOP
ELSE IF (smallnum .LT. largenum) THEN
                        total = largenum - smallnum
ELSE IF (smallnum .EQ. largenum) THEN
                        ASSIGN 533 TO equality
                        GOTO equality (133, 333, 533)
END IF
```

13.10. ELSE Statement

An ELSE statement specifies an additional statement block for conditional execution within a block IF construct. Unlike block IF and ELSE IF statements, an ELSE statement does not contain a logical expression to determine whether or not control transfers to the specified statement-block. A statement-block that corresponds to an ELSE statement executes only if no preceding statement-block in the block IF construct executes.

Syntax:

```
.
.
ELSE
  .
  .
  statement-block
  .
  .
```

An ELSE statement and its corresponding statement-block must follow a block IF statement and its corresponding statement-block or an ELSE IF statement and its corresponding statement-block. No other ELSE, or ELSE IF statements can follow an ELSE statement in a block IF construct. An END IF statement can follow an ELSE statement.

In the following example, the statement-block that corresponds to the ELSE statement in the block IF construct executes because the logical expression in the opening block IF statement evaluates to false.

Example:

```
largenum = 712
smallnum = 4
IF (largenum .LT. smallnum) THEN
                        factor = smallnum/2 – 2.612
                        unit5 = val1 + val2 * factor
ELSE
                        ASSIGN 2001 TO standard
                        GOTO standard (2001)
END IF
```

13.11. ELSE IF Statement

An ELSE IF statement specifies an additional statement block for conditional execution within a block IF construct. Like a block IF statement that indicates the beginning of a block IF construct, an ELSE IF statement contains a logical expression to determine whether or not control transfers to the specified statement-block. A block IF construct can contain any number of ELSE IF statements.

An ELSE IF statement and its corresponding statement-block must follow a block IF statement and its corresponding statement-block.

Syntax:

```
.
.
ELSE IF (logical-expression) THEN
.
.
statement-block
.
.
```

If the value of the logical-expression is true, control transfers to the statement-block. A statement-block can be empty. If the value of the logical-expression is false, control transfers to the next ELSE IF or ELSE statement, or the first executable statement following the END IF statement in that construct.

In the following example, there are two ELSE IF statements in the block IF construct. The logical-expression in the opening block IF statement is false. Therefore, control passes to the first ELSE IF statement. The logical-expression in the first ELSE IF statement is also false. Therefore, control passes to the second ELSE IF statement. The logical-expression in the second ELSE IF statement is true. Therefore, the statement block specified after the second ELSE IF statement executes.

Example:

```
smallnum = 56000
largenum = 56000
IF (smallnum .GT. largenum) THEN
                        STOP
ELSE IF (smallnum .LT. largenum) THEN
                        total = largenum – smallnum
ELSE IF (smallnum .EQ. largenum) THEN
                        ASSIGN 533 TO equality
                        GOTO equality (133, 333, 533)
END IF
```

13.12. Logical IF Statement

A logical IF statement conditionally executes a single Fortran-386 statement that you specify literally within the logical IF statement. The decision to execute the statement is based on the value of a logical expression.

Syntax:

IF (logical-expression) statement

Note that the statement is a complete, literal statement specification and not a statement label. If the value of the logical-expression is true, control transfers to the statement. If the value of the logical-expression is false, control transfers to the first executable statement following the logical IF statement in the program.

Examples:

The value of the logical-expression in the following logical IF statement is true. Therefore, the STOP statement specified in the logical IF statement executes.

```
largenum = 427
smallnum = 8.602
IF (largenum .GT. smallnum) STOP
```

The value of the logical-expression in the next logical IF statement is false. Therefore, the RETURN statement specified in the logical IF statement does not execute.

```
largenum = 427
smallnum = 8.602
IF (smallnum .GT. largenum) RETURN
```

13.13. PAUSE Statement

Use the PAUSE statement to temporarily suspend program execution. You can specify an optional message that displays on the console screen just before the program pauses.

Syntax:

PAUSE [display-message]

The optional display-message can be a character constant or a digit string of five digits or less. The message can serve as a prompt for user input from the console. If you specify a display-message, the PAUSE statement displays the message on the console screen, suspends program execution, and waits for user response from the console.

In the following example, the first PAUSE statement simply suspends program execution and waits for a response from the console. The second PAUSE statement displays the message TYPE ANY KEY TO CONTINUE before suspending execution.

Examples:

```
PAUSE
PAUSE 'TYPE ANY KEY TO CONTINUE'
```

13.14. RETURN Statement

Use a RETURN statement to terminate execution of a procedure and to transfer execution control back to the program unit that references the procedure. Procedures can contain any number of individual routines with multiple entry and return points. The RETURN statement serves as an intermediate termination point in a procedure that contains more than one individual routine. A procedure can contain any number of RETURN statements.

There are two forms of the RETURN statement. The first form is for use in a function procedure. The second form is for use in a subroutine procedure.

Syntax:

RETURN
RETURN [integer-expression]

Execution of a RETURN statement in a function procedure returns execution control to the program unit that referenced the function. The value of the function must be defined before the RETURN statement executes.

Execution of a RETURN statement in a subroutine procedure returns execution control to the program unit that called the subroutine. Use the optional integer expression to select an alternate return specifier. The integer expression indicates which alternate return asterisk in the SUBROUTINE or ENTRY statement dummy argument list to use for the return. A valid integer expression must be greater than or equal to 1 and less than or equal to the number of asterisks specified in the SUBROUTINE or ENTRY statement dummy argument list. Each asterisk in the dummy argument list corresponds to an actual argument supplied in the CALL statement that invokes the subroutine. Actual arguments are statement numbers that indicate the alternate return points. Refer to the "Alternate Return Specifiers" section for more information.

Execution of a RETURN statement in a procedure causes all program entities in that procedure to become undefined except for the following:

- entities in a blank (unnamed) common block
- entities that are initially defined but do not become redefined or undefined in the procedure
- entities in a named common block that appear in the procedure and in at least one other program unit that references the procedure

You can use the SAVE statement to retain the definition status of any program entity following the execution of a RETURN or END statement.

The RETURN statement is not required to terminate a procedure. The END statement declares the physical end of a procedure subprogram. An END statement used in a procedure has the same effect as a RETURN statement.

13.15. STOP Statement

Use the STOP statement to terminate program execution.

Syntax:

STOP [display-message]

The optional display-message can be a character constant or a digit string of five digits or less. If you specify a display-message, the STOP statement displays the message on the console screen, terminates program execution, and returns control to the operating system.

The first STOP statement in the example below terminates program execution and returns control to the operating system. The second STOP statement example displays the message END OF PROGRAM before terminating execution.

Example:

```
STOP
STOP 'END OF PROGRAM'
```

13.16. Unconditional GOTO Statement

Use an unconditional GOTO statement to transfer execution control to an executable statement. The executable statement must be in the same program unit as the unconditional GOTO statement. An unconditional GOTO statement transfers control to the same statement each time it executes.

Syntax:

GOTO statement-label

The unconditional GOTO statement transfers control to the statement identified by the statement-label. The statement-label must identify an executable statement that is in the same program unit as the GOTO statement.

In the example below, the unconditional GOTO statement transfers control to an executable statement with the statement label 2189. Every time the statement executes, it sends control to the same executable statement.

Example:

```
GOTO 2189
```

CHAPTER 14

Input/Output Statements

Fortran-386 I/O statements transfer data from one storage location to another within a processor. They also transfer data between a processor and any external device such as a console, printer, or a storage medium like a disk, floppy disk, or magnetic tape.

Fortran-386 provides three categories of I/O statements:

- data transfer statements
- file positioning statements
- auxiliary I/O statements

Data transfer statements move data between internal (processor) storage and some external medium. The data transfer statements are

- READ
- WRITE
- PRINT
- ENCODE
- DECODE

File positioning statements manipulate the position of the internal file pointer relative to a specific file. The file positioning statements are

- BACKSPACE
- REWIND
- ENDFILE

Auxiliary I/O statements manipulate the connection of I/O units to external media, and inquire about the characteristics of a particular connection. The auxiliary I/O statements are

- OPEN
- CLOSE
- INQUIRE

This chapter describes the syntax of each I/O statement in detail. The "Fortran-386 Input/Ouput System" chapter describes records, files, I/O units, and the I/O system in general. For reference in this section, the statement descriptions are listed alphabetically.

14.1. BACKSPACE Statement

The BACKSPACE statement moves the file pointer to the beginning of the preceding record.

Syntax:

BACKSPACE Unit-number
BACKSPACE(argument-list)

In the first form, Unit-number is an integer expression whose value is in the range 0 to 99. In the second form, argument-list is a list of specifiers that control the positioning process.

With either form, if the file has no preceding record, the I/O system ignores the BACKSPACE statement. If the preceding record is an endfile record, the BACKSPACE statement moves the file pointer to the beginning of the endfile record.

The Argument-List

The argument-list of the BACKSPACE statement consists of a comma separated list of one of more of the I/O specifiers listed below.

Unit Specifier

[UNIT =] Unit-number

Unit-number is an integer in the range 0 to 99 that specifies an external I/O unit. The keyword UNIT = is optional.

I/O-status Specifier

IOSTAT = Io-status

Io-status is an integer variable or integer array element. The I/O system assigns Io-status the following values based on the outcome of the positioning:

Io-status = 0 if no error has occurred
Io-status $\geq$ 1 if an error has occurred

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the BACKSPACE statement. If an error occurs while processing the BACKSPACE statement, the following actions occur:

1. The BACKSPACE operation is terminated.

2. The file position becomes undefined, and the only valid statements that you can execute are CLOSE, REWIND, INQUIRE, or BACKSPACE.

3. The I/O system sets the Io-status (if one is specified).

4. The flow of control resumes at the statement labeled with Error-label. If

there is no ERR specifier, a run-time error occurs.

Restrictions

- The BACKSPACE statement cannot reference an internal file.
- The file must be connected for sequential access.
- In the form BACKSPACE(argument-list), you must include an I/O unit specifier.
- You cannot use the BACKSPACE statement with a record that was written using list-directed formatting.

Examples

```
BACKSPACE 2
BACKSPACE(2,IOSTAT=errorflag,ERR=999)
```

14.2. CLOSE Statement

The CLOSE statement disconnects a file from an I/O unit. After the I/O unit is disconnected, a subsequent OPEN statement can connect it to the same file or a different file in the same program. Also, if the CLOSE statement disconnects a file, a subsequent OPEN statement can connect it to the same I/O unit or a different I/O unit, if the file still exists.

Syntax:

CLOSE(close-list)

where close-list is a comma separated list of one or more specifiers that control the close operation.

The Close-List

Unit Specifier

[UNIT =] Unit-number

Unit-number is an integer value between 0 and 99 that specifies an external I/O unit. The keyword UNIT = is optional.

I/O-status Specifier

IOSTAT = Io-status

Io-status is an integer or an element of an integer array. The I/O system assigns Io-status one of the following values based on the outcome of the close process:

Io-status = 0 if no error has occurred
Io-status $\geq$ 1 if an error has occurred
Io-status = -1 if an end-of-file condition has occurred

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program I/O unit as the CLOSE statement. If the I/O system encounters an error while processing the CLOSE statement, the following actions occur:

1. The CLOSE operation is terminated.

2. The file position becomes undefined, unless the error is an end-of-file condition. Then, the file pointer points just past the endfile record, and the only valid statements that you can execute are BACKSPACE, REWIND, or INQUIRE.

3. The I/O system sets the Io-status (if one is specified).

4. The flow of control resumes at the statement labeled with Error-label.

If there is no ERR specifier, a run-time error occurs.

Status Specifier

STATUS = Status

Status is a character string expression whose value must be one of the following:

'KEEP' or 'DELETE'

If you omit STATUS, the I/O system supplies the default STATUS = 'KEEP' unless the file already has a status of SCRATCH, in which case the default is STATUS = 'DELETE'.

Restrictions

- The CLOSE statement need not occur in the same program I/O unit as its corresponding OPEN statement.
- If the specified file does not exist, the I/O system ignores the CLOSE statement.
- If STATUS = 'KEEP', the file continues to exist after the I/O system executes the CLOSE statement.
- If STATUS = 'DELETE', the file does not exist after the I/O system executes the CLOSE statement.
- When a program terminates normally, the I/O system automatically closes all connected I/O units, and deletes all files with STATUS = 'SCRATCH'.

Examples

```
CLOSE(3)
CLOSE(3,IOSTAT=errorflag,ERR=999,STATUS='KEEP')
```

14.3. DECODE Statement

The DECODE statement transfers data from external character form to internal binary representation, using a format specification. DECODE is functionally equivalent to using a READ statement with formatted records on an internal file connected for sequential access.

Syntax:

DECODE(control-list)[transfer-list]

where control-list is a comma separated list of specifiers that control the transfer process, and transfer-list is the list of variables that receive the data after translation to internal binary representation.

The Control-list

Characters Specifier

Num-chars

Num-chars is an integer expression that designates the number of characters to translate to internal binary representation. Num-chars must be the first specifier in the control-list.

Format Specifier

Format

Format is a format identifier that controls the editing of the data during the transfer. Refer to "The FORMAT Statement and Format Specifications" chapter for more information. Format must be the second specifier in the control-list.

Location Specifier

Location

Location is the name of a variable, an array, or an array element that contains the characters to translate to internal binary representation. Location must be the third specifier in the control-list.

I/O-status Specifier

IOSTAT = Io-status

Io-status is an integer variable or an element of an integer array. The I/O system assigns Io-status one of the following values based on the outcome of the data transfer:

Io-status = 0 if no error has occurred
Io-status $\geq$ 1 if an error has occurred
Io-status = -1 if an end-of-string condition has occurred

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the DECODE statement. If the I/O system encounters an error while processing the DECODE statement, the following actions occur:

1. The DECODE operation is terminated.

2. The I/O system sets the Io-status (if one is specified).

3. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

Restrictions

- If Location is an array, DECODE processes the elements in normal column-major order.
- The process of format control is the same as for a formatted READ statement.
- The number of characters that DECODE can process depends on the data type of location. A character variable or a character array element can contain the same number of characters as its declared length, but a character array can contain a number of characters equal to the length of each element times the number of elements. For example, each element of an INTEGER*4 array can contain four characters, so the total number of characters DECODE can process is four times the number of array elements.

Examples

```
DECODE (3,'(Z)',100) var1, var2
DECODE (ichar, '(3I4)', block1) ivar1, ivar2, ivar3
```

14.4. ENCODE Statement

The ENCODE statement transfers data from internal binary representation to external character form, using a format specification. ENCODE is functionally equivalent to using a WRITE statement with formatted records on an internal file connected for sequential access.

Syntax:

ENCODE(control-list)[transfer-list]

where control-list is a comma separated list of specifiers that control the transfer process as described below, and transfer-list is the list of variables to translate to an internal binary representation.

The Control-list

Characters Specifier

Num-chars

Num-chars is an integer expression that designates the number of characters to translate to external form. Num-chars must be the first specifier in the control-list.

Format Specifier

Format

Format is a format identifier that controls the editing of the data during the transfer. Refer to the "Format Statement and Format Specifications" chapter for more information. Format must be the second specifier in the control-list.

Location Specifier

Location

Location is the name of a variable, an array, or an array element that contains the characters after translation to external form. Location must be the third specifier in the control-list.

I/O-status Specifier

IOSTAT = Io-status

Io-status is an integer variable or an element of an integer array. The I/O system assigns Io-status one of the following values based on the outcome of the data transfer:

Io-status = 0 if no error has occurred
Io-status $\geq$ 1 if an error has occurred

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the ENCODE statement. If the I/O system encounters an error while processing the ENCODE statement, the following actions occur:

1. The ENCODE operation is terminated.

2. The I/O system sets the Io-status (if one is specified).

3. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

Restrictions

- If Location is an array, ENCODE processes the elements in normal column-major order.
- The process of format control is the same as for a formatted WRITE statement.
- The number of characters that ENCODE can process depends on the data type of location. A character variable or a character array element can contain the same number of characters as its declared length, but a character array can contain a number of characters equal to the length of each element times the number of elements. For example, each element of an INTEGER*4 array can contain four characters, so the total number of characters ENCODE can process is four times the number of array elements.

Examples

```
ENCODE (3,'(Z)',arr1) var1
ENCODE (ichar,'(2I5,F7.2)',arr2) ivar1, ivar2, var3
```

14.5. ENDFILE Statement

The ENDFILE statement causes the I/O system to write an endfile record as the next record of the file.

Syntax:

ENDFILE Unit-number
ENDFILE(argument-list)

In the first form, Unit-number is an integer expression whose value is in the range 0 to 99. In the second form, argument-list is a comma separated list of specifiers that control the positioning process.

After executing the ENDFILE statement, the I/O system cannot execute any further data transfer statements until it executes a BACKSPACE or REWIND statement.

The Argument-list

Unit Specifier

[UNIT =] Unit-number

Unit-number is an integer in the range 0 to 99 and specifies an external I/O unit. The keyword UNIT = is optional.

I/O-status Specifier

IOSTAT = Io-status

Io-status is an integer variable or integer array element. The I/O system assigns Io-status the following values based on the outcome of the positioning:

Io-status = 0 if no error has occurred
Io-status $\geq$ 1 if an error has occurred

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the ENDFILE statement. If an error occurs while processing the ENDFILE statement, the following actions occur:

1. The ENDFILE operation is terminated.

2. The file position becomes undefined, and the only valid statements that you can execute are CLOSE, REWIND, BACKSPACE, or INQUIRE.

3. The I/O system sets the Io-status (if one is specified).

4. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

Restrictions

- The ENDFILE statement cannot reference an internal file.
- In the form, ENDFILE(argument-list), you must include an I/O unit specifier.
- The file must be connected for sequential access.

Examples

```
ENDFILE 4
ENDFILE(4,IOSTAT=errorflag,ERR=999)
```

14.6. INQUIRE Statement

The INQUIRE statement returns information about the characteristics of a named file, or the connection of a file to a particular I/O unit.

Syntax:

INQUIRE(File-name,inquire-list)
INQUIRE(Unit-number,inquire-list)

The first form is called inquire-by-file. File-name is the name of the file being inquired about. File-name need not exist or be connected.

The second form is called inquire-by-unit. The Unit-number is an integer value between 0 and 99 that specifies an external I/O unit.

In both forms, the inquire-list is a comma separated list of specifiers that describe the various characteristics of a I/O unit/file connection. When executing either form, the I/O system assigns values to these specifiers as described in the "Assignment Statements" chapter.

The Inquire-list

Access Method Specifier

ACCESS = Access-method

Access-method is a character variable or an element of a character array that the I/O system assigns the character value 'SEQUENTIAL' if the file being inquired about is connected for sequential access, or the character value

If the file is not connected, Access-method remains unchanged.

Blank Specifier

BLANK = Blank-type

Blank-type is a character variable or an element of a character array that the I/O system assigns the character value 'NULL' if the file is connected for formatted I/O and is using null blank control, or the character value 'ZERO' if the file is connected for formatted I/O and is using zero blank control. Refer to the "OPEN Statement" section for more information.

If the file is not connected, or it is not connected for formatted I/O, Blank-type remains unchanged.

Current Record-Length Specifier

RECL = Current-record-length

Current-record-length is an integer variable or an element of an integer array that the I/O system assigns the current value for the record length in the file being inquired about.

If the file is connected for formatted I/O, Current-record-length is the number of characters.

If the file is connected for unformatted I/O, Current-record-length is measured in bytes.

If the file is not connected, or the file is not connected for direct access, Current-record-length remains unchanged.

Current Name Specifier

NAME = Current-file-name

Current-file-name is a character variable or an element of a character array that the I/O system assigns the current name of the file being inquired about.

If the file has no name, Current-file-name remains unchanged.

Current Unit Specifier

NUMBER = Current-unit-number

Current-unit-number is an integer or an element of an integer array to which the I/O system assigns the number of the I/O unit currently connected to File-name. If no I/O unit is connected to File-name, Current-unit-number remains unchanged.

Direct Specifier

DIRECT = Direct-allowed

Direct-allowed is a character variable or an element of a character array that the I/O system assigns the character value 'YES' if direct access is allowed for the file being inquired about, or the character value 'NO' if direct access is not allowed for the file.

If the I/O system cannot determine whether direct access is allowed for the file, it assigns the value UNKNOWN.

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the INQUIRE statement. If the I/O system encounters an error while processing the INQUIRE statement, the following actions occur:

1. The INQUIRE operation is terminated.

2. The file position becomes undefined, unless the error is an end-of-file condition. Then, the file pointer points just past the endfile record, and the only valid statements that you can execute are CLOSE, BACKSPACE, or REWIND.

3. The I/O system sets the Io-status (if one is specified); all other specifiers become undefined.

4. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

Existence Specifier

EXIST = Exist-status

Exist-status is a logical variable or an element of a logical array.

When executing an inquire-by-file statement, the I/O system assigns Exist-status the logical value .TRUE. if File-name exists; otherwise, it assigns the logical value .FALSE..

When executing an inquire-by-unit statement, the I/O system assigns Exist-status the logical value .TRUE. if Unit-number exists; otherwise, it assigns the logical value .FALSE..

Formatted Specifier

FORMATTED = Formatted-allowed

Formatted-allowed is a character variable or an element of a character array that the I/O system assigns the character value 'YES' if the file being inquired about can contain formatted records, or the character value 'NO' if the file cannot contain formatted records.

If the I/O system cannot determine whether the file can contain formatted records, it assigns the character value 'UNKNOWN'.

I/O-status Specifier

IOSTAT = Io-status

Io-status is an integer or an element of an integer array. The I/O system assigns one of the following values based on the outcome of the inquiry.

Io-status = 0 if no error has occurred
Io-status $\geq$ 1 if an error has occurred
Io-status = -1 if an end-of-file condition has occurred

I/O Type Specifier

FORM = Io-type

Io-type is a character variable or an element of a character array that the I/O system assigns the character value 'FORMATTED' if the file being inquired about is connected for formatted I/O, or 'UNFORMATTED' if the file is connected for unformatted I/O.

If the file being inquired about is not connected, Io-type remains unchanged.

Named Specifier

NAMED = Named-status

Named-status is a logical variable or an element of a logical array. When the I/O system executes an inquire-by-unit statement, it assigns Named-status the logical value .TRUE. if the file connected to the Unit-number has a name; otherwise, it assigns the value .FALSE..

Next Record Specifier

NEXTREC = Next-record-number

Next-record-number is an integer variable or an element of an integer array that the I/O system assigns the value n+1, where n is the number of the last record the I/O system has read from or written to in the file being inquired about.

If the file is connected, but the I/O system has not yet read or written any records, Next-record-number is assigned the value 1.

If the file is not connected for direct access, or the file position is undefined because of a previous error, Next-record-number remains unchanged.

Open Specifier

OPENED = Open-status

Open-status is a logical variable or an element of a logical array.

When executing an inquire-by-file statement, the I/O system assigns Open-status the logical value .TRUE. if File-name is connected to a I/O unit; otherwise, it assigns the value .FALSE..

When executing an inquire-by-unit statement, the I/O system assigns Open-status the logical value .TRUE. if Unit-number is connected to a file; otherwise, it assigns the value .FALSE..

Sequential Specifier

SEQUENTIAL = Sequential-allowed

Sequential-allowed is a character variable or an element of a character array that the I/O system assigns the character value 'YES' if sequential access is allowed for the file, or the character value 'NO' if the sequential access is not allowed for the file.

If the I/O system cannot determine whether sequential access is allowed for the file, it assigns the character value 'UNKNOWN'.

Unformatted Specifier

UNFORMATTED = Unformatted-allowed

Unformatted-allowed is a character variable or an element of a character array that the I/O system assigns the character value 'YES' if the file being inquired about can contain unformatted records, or the character value 'NO' if the file cannot contain unformatted records.

If the I/O system cannot determine whether the file can contain unformatted records, it assigns the character value 'UNKNOWN'.

Restrictions

- The I/O system can execute the INQUIRE statement before, while, or after a file is connected to a I/O unit.
- All the values assigned to the inquire-list specifiers are those that are current when the I/O system executes the INQUIRE statement.
- Any variable or array element that becomes defined or undefined by being used as a specifier in an INQUIRE statement cannot be referenced by another

specifier in the same INQUIRE statement.

- When the I/O system executes an inquire-by-file statement, the specifiers:
  - Named-status
  - Current-file-name
  - Sequential-allowed
  - Direct-allowed
  - Formatted-allowed
  - Unformatted-allowed

  are assigned values only if File-name is a valid filename for the implementation and exists; otherwise, they all remain unchanged.

  Current-unit-number is assigned a value if and only if Open-status is TRUE.

  Also, if Open-status is assigned TRUE, then the specifiers:
  - Access-method
  - Io-type
  - Current-record-length
  - Next-record-number
  - Blank-type

  can also become defined.
- When the I/O system executes an inquire-by-unit statement, the specifiers
  - Current-unit-number
  - Named-status
  - Current-file-name
  - Access-method
  - Sequential-allowed
  - Direct-allowed
  - Io-type
  - Formatted-allowed
  - Unformatted-allowed
  - Current-record-length
  - Next-record-number
  - Blank-type

  are assigned values only if Unit-number exists and is connected to a file; otherwise they all remain unchanged.
- When the I/O system executes the INQUIRE statement, the specifiers Exist-status and Open-status are always assigned a value unless an error condition occurs.

Examples

```
INQUIRE(3)
INQUIRE(3,EXIST=inqvar1,OPENED=inqvar2,DIRECT=inqvar3,RECL=inqvar4)
INQUIRE('temps.dat',NUMBER=inqfvar1,BLANK=inqfvar2)
```

14.7. OPEN Statement

The OPEN statement can

- create a file and connect it to a I/O unit.
- create a preconnected file.
- connect an existing file to a I/O unit.
- change the characteristics of an existing I/O unit/file connection.

Syntax:

OPEN(open-list)

where open-list is a comma separated list of specifiers that control the opening process.

The Open-list

Access Method Specifier

ACCESS = Access-method

Access-method is a character string expression whose value must be either:

'SEQUENTIAL' or 'DIRECT'

If you omit ACCESS, the I/O system supplies the default ACCESS = 'SEQUENTIAL'.

If you specify ACCESS = 'DIRECT', you must also specify the record length (see below).

If the file already exists, Access-method must match the file's characteristics. If the file does not already exist, the OPEN statement creates it with the given Access-method.

Blank Specifier

BLANK = Blank-type

Blank-type is a character string expression whose value is either:

'NULL' or 'ZERO'

If you omit BLANK, the I/O system supplies the default BLANK = 'NULL'.

If you specify BLANK = 'NULL', the I/O system ignores any blank (20h) characters in numeric, formatted input fields, with the exception that a field of all blanks has the value zero (30h).

If you specify BLANK = 'ZERO', the I/O system treats all blanks, except leading blanks as zeros.

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the OPEN statement. If the I/O system encounters an error while processing the OPEN statement, the following actions occur:

1. The OPEN operation is terminated.

2. The file position becomes undefined, and the only valid statements that you can execute are CLOSE, BACKSPACE, REWIND, or INQUIRE.

3. The I/O system sets the Io-status (if one is specified).

4. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

File Specifier

FILE = File-name

File-name is the name of the file to be connected to the specified I/O unit. File-name must be the name of a file in the operating system's file structure.

If you omit the FILE specifier and the I/O unit is not preconnected, the I/O system connects the I/O unit to a file named fort.n, where n is the I/O unit number.

I/O-status Specifier

IOSTAT = Io-status

Io-status is an integer or an element of an integer array. The I/O system assigns Io-status one of the following values based on the outcome of the open process:

Io-status = 0 if no error has occurred
Io-status $\geq$ 1 if an error has occurred

Record Length Specifier

RECL = Record-length

Record-length is a positive integer expression whose value is the length of each record in the file. The file must be connected for direct access.

If the records are formatted, Record-length is the number of characters. If the records are unformatted, Record-length is the number of bytes.

Record Type Specifier

FORM = Record-type

Record-type is a character string expression whose value must be either:

'FORMATTED' or 'UNFORMATTED'

If you omit FORM, the I/O system supplies the default FORM = 'UNFORMATTED' when you connect the file for direct access, or FORM = 'FORMATTED' when you connect the file for sequential access.

Status Specifier

STATUS = Status

Status is a character string expression whose value must be one of the following:

'NEW' or 'OLD' or 'SCRATCH' or 'UNKNOWN'

If you omit STATUS, the I/O system supplies the default STATUS = 'UNKNOWN'. NEW and OLD are only valid if you also use a FILE specifier, or the file is preconnected.

If you specify STATUS = 'SCRATCH', the I/O system connects the specified I/O unit to a temporary file. The connection lasts until you execute a CLOSE statement (see below), or the program terminates.

If you specify STATUS = 'UNKNOWN' and the file exists, the I/O system connects the file. If you specify STATUS = 'UNKNOWN' and the file does not exist, the I/O system first creates the file and then connects

it.

Unit Specifier

[UNIT =] Unit-number

Unit-number is an integer value between 0 and 99 that specifies an external I/O unit. The keyword UNIT = is optional.

Restrictions

- The open-list must contain a Unit-number; all other specifiers are optional, but there can be only one of each.
- If you omit UNIT = , then the Unit-number must be the first specifier in the open-list.
- If the file is connected for direct access, the open-list must contain a Record-length.
- If the file is connected for sequential access, the open-list cannot contain a Record-length.
- If STATUS = 'SCRATCH', the open-list cannot contain a File-name.
- If the file is already connected to a I/O unit, the only specifier that can be changed with the OPEN statement is Blank-type. The file position is not affected.
- The specifier BLANK = is valid only with formatted records.

Examples

```
OPEN(3)
OPEN(UNIT=3,STATUS='SCRATCH')
OPEN(8,FILE='overdues.dat',IOSTAT=errorflag,ERR=999,ACCESS='DIRECT')
```

14.8. PRINT Statement

The PRINT statement transfers formatted data to the default output I/O unit. The PRINT statement is functionally equivalent to using a WRITE statement with formatted records.

Syntax:

PRINT format[,output-list]

where format is a format specification and output-list is the list of data items to be written.

When executing the PRINT statement, the I/O system does not print the first character in a formatted record. Instead, the I/O system uses this character to determine the vertical spacing to use before printing the remainder of the record. The PRINT statement then prints any remaining characters in the record on one line beginning at the lefthand margin.

The table below shows the amount vertical spacing determined by the first character.

Vertical Spacing in the PRINT Statement

Character	Hex Value	Vertical Space Output Before Printing Formatted Record
Blank	20h	Advance one line
0	30h	Advance two lines
1	31h	Advance to first line of next page
+	2Bh	No advance

The Output-list

The output-list can contain any of the following:

- a variable name
- an array name
- an array element name
- an expression containing any of the types CHARACTER, COMPLEX, DOUBLE PRECISION, INTEGER, LOGICAL, or REAL

If you use an array name in the output-list, the I/O system writes the entire array in the normal column-major order.

Implied-DO List

The PRINT statement can write a range of subscripted array elements from the output-list using an implied-DO list of the form:

(input-list,var = e1,e2[,e3])

where var and e1,e2, and e3 are the same as described in the explanation of the DO statement in the "Control Statements" chapter. The output-list can contain other (nested) implied-DO lists.

Restrictions

- The data must be formatted.

Examples

```
PRINT 35, name,score
PRINT *, employee
```

14.9. READ Statement

The READ statement transfers data from a specific I/O unit to internal (processor) storage. You can use the READ statement to reference both internal and external files. The READ statement can transfer both formatted and unformatted records.

Syntax:

READ(control-list)[input-list]
READ Format[,input-list]

In the first form, the control-list is a list of comma separated specifiers that control the transfer process. In the second form, format is a format specification. Refer to the "Format Statement and Format Specifications" chapter for more information on format specification. In both forms, input-list is the list of variables that receive the input data.

The Control-list

Unit Specifier

[UNIT =] Unit-id

Unit-id can be an unsigned integer in the range 0 to 99 that identifies an external I/O unit, an asterisk (*) specifying the default input I/O unit, or an internal file. The keyword UNIT = is optional.

Format Specifier

[FMT =] Format

Format is a format identifier that controls the editing of the data during the transfer (see "The Format Statement and Format Specification" chapter). The keyword FMT = is optional.

Record Specifier

REC = Record-number

Record-number is a nonzero integer expression that specifies the number of the record to read.

I/O-status Specifier

IOSTAT = Io-status

Io-status is an integer variable or an element of an integer array. The I/O system assigns Io-status one of the following values based on the outcome of the data transfer:

Io-status = 0 if no error has occurred
Io-status $\geq$ 1 if an error has occurred
Io-status = -1 if an end-of-file condition has occurred

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the READ statement. If the I/O system encounters an error while processing the READ statement, the following actions occur:

1. The READ operation is terminated.

2. The file position becomes undefined, unless the error is an end-of-file condition. Then, the file pointer points just past the endfile record, and the only valid statements that you can execute are CLOSE, BACKSPACE, REWIND, or INQUIRE.

3. The I/O system sets the Io-status (if one is specified).

4. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

End Specifier

END = End-label

End-label is the label of an executable statement in the same program unit as the I/O statement. When the I/O system encounters an end-of-file condition while processing the READ statement, the following actions occur:

1. The READ operation is terminated.

2. The file pointer points just past the endfile record, and the only valid statements that you can execute are CLOSE, BACKSPACE, REWIND or INQUIRE.

3. The I/O system sets the Io-status = -1 (if one is specified).

4. The flow of control resumes at the statement labeled with End-label. If there is no END specifier, a run-time error occurs.

The Input-list

The input-list can contain any of the following:

- a variable name
- an array name
- an array element name

If you use an array name in the input-list, the I/O system reads the entire array in the normal column-major order.

Implied-DO List

The READ statement can read a range of subscripted array elements from the input-list using an implied-DO list of the form:

(input-list,var = e1,e2[,e3])

where var and e1, e2, and e3 are the same as described in the explanation of the DO statement in "Control Statements" chapter. The input-list can contain other (nested) implied-DO lists.

Restrictions

- The control-list must contain an I/O unit specifier, but can contain only one of each of the other specifiers.
- If you do not use UNIT = , then the Unit-id must be the first specifier in the control-list.
- If you do not use FMT = , then you cannot use UNIT = and the Format must be the second specifier in the control-list.
- In the form READ Format[,input-list], the I/O unit is the default input unit.
- If Unit-id designates an internal file, the control-list must contain a format specification other than an asterisk (*), but cannot contain a record number.
- If the Format is an asterisk (*), specifying list-directed formatting, the control-list cannot contain a record number specifier.
- If the file is connected for direct access, the control-list must contain a record number.
- The END specifier is only valid if the file is connected for sequential access.

Examples

```
READ(10,200) name,score
READ(10,200,IOSTAT=errorflag,ERR=350,END=999) name,score
READ(3,REC=105,IOSTAT=errorflag,ERR=200,END=999) altitude
READ(3,REC=(2j + k),IOSTAT=errorflag,ERR=200,END=999) altitude
READ(5,*) employee
```

14.10. REWIND Statement

The REWIND statement moves the file pointer to the initial point of the file.

Syntax:

REWIND Unit-number
REWIND(argument-list)

In the first form, Unit-number is an integer expression whose value is in the range 0 to 99. In the second form, argument-list is a comma separated list of specifiers that control the positioning process.

With both forms, if the file pointer is already positioned at the initial point, the I/O system ignores the REWIND statement.

The Argument-list

Unit Specifier

[UNIT =] Unit-number

Unit-number is an integer in the range 0 to 99 that specifies an external I/O unit. The keyword UNIT = is optional.

I/O-status Specifier

IOSTAT = Io-status

Io-status is an integer variable or integer array element. The I/O system assigns Io-status the following values based on the outcome of the positioning:

Io-status = 0 if no error has occurred
Io-status $\geq$ 1 if an error has occurred

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the REWIND statement. If an error occurs while processing the REWIND statement, the following actions occur:

1. The REWIND operation is terminated.

2. The file position becomes undefined, and the only valid statements that you can execute are CLOSE, BACKSPACE, REWIND, or INQUIRE.

3. The I/O system sets the Io-status (if one is specified).

4. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

Restrictions

- The REWIND statement cannot reference an internal file.
- In the form REWIND(argument-list), you must include an I/O unit specifier.
- The file must be connected for sequential access.

Examples

```
REWIND 4
REWIND(4,IOSTAT=errorflag,ERR=999)
```

14.11. WRITE Statement

The WRITE statement transfers data from internal (processor) storage to a specific I/O unit. You can use the WRITE statement to reference both internal and external files. The WRITE statement can transfer both formatted and unformatted records.

Syntax:

WRITE(control-list)[output-list]

where control-list is a comma separated list of specifiers that control the transfer process, and output-list is the list of variables to be written.

The Control-list

Unit Specifier

[UNIT =] Unit-id

Unit-id can be an unsigned integer in the range 0 to 99 that designates an external I/O unit, an asterisk (*) specifying the default output I/O unit, or an internal file. The keyword UNIT = is optional.

Format Specifier

[FMT =] Format

Format is a format identifier that controls the editing of the data during the transfer. Refer to the "Format Statement and Format Specifications" chapter for more information. The keyword FMT = is optional.

Record Specifier

REC = Record-number

Record-number is a nonzero integer expression that specifies the number of the record to write.

I/O-status Specifier

IOSTAT = Io-status

Io-status is an integer variable or an element of an integer array. The I/O system assigns Io-status one of the following values based on the outcome of the data transfer:

Io-status = 0 if no error has occurred
Io-status $\geq$ 1 if an error has occurred

Error Specifier

ERR = Error-label

Error-label is the label of an executable statement in the same program unit as the WRITE statement. If the I/O system encounters an error while processing the WRITE statement, the following actions occur:

1. The WRITE operation is terminated.

2. The file position becomes undefined, and the only valid statements that you can execute are CLOSE or INQUIRE.

3. The I/O system sets the Io-status (if one is specified).

4. The flow of control resumes at the statement labeled with Error-label. If there is no ERR specifier, a run-time error occurs.

The Output-list

The output-list can contain any of the following:

- a variable name
- an array name
- an array element name
- an expression containing any of the types CHARACTER, COMPLEX, DOUBLE PRECISION, INTEGER, LOGICAL, or REAL

If you use an array name in the output-list, the I/O system writes the entire array in the normal column-major order.

Implied-DO List

The WRITE statement can write a range of subscripted array elements from the output-list using an implied-DO list of the form:

(output-list,var = e1,e2[,e3])

where var and e1,e2, and e3 are the same as described in the explanation of the DO statement in the "Control Statements" chapter. The output-list can contain other (nested) implied-DO lists.

Restrictions

- The control-list must contain an I/O unit specifier, but can contain only one of each of the other specifiers.
- If you do not use UNIT = , then the Unit-id must be the first specifier in the control-list.
- If you do not use FMT = , then you cannot use UNIT = and the Format must be the second specifier in the control-list.
- If Unit-id designates an internal file, the control-list must contain a format specification other than an asterisk (*), but cannot contain a record number.
- If the Format is an asterisk (*) specifying list-directed formatting, the control-list cannot contain a record number specifier.

- If the file is connected for direct access, the control-list must contain a record number.

Examples

```
WRITE(10,200) name,score
WRITE(10,200,IOSTAT=errorflag,ERR=350) name,score
WRITE(3,REC=105,IOSTAT=errorflag,ERR=200) altitude
WRITE(3,REC=(2j + k),IOSTAT=errorflag,ERR=200) altitude
WRITE(6,*) employee
```

CHAPTER 15

The FORMAT Statement and Format Specification

This section describes the various methods of format specification, including the FORMAT statement and list-directed I/O. It also describes how to use edit descriptors when performing formatted data transfer.

15.1. Specifying Formats

There are three ways to specify a format:

- explicitly, in a FORMAT statement
- implicitly, as a character variable, element of a character array, or any character expression that evaluates to a valid format specification
- implicitly, as list-directed formatting (see the "List-directed Formatting" section below)

15.1.1. The FORMAT Statement

The FORMAT statement is a labeled statement that defines a format specification. It must appear in the same program unit where it is referenced.

Syntax:

statement-label FORMAT format-specification

where format-specification is a valid form of format specification as described below.

15.1.2. Character Format Specification

A character format specification must have the form of a valid format specification starting at the leftmost character position. It must be enclosed in parentheses, and any characters following the right parenthesis do not affect the format specification.

If a format specification is given as a character array name and the specification's length exceeds the length of the first array element, the specification becomes the concatenation of all the array elements in normal column-major order.

If the format specification is an array element, the specification's length cannot exceed the length of the array element.

15.2. General Form For Format Specification

A format-specification has the general form

([format list])

The format list is a list of items enclosed in parentheses. The items in the format list can be any of the following:

[r] repeatable-edit-descriptor
nonrepeatable-edit-descriptor

[r] format list

where repeatable-edit-descriptor and nonrepeatable-edit-descriptor are special character strings that describe the kind of editing being performed (see the descriptions below). "r" is a positive integer constant called the repeat-factor. If omitted, the I/O system supplies the default value r = 1.

The format list can be empty only if the corresponding I/O- list is also empty. If format list contains another (nested) format-list, the nested list cannot be empty.

If the I/O- list contains at least one item, the format list must contain at least one repeatable edit descriptor.

15.3. Format Control

The interaction between the I/O- list and the format specification is a dynamic process called format control.

Format control always proceeds from left to right, matching each item in the I/O- list with the next repeatable edit descriptor. There is no match between I/O- list items and nonrepeatable edit descriptors. Format control executes nonrepeatable edit descriptors as they are encountered.

If an edit descriptor has a repeat-factor "r", format control processes the I/O- list as if it contained "r" consecutive items.

If the format list ends before reaching the end of the I/O- list, format control reverts to the beginning of the last nested format list, if there is one. If there is none, format control reverts to the beginning of the format-specification and again passes through the I/O- list. Each time format control reverts, it accesses a new record.

15.4. Using Repeatable Edit Descriptors

Repeatable edit descriptors control the editing of character, logical, and numeric data. Each item in the I/O- list corresponds to a repeatable edit descriptor in the format list.

Each repeatable edit descriptor can be preceded by a repeat-factor, which determines how many times to repeat the edit specified by the edit descriptor.

Each repeatable edit descriptor consists of a single letter together with a number. The letter indicates the type of data to edit, and the number indicates the size of the data field.

Repeatable Edit Descriptors	
Syntax	Type of Descriptor
A[w]	Alphanumeric Descriptor
Dw.d	Floating-point Descriptor
Ew.d[Ee]	Floating-point Descriptor
Fw.d	Floating-point Descriptor
Gw.d[Ee]	Floating-point Descriptor
Iw	Integer Descriptor
Iw.m	Integer Descriptor
Lw	Logical Descriptor

In the table above, A, D, E, F, G, I, and L indicate the type of data to edit; "w" is a positive integer constant called the field width, and indicates the number of characters in the field; "d" is an unsigned integer constant indicating the number of digits following the decimal point; "e" is a positive integer constant indicating the number of digits in the ex-

ponent.

The following subsections describe how to use each repeatable edit descriptor to edit alphanumeric, numeric, and logical data.

15.4.1. Alphanumeric Editing

The A[w] edit descriptor edits CHARACTER or HOLLERITH data. If w is present, the field width is w characters. If w is not present, the field width is the length of the data item in the I/O- list.

Let len be the actual length of the item in the I/O- list. The following rules apply to A[w]:

On input,

- if w $\geq$ len, the I/O system transfers the rightmost len characters. If w $<$ len, the I/O system transfers w characters and they are left-justified, with len $-$ w trailing blanks.

On output,

- if w $>$ len, the I/O system transfers w $-$ len blanks, followed by len characters. If w $\leq$ len, the I/O system transfers the leftmost w characters.

Examples of A[w] Editing		
On Input		
Format	I/O List Item	Input Result
A10	'Mathematical'	Mathematic␣␣
A15	'Mathematical'	Mathematical
On Output		
Format	I/O List Item	Output Result
A10	Mathematical	Mathematic
A15	Mathematical	␣␣␣Mathematical

15.4.2. Numeric Editing

The D, E, F, G, and I edit descriptors edit numeric data. D, E, F, and G edit any floating-point type such as REAL*4, REAL*8, COMPLEX*8, or COMPLEX*16. I edits INTEGER data. E and G produce floating-point numbers in scientific notation (on output only).

The following rules apply to all the numeric edit descriptors:

On input,

- the I/O system ignores leading blanks, except that a field of all blanks is treated as a zero. Treatment of other blanks is determined by the BLANK specifier in the OPEN statement, and the settings of the nonrepeatable edit descriptors BN and BZ.
- a decimal point in the input field overrides the placement of the decimal point specified by a D, E, F, or G edit descriptor. Also, the input field can contain more digits than the processor needs to approximate the value.

On output,

- all negative values are prefixed with a minus sign. A positive value or zero can have a plus sign as controlled by the S, SS, and SP nonrepeatable edit descriptors.
- the I/O system right justifies all numeric values, and when necessary, pads with blanks on the left.
- if the characters in the output field exceed the field width w, the I/O system produces a field of w asterisks (*).

15.5. Floating-point Editing, D and E

The Dw.d and Ew.d[Ee] edit descriptors describe a field whose width is "w" positions with a fractional part containing "d" digits (unless the scale factor "k" > 1), and an exponent of "e" digits. When using the Dw.d and Ew.d[Ee] edit descriptors, the matching I/O- list item must be a floating-point type.

The following rules apply to Dw.d and Ew.d[Ee]:

On input,

- the input field is identical to that of the Fw.d edit descriptor; "e" has no effect.

On output,

- if the scale factor "k" = 0, the output field has the form $[+-][0].x1x2x3...xd\ exp$

  where x1x2x3...xd are the "d" most significant digits of the value after rounding, and "exp" is a decimal exponent. The form of "exp" depends on its absolute value as shown in the following table. Note that Dw.d and Ew.d are not valid if $|exp| > 999$.

D and E Editing - Exponent Forms		
Edit Descriptor	Absolute Value of exp	Exponent Form
Ew.d	|exp| ≤ 99	E+ z1z2 or 0+ z1z2
	99 < |exp| ≤ 999	+ z1z2z3
Ew.d[Ee]	|exp| ≤ (10**e)−1	E+ z1z2z3...ze
Dw.d	|exp| ≤ 99	D+ z1z2 or E+ z1z2 or + 0z1z2
	99 < |exp| ≤ 999	+ z1z2z3 (z is a digit)

- The scale factor "k" also controls decimal normalization (see the kP nonrepeatable edit descriptor) according to the following rules:

  • If −d < k < 0, then the output field has |k| leading zeros and d − |k| significant digits following the decimal point.

  • If 0 < k < d+2, then the output field has k significant digits to the left of the decimal point and d − k + 1 significant digits to the right of the decimal point.

  • No other values of k are valid.

Examples of Dw.d and Ew.d[Ee] Editing		
On Input		
Format	I/O List Item	Input Result
D9.2	999999.99	9.9999999D+05
D14.4	20583.4077D+03	2.05834077D+07
E9.2	3.31587E2	3.31587E2
E10.3	␣␣81.081E3	8.1081E4
On Output		
Format	I/O List Item	Output Result
D15.3	0.0181	␣␣␣␣␣0.181D−01
D8.1	0	␣0.0D+00
E10.2	1216641.731	␣␣0.12E+07
E12.4	1216641.731	␣␣0.1217E+07

15.6. Floating-point Editing, F

The Fw.d edit descriptor describes a field whose width is w positions, with a fractional part containing d digits. When using Fw.d, the I/O- list item must be a floating-point type.

The following rules apply to Fw.d:

On input,

- the field can contain an optional sign, followed by digits that can optionally contain a decimal point. If there is no decimal point, the I/O system interprets the rightmost "d" digits in the field as the fractional part. The input field can contain more digits than are needed by the processor to approximate the value.
- the input field can be followed by an exponent expressed as

  • a signed integer constant

  • the character D or E followed by zero or more blanks, followed by an optionally signed integer constant.

On output,

- the output field consists of any necessary blanks followed by digits containing a decimal point that represent the internal value rounded to "d" fractional digits and modified by the established scale factor (see the kP nonrepeatable edit descriptor).
- if the value is negative, the output field is prefixed with a minus sign. If the value is positive, the output field can have an optional plus sign.
- if the absolute value of the internal data is less than one, the output field can have an optional zero immediately to the left of the decimal point.

Examples of Fw.d Editing		
On Input		
Format	I/O List Item	Input Result
F7.2	6671878	66718.78
F9.5	−10.24E+2	−1024.0
On Output		
Format	I/O List Item	Output Result
F9.4	2.71828	␣␣␣2.7183
F8.3	−98.87314	␣−98.873

15.7. Floating-point Editing, G

The Gw.d[Ee] edit descriptors describe a field whose width is w positions with a fractional part containing "d" digits (unless the scale factor k > 1), and an exponent of "e" digits. When using the Gw.d[Ee] edit descriptor, the I/O- list item must be a floating-point type.

The following rules apply to Gw.d[Ee]:

On input,

- the input field is identical to that of the Fw.d edit descriptor; "e" has no effect.

On output,

- the output field depends on the M, the magnitude of the I/O- list item.

  • If M < 0.1 or M ≥ 10**d, then the output field is identical to that produced by Gw.d[Ee] using the current scale factor k.

  • If 0.1 ≤ M < 10**d, then M is inside the range that permits Fw.d editing. In this case, the I/O system ignores the current scale factor k, and M produces an equivalent conversion as shown in the following table.

Gw.d[Ee] Conversion when 0.1 ≤ M < 10**d	
Value of M	Equivalent Conversion
0.1≤M< 1	F(w−n).d,n(b)
1≤M<10	F(w−n).(d−1),n(b)
10**(d−2)≤M<10**(d−1)	F(w−n).1,n(b)
10**(d−1)≤M<10**d	F(w−n).0,n(b)
n(b) = 4 blank spaces for Gw.d n(b) = e+2 blank spaces for Gw.d[Ee]	

Examples of Gw.d[Ee] Editing		
On Input		
Format	I/O List Item	Input Result
G7.2	6671878	66718.78
G9.5	−10.24E+2	−1024.0
On Output		
Format	I/O List Item	Output Result
G12.5	864.50695	bb864.51bbbb
G12.5	−1910.66666	b−1910.7bbbb

15.8. Complex Editing

A complex data value is represented as a pair of values: a real part and an imaginary part. Editing of complex data is accomplished by the successive interpretation of two D, E, F, or G edit descriptors. The first descriptor describes the real part, and the second descriptor describes the imaginary part.

The two edit descriptors can be different, and nonrepeatable edit descriptors can appear between any two successive D, E, F, or G edit descriptors.

Examples of Complex Editing		
On Input		
Format	I/O List Item	Input Result
2F9.3	28829809856777.765	288298.098, 56777.765
D9.2,E9.2	999999.993,31587E2	9.9999999D+05, 3.31587E2
On Output		
Format	I/O List Item	Output Result
D8.1,D8.3	0.0, 3.14159	b0.0D+00.314D+01
2E9.2	283.2394, 0.129312	b0.28E+03b0.13E+00

15.8.1. Integer Editing

The Iw and Iw.m edit descriptors describe a field whose width is "w" positions. When using Iw or Iw.m, the I/O- list must be of type INTEGER, and consist of at least one digit.

The following rules apply to Iw and Iw.m:

On input,

- the input field can be an optionally signed integer constant. Leading blanks are treated as described above.

On output,

- for Iw, the output field consists of an integer constant. If the value is positive, it can be prefixed with an optional plus sign. If the value is negative, it is prefixed by a minus sign. Leading blanks are treated as described above.
- for Iw.m, the output field is identical to that produces by Iw, except that it must have at least "m" digits, and if necessary, have leading zeros. The value of "m" cannot exceed "w". If m = 0 and the internal value of the I/O- list item is zero, the output field consists of all blanks regardless of any sign control in effect.

Examples of Iw and Iw.m Editing		
On Input		
Format	I/O List Item	Input Result
I5	+128	128
I5	−999	−999
I5	␣␣␣␣0	0
On Output		
Format	I/O List Item	Output Result
I4	+128	␣␣128
I4	−999	␣−999
I5.3	1	␣␣001
I5.3	−1	␣−001

15.8.2. Logical Editing

The Lw edit descriptor describes a field whose width is w positions. When using Lw, the I/O- list item must be of type LOGICAL.

The following rules apply to Lw:

On input,

- the field can contain optionally contain leading blanks and a decimal point, followed by the character T for true or F for false. The logical constants .TRUE. and .FALSE. are also acceptable as input.

On output,

- the output field contains (w−1) leading blanks, followed by the character T if the value is true, or F if it is false.

Examples of Lw Editing		
On Input		
Format	I/O List Item	Input Result
L1	T	.TRUE.
L3	␣␣T	.TRUE.
L7	.FALSE.	.FALSE.
On Output		
Format	I/O List Item	Output Result
L4	.TRUE.	␣␣␣T
L1	.FALSE.	F

15.9. Using Nonrepeatable Edit Descriptors

Nonrepeatable edit descriptors are not associated with specific data items in the I/O- list. Instead, they control such things as column position, spacing, sign control, blank control, and line termination.

Nonrepeatable Edit Descriptors	
Syntax	Type of Descriptor
'c1,c2,...cn'	Apostrophe (Literal-string) Descriptor
nHc1,c2,...cn	Hollerith-string Descriptor
BN	Blank-control Descriptor
BZ	Blank-control Descriptor
kP	Scale-factor Descriptor
S	Sign-control Descriptor
SP	Sign-control Descriptor
SS	Sign-control Descriptor
Tc	Position Descriptor
TLc	Position Descriptor
TRc	Position Descriptor
nX	Position Descriptor
/	Line-termination Descriptor
:	Conditional line-termination Descriptor

In this table, "c" is any printable ASCII character, "n" is a positive integer constant, and "k" is an optionally signed integer constant that represents a scale factor.

The following subsections describe how to use each nonrepeatable edit descriptor.

15.9.1. Apostrophe Descriptor '

The I/O system transfers literal character strings when they are enclosed in single apostrophes. This is called apostrophe editing, and is valid only on output. The field width is length of the character string. A single apostrophe inside the string must be written as two consecutive apostrophes.

Example of Apostrophe Editing	
Format	Output Result
'Today''s␣date␣is:'	Today's␣date␣is:

15.9.2. Hollerith Descriptor

The nH edit descriptor is an alternative method for transferring literal character strings. nH causes the I/O system to transfer "n" characters following the H. Like apostrophe editing, it is valid only on output.

If the character string contains a single apostrophe, it is counted as one character when specifying "n".

Examples of Hollerith Editing		
Format	I/O List Item	Output Result
3H	ABC	ABC
4H	IT'S	IT'S

15.9.3. Blank-control Descriptors BN and BZ

BN and BZ control the interpretation of blanks (other than leading blanks) on input only with D, E, F, G, and I editing; they have no effect on output.

Prior to executing any formatted I/O statement, the BLANK specifier currently in effect for the unit, determines the interpretation of blanks.

When the I/O system encounters BN in a format specification, it ignores all blank characters in any succeeding input fields. Ignoring blanks has the same effect as removing blanks, right-justifying the field, and replacing the blanks as leading blanks.

When the I/O system encounters BZ in a format specification, it treats all blank characters in succeeding input fields as zeros.

Once specified, BN and BZ remain in effect until changed explicitly, or the I/O statement finishes executing.

Examples of BN and BZ Editing		
Format	I/O List Item	Input Result
BN	12␣34␣	1234
BZ	12␣34␣	120340

15.9.4. Scale-factor Descriptor kP

The kP edit descriptor establishes a scale factor when using D, E, F, or G editing.

Prior to executing an I/O statement, the scale factor is zero. Once set with the kP descriptor, the value of k remains in effect until changed with another another kP descriptor, or the I/O statement finishes executing.

The scale factor k produces the following effects:

On input,

- "k" has no effect with D, E, F, and G editing if there is an exponent in the field.
- with D, E, F, and G editing, the externally represented number equals the internal representation multiplied by 10**k.

On output,

- with D and E editing, the mantissa is multiplied by 10**k, which moves the decimal point "k" positions to the right (or left, if negative), and the exponent is reduced by "k".
- with F editing, the externally represented number equals the internal representation multiplied by 10**k.
- with G editing, "k" is ignored unless the output value is outside the range of F editing. If E editing is required, "k" has the same effect as described for E editing.

Note: When kP immediately follows a D, E, F, or G edit descriptor, a comma is not required between items.

Table 14-15 illustrates the effect of a scale factor when used with floating-point edit descriptors.

Examples of Floating-point Editing with a Scale Factor		
On Input		
Format	I/O List Item	Input Result
2PF10.4	␣␣125.63␣␣	1.2563
On Output		
Format	I/O List Item	Output Result
2PF10.2	104.12345	10412.345
2PD15.3	0.0181	18.10D−03

15.9.5. Sign-control Descriptors S, SP, and SS

The S, SS, and SP edit descriptors control the optional plus sign character in numeric output fields. S, SS, and SP affect the D, E, F, G and I edit descriptors on output only; they have no effect on input.

When executing any formatted I/O statement, the I/O system normally has the option to produce a plus sign in numeric output fields. An SP edit descriptor directs the I/O system to always produce a plus sign in any subsequent position that normally contains an optional plus sign.

An SS edit descriptor directs the I/O system to always suppress a plus sign in any subsequent position that normally contains an optional plus sign.

An S edit descriptor restores to the I/O system the option of producing a plus sign in numeric output fields.

Examples of Editing with Sign-control Descriptors		
Format	I/O List Item	Output Result
SS	5	5
SP	5	+5

15.9.6. Position Descriptors Tc, TLc, TRc, and nX

The Tc, TLc, TRc, and nX edit descriptors determine the position at which the I/O system transfers the next character to or from a record.

Tc specifies that transfer of the next character to or from a record occurs at the cth position.

TRc specifies that transfer of the next character to or from a record occurs at c positions to the right of the current position.

TLc specifies that transfer of the next character to or from a record occurs at c positions to the left of the current position. If the current position is less than or equal to c, TLc transfers the next character at position one of the current record.

nX specifies that transfer of the next character to or from a record occurs at n positions to the right of the current record.

The following rules apply to the position descriptors:

On input,

- T can specify a position in either direction from the current position. This allows the I/O system to process part of a record more than once, possibly with different editing.
- nX can specify a position beyond the last position in a record if no characters are transferred from such a position.

On output,

- when the I/O system transfers characters to positions at or following the position specified by Tc, TRc, TLc, or nX, any positions that are skipped and not previously filled are padded with blanks.
- a Tc, TRc, TLc or nX edit descriptor cannot replace an existing character within a record, but they can affect position such that subsequent editing causes a replacement.

15.9.7. Line-termination Descriptor /

The line-termination descriptor / indicates the end of data transfer on the current record.

The following rules apply to the line-termination descriptor:

On input,

- if the file is connected for sequential access, the I/O system skips the rest of the current record, and positions the file at the initial point of the next record, which then becomes the current record.
- if the file is positioned at the initial point of a record, the I/O system skips the entire record.

On output,

- if the file is connected for sequential access, the I/O system creates a new record, which then becomes the current and last record in the file.
- if the file is connected for direct access, the I/O system increments the current record number by one, and positions the file at the initial point of the new record, which then becomes the current record.
- the I/O system can output an empty record. If the file is connected for direct access or is an internal file, the record contains blanks.

Note: A comma is not required before or after the / and any I/O- list items.

The following example illustrates the line-termination descriptor.

Format	Output Result
(1X,'ABC'//1X,'DEF')	␣ABC
	␣
	␣
	␣DEF

15.9.8. Conditional Line-termination Descriptor, :

The conditional line-termination descriptor ":" terminates format control if there are no more items in the I/O- list. If there are remaining items in the I/O- list, the I/O sys-

tem ignores the ":" descriptor.

The following example illustrates the conditional line-termination descriptor.

```
                Print Statement                 Output Result
                PRINT 10,5
10              FORMAT (IX,'I=',I2,'␣J=',I2)    I=␣5␣J=

                PRINT 20,6
20              FORMAT (IX,'I=',I2,:'␣J=',I2)   I=␣6
```

15.10. List-directed Formatting

List-directed formatting is specified by an asterisk (*) as the format specification in an I/O statement. An explicit FORMAT statement is not required.

A list-directed file is an external file containing list-directed records. A list-directed record is a sequence of characters that are either values or value separators.

Each value can be

- a constant
- a null value
- one of the forms r*c or r*

where r is an positive integer constant repeat-factor, and c is a character value. r*c is equivalent to r successive occurrences of c; r* is equivalent to r successive null values. Neither form can contain any embedded blanks, other than those within c.

A value separator can be

- a comma, optionally preceded or followed by one or more contiguous blanks
- a slash (/), optionally preceded or followed by one or more contiguous blanks
- one or more contiguous blanks between two constants, or following the last constant

The following rules apply to list-directed formatting:

- The I/O system treats blanks as separators, so embedded blanks are allowed only inside character strings.
- The I/O system treats the end of a record as a blank, except inside a character string.
- The end of a record following any separator with or without any intervening blanks does not imply a null value.
- There are two ways to specify a null value:
  - with the r* form
  - by having no characters precede the first value separator, or appear between the successive value separators.

15.10.1. List-directed Input

When the I/O system executes a list-directed READ statement, it begins a new record, and formats each input value using the data type and field width of the corresponding I/O- list item to generate an equivalent edit descriptor.

The following rules apply to list-directed input:

- When the I/O system encounters a null value while executing a list-directed input statement, the null value does not affect the corresponding I/O- list item.

The item retains its value, or if it is undefined, it remains undefined.

- A null value cannot appear as the real or imaginary part of a complex constant, but a single null value can represent a whole complex constant.
- When the I/O system encounters a slash (/) as a value separator while executing a list-directed input statement, it stops executing the statement at that point, and treats any further items in the I/O- list as null values.
- If the I/O- list item is of type CHARACTER*n, the input value must be a nonempty character string enclosed in single apostrophes. Commas, blanks, and slashes (/) are all valid inside character-string constants. An apostrophe inside a character string must be written as two consecutive apostrophes.

  A character-string constant can continue from the end of one record to the beginning of the next for as many records as are needed. The end of a record does not cause a blank character to appear in the constant.

  If w is the field width and the input value is CHARACTER*n, the I/O system transfers characters as follows:
  - if $w \leq n$, the leftmost w characters are transferred
  - if $w > n$, the leftmost n characters are transferred and the remaining $w - n$ positions are padded with blanks

  This is the same effect as assigning the I/O- list item in an ordinary assignment statement.
- If the I/O- list is of type COMPLEX*8 or COMPLEX*16, the input value consists of a pair of numeric input fields separated by a comma, and enclosed in parentheses. The first field contains the real part of the complex constant, and the second field contains the imaginary part. Each field can be preceded or followed by one or more contiguous blanks.
- If the I/O- list item is of type LOGICAL, the input value cannot contain any commas or slashes (/) embedded among the optional characters after the T or F.
- If the I/O- list item is of type REAL*4 or REAL*8, the input value has the form of a numeric input field suitable for F editing. That is, it has no fractional digits unless a decimal point appears in the field.

The table below summarizes the correspondence between the data type of the I/O- list item and the equivalent edit descriptors.

Equivalence of Edit Descriptors Using List-directed Input		
Data Type of Input Item	Equivalent Edit Descriptor	
CHARACTER*n	Aw	if $w \leq n$
	An,(w−n)X	if $w > n$
COMPLEX*8 or COMPLEX*16	(Fw.0,Fw.0)	
LOGICAL	Lw	
REAL*4 or REAL*8	Fw.0	

15.10.2. List-directed Output

When the I/O system executes a list-directed WRITE or PRINT statement, it begins a new record, and formats each output value using the data type and field width of the corresponding I/O- list item to generate an equivalent edit descriptor.

The following rules apply to list-directed output:

- Each output record begins with a blank to provide carriage control when printing.
- Output values are separated by one or more blanks.
- The I/O system treats blanks as separators, so embedded blanks are allowed only inside character strings.
- The I/O system treats the end of a record as a blank, except inside a character string.
- When the I/O system outputs a CHARACTER*n constant, the constant is not enclosed in single apostrophes, or preceded or followed by a value separator. The I/O system can insert a blank for carriage control if a record begins with the continuation of a character constant from the preceding record.
- When the I/O system outputs a COMPLEX*n constant, the constant is enclosed in parentheses with a comma separating the real and imaginary parts, which are edited according the rules for REAL*k values where k = n/2.
- The I/O system outputs an INTEGER*n value using the Iw edit descriptor.
- The I/O system outputs a LOGICAL constant using T for the value true or F for the value false.
- The I/O system outputs a REAL*n constant, the constant is represented using G format.

The table below summarizes the correspondence between the data type of the I/O- list item and the equivalent edit descriptors.

Equivalence of Edit Descriptors Using List-directed Output

Data Type	Output Format
LOGICAL	I5
INTEGER*2	I7
INTEGER*4	I12
REAL*4	1PG15.7
REAL*8	1PG25.16
COMPLEX*8	1X,'(',1PG14.7,',', 1PG14.7,')'
COMPLEX*16	1X,'(',1PG25.16,',',1PG25.16,',)'
CHARACTER*n	1X, An

CHAPTER 16

Fortran-386 Intrinsic Functions

This section presents the Fortran-386 intrinsic functions in alphabetical order according to the generic function name. Each function description contains a brief identification of function usage, a syntax specification, a description of the function, and a listing of specific names.

You can reference an intrinsic function with the generic name or you can use a specific function name. The specific name- list identifies the data type for the arguments and the type of return value. Specific names reference a particular part of a function to perform a more specific operation. A dash in the specific name column indicates that you must use the generic name for the corresponding specific operation.

16.1. ABS Function

Usage:

absolute value

Syntax:

x=ABS(number)

The ABS function returns the absolute value of an integer, real, or complex number.

Note that the INT function returns the absolute value of a complex number with a real data type.

Specific Names	Type of Argument	Type of Return Value {x}
IABS	INTEGER	INTEGER
ABS	REAL*4	REAL*4
DABS	REAL*8	REAL*8
CABS	COMPLEX	REAL*8

16.2. ACOS Function

Usage:

trigonometric arccosine

Syntax:

x = ACOS(number)

The ACOS function returns the trigonometric arccosine of a real number.

Specific Names	Type of Argument	Type of Return Value {x}
ACOS	REAL*4	REAL*4
DACOS	REAL*8	REAL*8

16.3. AIMAG Function

Usage:

imaginary part of a complex value

Syntax:

x = AIMAG(complex-number)

The AIMAG function returns the imaginary part of a complex-number with a real data type.

Specific Names	Type of Argument	Type of Return Value {x}
AIMAG	COMPLEX	REAL*4

16.4. AINT Function

Usage:

truncation

Syntax:

x = AINT(number)

The AINT function truncates a real number to an integer but maintains the original data type specification. The AINT function does not convert a number to the integer data type. If the number you want to truncate is an integer, the AINT function simply returns that integer.

If the number you want to truncate is a real number with an absolute value less than 1, the AINT function returns 0. If the number you want to truncate is a real number with an absolute value greater than 1, the AINT function returns the largest integer that does not exceed the value of the original number.

Specific Names	Type of Argument	Type of Return Value {x}
AINT	REAL*4	REAL*4
DINT	REAL*8	REAL*8

16.5. ANINT Function

Usage:

nearest whole number

Syntax:

x = ANINT(number)

The ANINT function translates a real number to the nearest whole number value and maintains the original data type. If the number you want to translate is an integer, the AINT function simply returns that integer.

Specific Names	Type of Argument	Type of Return Value {x}
ANINT	REAL*4	REAL*4
DNINT	REAL*8	REAL*8

16.6. ASIN Function

Usage:

trigonometric arcsine

Syntax:

x = ASIN(number)

The ASIN function returns the trigonometric arcsine of a real number.

Specific Names	Type of Argument	Type of Return Value {x}
ASIN	REAL*4	REAL*4
DASIN	REAL*8	REAL*8

16.7. ATAN Function

Usage:

trigonometric arctangent

Syntax:

x = ATAN(number)

The ATAN function returns the trigonometric arctangent of a real number.

Specific Names	Type of Argument	Type of Return Value {x}
ATAN	REAL*4	REAL*4
DATAN	REAL*8	REAL*8

16.8. ATAN2 Function

Usage:

trigonometric arctangent of a quotient

Syntax:

x = ATAN2(number,number)

The ATAN2 function returns the trigonometric arctangent of a quotient. The first number argument is the dividend. The second number argument is the divisor.

Specific Names	Type of Argument	Type of Return Value {x}
ATAN2	REAL*4	REAL*4
DATAN2	REAL*8	REAL*8

16.9. CHAR Function

Usage:

character data type conversion

Syntax:

x = CHAR(integer)

The CHAR function converts an integer value to the corresponding ASCII character representation. The integer argument for CHAR must range from 0 to 127.

Specific Names	Type of Argument	Type of Return Value {x}
CHAR	INTEGER	CHARACTER

16.10. CMPLX Function

Usage:

numeric data type conversion

Syntax:

x = CMPLX(number[,number])

The CMPLX function converts an integer or real number to a complex number. If you use CMPLX with one argument, the function uses the argument for the real portion of the complex value. The imaginary portion becomes 0.

If you use CMPLX with two arguments, the function uses the first argument for the real portion of the complex value and the second argument for the imaginary part. Both arguments must have the same data type.

Specific Names	Type of Argument	Type of Return Value {x}
-	INTEGER	COMPLEX
-	REAL*4	COMPLEX
-	REAL*8	COMPLEX
-	COMPLEX	COMPLEX

16.11. CONJG Function

Usage:

conjugate of a complex argument

Syntax:

x = CONJG(complex-number)

The CONJG function returns the conjugate of a complex-number.

Specific Names	Type of Argument	Type of Return Value {x}
CONJG	COMPLEX*8	COMPLEX
DCONJG	COMPLEX*16	COMPLEX*16

16.12. COS Function

Usage:

trigonometric cosine

Syntax:

x = COS(number)

The COS function returns the trigonometric cosine of a real or complex number.

Specific Names	Type of Argument	Type of Return Value {x}
COS	REAL*4	REAL*4
DCOS	REAL*8	REAL*8
CCOS	COMPLEX	COMPLEX

16.13. COSH Function

Usage:

trigonometric hyperbolic cosine

Syntax:

x = COSH(number)

The COSH function returns the trigonometric hyperbolic cosine of a real number.

Specific Names	Type of Argument	Type of Return Value {x}
COSH	REAL*4	REAL*4
DCOSH	REAL*8	REAL*8

16.14. DBLE Function

Usage:

numeric data type conversion

Syntax:

x = DBLE(number)

The DBLE function converts an integer, real, or complex number to a double precision real number. If the number you want to convert is already a double precision real number, the DBLE function simply returns that double precision number.

For a complex number, the DBLE function ignores the imaginary portion and returns the real portion converted to double precision.

Specific Names	Type of Argument	Type of Return Value {x}
-	INTEGER	REAL*8
-	REAL*4	REAL*8
-	REAL*8	REAL*8
-	COMPLEX	REAL*8

16.15. DIM Function

Usage:

positive difference

Syntax:

x = DIM(number,number)

The DIM function returns the difference between two integers or real numbers, if that difference is a positive value. If the first number argument is greater than the second, DIM returns the positive difference. If the first number argument is less than the second, DIM returns 0.

Specific Names	Type of Argument	Type of Return Value {x}
IDIM	INTEGER	INTEGER
DIM	REAL*4	REAL*4
DDIM	REAL*8	REAL*8

16.16. DPROD Function

Usage:

double precision product

Syntax:

x = DPROD(real-number,real-number)

The DPROD function returns the product of two real number factors as a double precision real number.

Specific Names	Type of Argument	Type of Return Value {x}
DPROD	REAL*4	REAL*8

16.17. EXP Function

Usage:

exponential

Syntax:

x = EXP(number)

The EXP function returns the constant e raised to a specified real or complex exponent. The number is the specified exponent.

Specific Names	Type of Argument	Type of Return Value {x}
EXP	REAL*4	REAL*4
DEXP	REAL*8	REAL*8
CEXP	COMPLEX	COMPLEX

16.18. ICHAR Function

Usage:

character data type conversion

Syntax:

x = ICHAR(character)

The ICHAR function converts an ASCII character to its corresponding decimal integer value. The character argument for ICHAR must have a length of 1.

Specific Names	Type of Argument	Type of Return Value {x}
ICHAR	CHARACTER	INTEGER

16.19. INDEX Function

Usage:

index of a substring

Syntax:

x = INDEX(string,substring)

The INDEX function returns an integer value that indicates the starting position of a substring within a string.

Specific Names	Type of Argument	Type of Return Value {x}
INDEX	CHARACTER	INTEGER

16.20. INT Function

Usage:

numeric data type conversion

Syntax:

x = INT(number)

The INT function converts a real or complex number to an integer. If the number you want to convert is already an integer, the INT function simply returns that integer.

If the number you want to convert is a real number with an absolute value less than 1, the INT function returns 0. If the number you want to convert is a real number with an absolute value greater than 1, the INT function returns the largest integer that does not exceed the value of the original number.

If the number you want to convert is a complex number, the INT function applies the same rules as for real numbers to the real portion of the complex number. The INT function ignores the imaginary portion of a complex number.

Specific Names	Type of Argument	Type of Return Value {x}
-	INTEGER	INTEGER
INT	REAL*4	INTEGER
IFIX	REAL*4	INTEGER
IDINT	REAL*8	INTEGER
-	COMPLEX*8	INTEGER

16.21. LEN Function

Usage:

character value length

Syntax:

x = LEN(character-entity)

The LEN function returns an integer value that indicates the length of a specified character-entity.

Specific Names	Type of Argument	Type of Return Value {x}
LEN	CHARACTER	INTEGER

16.22. LGE Function

Usage:

character value relational

Syntax:

x = LGE(string,string)

The LGE function compares two strings according to the rules of the ASCII character collating sequence. LGE returns a logical true value if the first string argument equals or follows the second string argument in the collating sequence. Otherwise, LGE returns a logical false.

If the string arguments are unequal in length, the LGE function pads the shorter string on the right with blanks.

Specific Names	Type of Argument	Type of Return Value {x}
LGE	CHARACTER	LOGICAL

16.23. LGT Function

Usage:

character value relational

Syntax:

x = LGT(string,string)

The LGT function compares two strings according to the rules of the ASCII character collating sequence. LGT returns a logical true value if the first string argument follows the second string argument in the collating sequence. Otherwise, LGT returns a logical false.

If the string arguments are unequal in length, the LGT function pads the shorter string on the right with blanks.

Specific Names	Type of Argument	Type of Return Value {x}
LGT	CHARACTER	LOGICAL

16.24. LLE Function

Usage:

character value relational

Syntax:

x = LLE(string,string)

The LLE function compares two strings according to the rules of the ASCII character collating sequence. LLE returns a logical true value if the first string argument equals or precedes the second string argument in the collating sequence. Otherwise, LLE returns a logical false.

If the string arguments are unequal in length, the LLE function pads the shorter string on the right with blanks.

Specific Names	Type of Argument	Type of Return Value {x}
LLE	CHARACTER	LOGICAL

16.25. LLT Function

Usage:

character value relational

Syntax:

x = LLT(string,string)

The LLT function compares two strings according to the rules of the ASCII character collating sequence. LLT returns a logical true value if the first string argument precedes the second string argument in the collating sequence. Otherwise, LLT returns a logical false.

If the string arguments are unequal in length, the LLT function pads the shorter string on the right with blanks.

Specific Names	Type of Argument	Type of Return Value {x}
LLT	CHARACTER	LOGICAL

16.26. LOG Function

Usage:

natural logarithm

Syntax:

x = LOG(number)

The LOG function returns the natural logarithm of a real or complex number.

Specific Names	Type of Argument	Type of Return Value {x}
ALOG	REAL*4	REAL*4
DLOG	REAL*8	REAL*8
CLOG	COMPLEX	COMLPEX

16.27. LOG10 Function

Usage:

common logarithm

Syntax:

x = LOG10(number)

The LOG10 function returns the base 10 logarithm of a real number.

Specific Names	Type of Argument	Type of Return Value {x}
ALOG10	REAL*4	REAL*4
DLOG10	REAL*8	REAL*8

16.28. MAX Function

Usage:

selecting the largest value

Syntax:

x = MAX(number,number[,number...])

The MAX function returns the largest value from a list of integer or real numbers. All arguments that you specify must have the same data type.

The specific function AMAX0 determines the largest value from a list of integers and converts the data type to real. MAX1 determines the largest value from a list of real numbers and converts the data type to integer.

Specific Names	Type of Argument	Type of Return Value {x}
MAX0	INTEGER	INTEGER
AMAX1	REAL*4	REAL*4
DMAX1	REAL*8	REAL*8
AMAX0	INTEGER	REAL*4
MAX1	REAL*4	INTEGER

16.29. MIN Function

Usage:

selecting the smallest value

Syntax:

x = MIN(number,number[,number...])

The MIN function returns the smallest value from a list of integer or real numbers. All arguments that you specify must have the same data type.

The specific function AMIN0 determines the smallest value from a list of integers and converts the data type to real. MIN1 determines the smallest value from a list of real numbers and converts the data type to integer.

Specific Names	Type of Argument	Type of Return Value {x}
MIN0	INTEGER	INTEGER
AMIN1	REAL*4	REAL*4
DMIN1	REAL*8	REAL*8
AMIN0	INTEGER	REAL*4
MIN1	REAL*4	INTEGER

16.30. MOD Function

Usage:

remaindering

Syntax:

x = MOD(number,number)

The MOD function returns the remainder from an integer or real number division. MOD divides the first number argument by the second and returns the remainder.

Specific Names	Type of Argument	Type of Return Value {x}
MOD	INTEGER	INTEGER
AMOD	REAL*4	REAL*4
DMOD	REAL*8	REAL*8

16.31. NINT Function

Usage:

nearest integer

Syntax:

x = NINT(number)

The NINT function converts a real number to the nearest integer value. Note that the NINT function converts the data type to integer.

Specific Names	Type of Argument	Type of Return Value {x}
NINT	REAL*4	INTEGER
IDNINT	REAL*8	INTEGER

16.32. REAL Function

Usage:

numeric data type conversion

Syntax:

x = REAL(number)

The REAL function converts an integer, real, or complex number to the REAL*4 data type. If the number you want to convert is already a REAL*4, the REAL function simply returns that number.

For a complex number, the REAL function simply ignores the imaginary portion and returns the real portion.

Specific Names	Type of Argument	Type of Return Value {x}
REAL	INTEGER	REAL*4
FLOAT	INTEGER	REAL*4
-	REAL*4	REAL*4
SNGL	REAL*8	REAL*4
-	COMPLEX	REAL*4

16.33. SIGN Function

Usage:

transfer of sign

Syntax:

x = SIGN(number,number)

The SIGN function transfers the sign of the second number argument to the first number argument. SIGN returns the absolute value of the first argument if the second argument is greater than or equal to 0, and the negative of the first argument if the second argument is less than 0.

Specific Names	Type of Argument	Type of Return Value {x}
ISIGN	INTEGER	INTEGER
SIGN	REAL*4	REAL*4
DSIGN	REAL*8	REAL*8

16.34. SIN Function

Usage:

trigonometric sine

Syntax:

x = SIN(number)

The SIN function returns the trigonometric sine of a real or complex number.

Specific Names	Type of Argument	Type of Return Value {x}
SIN	REAL*4	REAL*4
DSIN	REAL*8	REAL*8
CSIN	COMPLEX	COMPLEX

16.35. SINH Function

Usage:

trigonometric hyperbolic sine

Syntax:

x = SINH(number)

The SINH function returns the trigonometric hyperbolic sine of a real number.

Specific Names	Type of Argument	Type of Return Value {x}
SINH	REAL*4	REAL*4
DSINH	REAL*8	REAL*8

16.36. SQRT Function

Usage:

square root

Syntax:

x = SQRT(number)

The SQRT function returns the square root of a real or complex number.

Specific Names	Type of Argument	Type of Return Value {x}
SQRT	REAL*4	REAL*4
DSQRT	REAL*8	REAL*8
CSQRT	COMPLEX	COMPLEX

16.37. TAN Function

Usage:

trigonometric tangent

Syntax:

x = TAN(number)

The TAN function returns the trigonometric tangent of a real number.

Specific Names	Type of Argument	Type of Return Value {x}
TAN	REAL*4	REAL*4
DTAN	REAL*8	REAL*8

16.38. TANH Function

Usage:

trigonometric hyperbolic tangent

Syntax:

x = TANH(number)

The TANH function returns the trigonometric hyperbolic tangent of a real number.

Specific Names	Type of Argument	Type of Return Value {x}
TANH	REAL*4	REAL*4
DTANH	REAL*8	REAL*8

CHAPTER 17

ASCII and Hexadecimal Conversions

DECIMAL	ASCII	HEX	DECIMAL	ASCII	HEX	DECIMAL	ASCII	HEX	DECIMAL	ASCII	HEX
0	NUL	00	32	SP	20	64	@	40	96	`	60
1	SOH	01	33	!	21	65	A	41	97	a	61
2	STX	02	34	"	22	66	B	42	98	b	62
3	ETX	03	35	#	23	67	C	43	99	c	63
4	EOT	04	36	$	24	68	D	44	100	d	64
5	ENQ	05	37	%	25	69	E	45	101	e	65
6	ACK	06	38	&	26	70	F	46	102	f	66
7	BEL	07	39	'	27	71	G	47	103	g	67
8	BS	08	40	(	28	72	H	48	104	h	68
9	HT	09	41	)	29	73	I	49	105	i	69
10	LF	0A	42	*	2A	74	J	4A	106	j	6A
11	VT	0B	43	+	2B	75	K	4B	107	k	6B
12	FF	0C	44	,	2C	76	L	4C	108	l	6C
13	CR	0D	45	–	2D	77	M	4D	109	m	6D
14	SO	0E	46	.	2E	78	N	4E	110	n	6E
15	SI	0F	47	/	2F	79	O	4F	111	o	6F
16	DLE	10	48	0	30	80	P	50	112	p	70
17	DC1	11	49	1	31	81	Q	51	113	q	71
18	DC2	12	50	2	32	82	R	52	114	r	72
19	DC3	13	51	3	33	83	S	53	115	s	73
20	DC4	14	52	4	34	84	T	54	116	t	74
21	NAK	15	53	5	35	85	U	55	117	u	75
22	SYN	16	54	6	36	86	V	56	118	v	76
23	ETB	17	55	7	37	87	W	57	119	w	77
24	CAN	18	56	8	38	88	X	58	120	x	78
25	EM	19	57	9	39	89	Y	59	121	y	79
26	SUB	1A	58	:	3A	90	Z	5A	122	z	7A
27	ESC	1B	59	;	3B	91	[	5B	123	{	7B
28	FS	1C	60	<	3C	92	\	5C	124	|	7C
29	GS	1D	61	=	3D	93	]	5D	125	}	7D
30	RS	1E	62	>	3E	94	^	5E	126	~	7E
31	US	1F	63	?	3F	95	_	5F	127	DEL	7F

CHAPTER 18

Fortran Glossary

The FORTRAN-77 standard clearly defines all concepts and terminology specific to the language. This glossary presents the general Fortran-386 terms listed alphabetically for easy reference. Most of the terms are explained in greater detail within the manual along with terms of more specialized meaning.

array: Sequence of data items collectively identified with one unique symbolic name and a data type.

array elements: Individual data items that form an array. To reference a particular element in an array, specify the array name with a subscript. The subscript value is an integer expression that determines which element is referenced.

array declarator: Symbolic name and number of dimensions in an array. The number of dimensions determines the number and configuration of array elements.

association: Enables data to be identified by different symbolic names within the same program unit or in different program units within the same executable program. There are four forms of association: common, equivalence, argument, and entry.

block data subprogram: Nonexecutable program unit used to provide initial values for variables and array elements in named common blocks. A block data subprogram has a BLOCK DATA statement as its first statement.

character storage unit: Amount of storage required to hold one character of data. Fortran-386 uses one byte of storage for one character of data. The storage unit establishes a means of referring to data storage without implying a specific storage technology.

comment line: Character sequence within the program code, used to provide program documentation. A comment line does not affect an executable program in any way. All comment lines start with the letter C or an asterisk.

constant: Program entity that has an unchanging value during the execution of a program. Constants can be arithmetic constants, logical constants, or character constants.

continuation line: Used to contain portions of a Fortran-386 statement that exceed the 72 character columns available in the initial line for statement syntactic items. A statement can have up to nineteen continuation lines.

defined: Definition status of a program entity. A defined entity has a value that does not change until the entity becomes undefined or is redefined with a different value. An entity must be defined before it can be referenced.

definition status: Defined or undefined condition of a syntactic entity.

dummy argument: Symbolic name or an asterisk, *, used in the argument list of a procedure. Symbolic name dummy arguments hold a place for actual arguments passed to the procedure in the procedure reference. Symbolic name dummy arguments can be variables, arrays, array elements, functions, or subroutines, but must correspond to the number, type, and sequence of actual arguments in the procedure reference. An asterisk dummy argument indicates that the corresponding actual argument in a subroutine reference is an alternate return specifier for the subroutine.

entity: Generic term that refers to any language or program element, such as a program unit, a procedure, a variable, or an array. The term syntactic entity or syntactic item refers to individual elements that make up statements and expressions, such as statement labels, keywords, symbolic names, constants, operators, and special characters.

executable program: Program units that consists of a main program and any number, including zero, of subprograms and external procedures. An executable program cannot have more than one main program.

executable statement: Any statement that specifies some processing action, such as a GOTO or RETURN statement.

external procedure: Fortran-386 subroutine or external function specified outside of the program unit that calls or references it. External procedures can be written in another language for use in a Fortran-386 program.

function subprogram: Executable procedure that can be referenced in an expression. A function subprogram returns a value to the expression that references it. There are three categories of functions: intrinsic, statement, and external.

initial line: First line of a Fortran-386 statement. If the statement exceeds the initial line, you can use up to nineteen continuation lines.

initially defined: Definition status of an entity. An entity is initially defined if it is assigned a value in a DATA statement.

keyword: Sequence of letters that has a special purpose in Fortran-386. Keywords identify Fortran-386 statements, intrinsic functions, or statement separators. Examples are DIMENSION, CONTINUE, ABS, SQRT, THEN, and TO.

list: Nonempty sequence of syntactic entities separated by commas. The entities in the list are called list items.

main program: Program unit that receives control from the operating system to begin execution of a Fortran-386 program. Main programs can execute isolated from any other program unit or can call and reference subprograms during execution. However, you cannot reference the main program from a subprogram. There can be only one main program in an executable Fortran-386 program.

nonexecutable statement: Statements that classify and define program units, specify entry points in subprograms, specify editing information, and specify initial values and execution characteristics for data.

numeric storage unit: Amount of storage required to hold an integer, real, or logical numeric value. A double precision or complex numeric value uses two numeric storage units in a storage sequence. The storage unit establishes a means of referring to data storage without implying a specific storage technology.

procedure, procedure subprogram: Subroutine and function subprograms. There are three categories of function subprograms: intrinsic functions, statement functions, and external functions. The term external procedure refers to subroutines and external functions only. You can write external procedures in another programming language for use in a Fortran-386 program.

program unit: Sequence of Fortran-386 statements and optional comment lines. A program unit is either a main program or subprogram.

reference: Applies to syntactic entities and function procedures. To reference a syntactic entity means to use the name of an entity in a statement that requires the value of that particular entity for execution within the program context. To reference a function procedure means to use the name of a function in an expression or statement that requires that particular function operation for execution within the program context.

scope: Extent to which a given symbolic name or statement label can affect a program. For example, a statement label has the scope of a program unit. A statement label in one program unit does not affect any other program unit.

sequence: Set of elements ordered by a one-to-one correspondence with the numbers 1, 2, 3, through n. The number of elements in a sequence is the number n. An empty sequence contains no elements.

statement: Sequence of syntactic items. Except for assignment and statement function statements, all statements begin with a keyword. Statements are written in one or more lines.

statement label: Sequence of one to five digits. One of the digits must be nonzero. Statement labels are used to identify specific statements in a program.

subprogram: Program unit that is called or referenced from either the main program or another subprogram. There are two classes of subprograms: block data and procedures.

subroutine subprogram: External procedure. An external procedure is specified outside of the program unit that calls it. A subroutine is referenced with the CALL statement.

substring: Contiguous sequence of characters that represents a portion of a character datum. A character datum is a string of one or more characters. Substrings are identified with a substring name. The substring name is used to define and reference the substring.

symbolic name: A sequence of alphanumeric characters. The first character in a symbolic name must be a letter. Symbolic names can identify constants, variables, arrays, main programs, subprograms, common blocks, and dummy procedures. Character sequences that serve within the program context as format edit descriptors and keywords are not considered symbolic names.

syntactic item, syntactic entity: Generic terms that refer to individual elements that make up statements and expressions, such as statement labels, keywords, symbolic names, constants, operators, and special characters. See **entity** for more information.

variable: Entity that can assume a changing value during program execution through implicit or explicit redefinition. A variable has both a name and a type.

CHAPTER 19

Interfacing FORTRAN and C

This chapter describes the mechanism for interfacing C and FORTRAN routines within the same program.

19.1. Calling a C routine from Fortran

The Fortran compiler appends a "_" to every external name (function, subroutine and common), while the C compiler does not. To translate a Fortran external name into a C external name all dollar signs ($) are deleted and an underscore (_) is appended. Therefore, Fortran can only call a C routine whose name ends in "_".

All Fortran arguments are passed by reference. Therefore, each parameter in the called C routine is a pointer of the appropriate type, as follows:

Fortran Passes	C Receives
REAL*4	float *
REAL*8	double *
INTEGER*4	long *
INTEGER*2	short *
INTEGER*1	char *
LOGICAL*4	long *
LOGICAL*2	short *
LOGICAL*1	char *
COMPLEX	struct complex {float realpart, imagpart;} *
COMPLEX*16	struct dcomplex {double realpart, imagpart;} *
CHARACTER	char *

In the case of a passing a CHARACTER argument, Fortran not only passes a pointer to the "char" variable, but also passes the length of the CHARACTER variable, as an "int" (NOT as an "int *") at the end of the argument list. Fortran CHARACTER strings constants are null terminated.

If the C routine being called from Fortran is a function, then the return types correspond as follows: an "int" C function must be declared either as INTEGER and LOGICAL in the calling Fortran routine. A "float" or "double" C function must be declared as DOUBLE PRECISION in the calling Fortran routine. Since C usually promotes "float" return values to "double", REAL return values usually cannot be returned from C. COMPLEX, DOUBLE COMPLEX, and CHARACTER are returned by passing the address of where the return value is to be stored as an extra first parameter to the C function. The length of a CHARACTER return value is passed as an extra second "int" parameter to the C function.

The alternate return statement or Fortran, "RETURN e", is equivalent to "return(e)" in C. The Fortran caller does a computed GOTO statement on the return value to implement the alternate return.

19.2. Calling a Fortran routine from C

The Fortran compiler appends a "_" to every external name, while the C compiler does not. Therefore, a C routine can only call a Fortran routine by referring to it with an explicit "_" on the end of the name. The same naming convention is used for a C routine to reference a Fortran named COMMON block.

All Fortran parameters are passed by reference. Therefore, the corresponding argument in the C call must be a pointer of the appropriate type. For example, to pass a scalar variable X from C to Fortran, pass &X.

C Passes	Fortran Receives
float *	REAL*4
double *	REAL*8
long *	INTEGER*4
short *	INTEGER*2
char *	INTEGER*1
long *	LOGICAL*4
short *	LOGICAL*2
char *	LOGICAL*1
struct complex {float re, im;} *	COMPLEX*8
struct dcomplex {double re, im;} *	COMPLEX*16
char *	CHARACTER

In the case of a passing a CHARACTER argument, C must not only pass a pointer to the "char" variable, but must also pass the length of the "char" variable, as an "int" (NOT as an "int *") at the end of the argument list.

If the Fortran routine being called from C is a FUNCTION, then the return types correspond as follows: an INTEGER or LOGICAL Fortran FUNCTION must be declared as 'int" in the calling C routine. A DOUBLE PRECISION Fortran function must be declared as "double" in the calling C routine. Since C usually promotes "float" return values to "double", a REAL return value may not be accessible in C. COMPLEX, DOUBLE COMPLEX, and CHARACTER are returned from the called Fortran routine by passing the address of where the return value is to be stored as an extra first parameter to the C function. The length of a CHARACTER return value is passed as an extra second "int" parameter to the C function.

The alternate return statement or Fortran, "RETURN e", has no equivalent in C.

CHAPTER 20

Fortran Runtime Library

The Green Hills Fortran-386 Runtime Library is supplied with the Fortran-386 compiler. It is supplied as either an object code library or as source code, depending on the environment. The Fortran-386 Runtime Library requires the functions available in a standard C runtime library. On UNIX systems, the standard C library should be loaded along with the Fortran-386 runtime library (this will normally be done by default). For non-UNIX systems which do not already have a C library, Green Hills supplies the C-386 Runtime Library.

The use of these libraries requires no licensing except the standard Fortran-386 compiler license. Under this license, unlimited distribution of programs incorporating the library code is permitted without further charge. However, distribution of the source code or object code of the library is not permitted.

The Fortran-386 Runtime Library includes all of the functions required by the ANSI IEEE standard plus access to many additional functions in the C Library.

20.1. UNIX Fortran Runtime Library

If you are running under the UNIX operating system you may be using the AT&T or Berkeley UNIX Fortran runtime libraries. Fortran-386 can operate with the standard UNIX Fortran runtime libraries as well as the Fortran-386 Runtime Library. The choice of which runtime libraries to use is made by the compiler software system integrator. Refer to your Host Documentation for more information about runtime libraries.

CHAPTER 21

80386 Target

21.1. Introduction

This chapter describes the 80386 target environment for Fortran-386. There are currently two target environments: UNIX System V, and MS-DOS or cross development using Phar Lap tools.

21.2. 80386 Characteristics

The 80386 memory is byte addressed with 32 bit addresses. Bytes are ordered with the least significant byte of a multiple byte value stored at the lowest address, as on the DEC VAX, opposite of the IBM/370. Bits are numbered with bit zero as the least significant bit.

Floating point is IEEE 754 format (32 and 64 bits), least significant byte at the lowest address. The Intel 80287/80387 is supported by default; the Weitek 1167 coprocessor is supported with the -X143 switch.

Character encoding is ASCII.

The stack is always four byte aligned.

Data Type	Size	Alignment
INTEGER (default)	32	32
INTEGER (-i2)	16	16
INTEGER*1	8	8
INTEGER*2	16	16
INTEGER*4	32	32
LOGICAL	32	32
LOGICAL*1	8	8
LOGICAL*2	16	16
LOGICAL*4	32	32
REAL	32	32
REAL*4	32	32
REAL*8	64	32(64 with -X302)
DOUBLE PRECISION	64	32(64 with -X302)
CHARACTER*1	8	8
CHARACTER*n	8*n	8*n
COMPLEX	64	32(64 with -X302)
COMPLEX*8	64	32(64 with -X302)
COMPLEX*16	128	32(64 with -X302)
DOUBLE COMPLEX	128	32(64 with -X302)

21.3. UNIX System V Target Environment

The output of the compiler is UNIX System V 80386 Assembler Language by default. Phar Lap code, for use under MS-DOS or other environments, can be generated with the -X214 switch.

The -g option generates Common Object File Format (COFF) ".def" symbolic debug pseudo-ops in the System V assembler language output. The assembler and linker understand and process the symbolic debug entries in the object files. The "sdb" symbolic debugger can be used with Fortran-386 output.

21.3.1. Calling Conventions

Arguments are evaluated from right to left. Each argument is pushed on the stack after it is evaluated.

The stack is always aligned on a four byte boundary.

The address of each argument is pushed on the stack as a 32 bit pointer.

Scalar values are returned in EAX. When the size of the return value is specified as less than 32 bits only the required number of bits can be depended on in EAX.

Floating point values are returned in the top stack entry (FP0) in the 80287/80387 environment. They are returned in f2 or f2/f3 in the Weitek 1167 environment.

A call to a subroutine or function uses the "call" instruction. The return from a subroutine or function uses the "ret" instruction.

A subroutine or function is assumed to destroy EAX, ECX, and EDX. The Weitek registers FP1 through FP23 are also assumed to be destroyed. The condition codes are undefined at the return of a subroutine or function. All other registers are saved and restored by a subroutine or function if they are used.

The compiler has two options for generating the local frame for a subroutine or function. By default, no frame pointer is saved, all accesses to parameters on the stack are done with the ESP relative addressing mode, and EBP is used as a scratch register.

If the -g or -ga compile time options is specified, or if there is local data space, the subroutine or function will save the old frame pointer and set up a new one on entry and restore the old frame pointer on exit. Accesses to parameters or local stack storage will be made with EBP relative addressing modes.

Following the return of the subroutine or function, any arguments pushed on the stack are removed. Parameters are removed from the stack by an add immediate to the stack pointer.

CHAPTER 22

Optimization

22.1. Introduction

Fortran-386 does many optimizations which are not available in other Fortran compilers. These optimizations can reduce the size of a program by up to 30% and increase its speed by up to four times. Fortran-386 performs all of the optimizations performed by most other Fortran compilers. It folds constant expressions, converts multiplications into shifts and divides into multiplications when it is advantageous, and eliminates redundant jumps and unreachable code.

22.2. General Optimizations

General Optimizations always make programs smaller and faster.

22.2.1. Register Allocation by Coloring

Register allocation by coloring is used to keep the most commonly used values in registers at all times. The entire subroutine or function is examined to determine which local variables and parameters are used most frequently. The most commonly used variables and parameters are allocated to machine registers. No memory is allocated for them. This optimization has a significant savings in execution speed and it saves a great deal of space. Referencing a variable in a register usually takes one-third of the space and one-third of the time of referencing a variable in memory.

The register allocator uses data flow analysis to find the lifetime of each variable. Using this information, it increases the number of variables which are stored in registers by using the same register for several variables in the same subroutine or function. Two variables may be allocated to the same register if there is no place in the program in which both variables hold a value that will be used later on. Most of the time, all local variables are kept in registers and none in memory.

By default, any integer, real, or logical local variable of the main program or a subroutine or function is a candidate for allocation to a register, unless it is passed to a subroutine or function.

By default, all register candidates will be allocated to the available registers so as to give either the fastest or densest code possible (as controlled by the -OL compile time option). Most Fortran compilers will allocate all local variables in memory. Fortran-386 will allocate as many local variables to registers as it can. Fortran-386 is much better than most Fortran compilers in its register allocation.

In the following example, Fortran-386 allocates i and j to the same register because their lifetimes do not overlap.

```
      subroutine proc
      integer i,j
      i = 1
10    call f
      i = i + 1
      if (i .lt. 10) goto 10
      j = 1
20    call g
      j = j + 1
      if (j .lt. 10) goto 20
      end
```

```
                Fortran-386                          UNIX f77
                -----------                          --------
proc:                                       proc:
        pushl   %esi                                pushl   %ebp
                                                    movl    %esp,%ebp
                                                    subl    $8,%esp
        movl    $1,%esi                             movl    $1,-4(%ebp)
.L7:                                        .L17:
        call    f
        incl    %esi
                                                    movl    $10,%eax
        cmpl    $10,%esi                            cmpl    %eax,-4(%ebp)
        jl      .L7                                 jl      .L18
        movl    $1,%esi                             movl    $1,-8(%ebp)
                                                    jmp     .L22
                                            .L18:
                                                    call    f
                                                    incl    -4(%ebp)
                                                    jmp     .L17
.L4:                                        .L23:
        call    g                                   call    g
        incl    %esi                                incl    -8(%ebp)
                                            .L22:
                                                    movl    $10,%eax
        cmpl    $10,%esi                            cmpl    %eax,-8(%ebp)
        jl      .L4                                 jl      .L23
        popl    %esi                                leave
        ret                                         ret

/i      %esi    local
/j      %esi    local
        ---------------                             ---------------
        35 bytes                                    62 bytes
```

The savings by Fortran-386 can be summarized as:

Put i and j in %esi	15 bytes
Improve enter/exit code	2 byte
Use cmpl $10	6 bytes
Rotate loop	4 bytes

The improvement by the Fortran-386 optimizer can be summarized as:

Put i and j in r25	2 memory references per iteration
Rotate Loop	1 instruction per iteration

22.2.1.1. Fortran Local Variables

Many Fortran programmers have made use of the (undocumented and non-standard) assumption that local variables in a subroutine or function will retain their values between calls to the subroutine or function. This was true of the IBM Fortran compilers and has become a de facto standard. Any implementation of Fortran which does not retain the values of local variables between calls will not be able to run many important Fortran programs. Therefore, most implementations of Fortran allocate all local variables in memory. The code generated by these compilers (such as UNIX f77) is very bad because all variables are in memory. Even simple counter and index variables must be loaded from memory every time they are used.

Fortran-386 makes a very important optimization. Since it knows the complete data flow within a subroutine or function, it checks each Fortran local variable to see if the variable is ever referenced before it is stored into. If so, it is using the value from a previous call to the subroutine or function, and the variable must be allocated in memory. However, most variables are initialized before they are referenced, and these variables are treated as temporary variables and are candidates for allocation to registers. In Fortran-386 most local variables are allocated in registers instead of memory.

22.2.2. Memory Allocation

Fortran-386 allocates variables based on their size, frequency of use, and other factors. Variables which are never used are usually not allocated. Variables are usually sorted to allocate the smaller and more frequently used variables first, and the larger and less frequently used variables later. This allows the use of small address offsets to access commonly used variables. If the compiler allocated some very large variable first, small address offsets might not be able to access variables allocated after it. By putting the smallest and most frequently used variables first, the compiler makes the greatest possible use of small address offsets. Some variables which other compilers would allocate in memory are allocated in registers as explained in the section "Register Allocation by Coloring".

22.2.3. Entry and Exit Code Optimization

Most compilers use a frame pointer register in each subroutine or function. The frame pointer is used to access local variables, to point up the call stack to allow stack traces to be printed during debugging, and to unwind the stack for an exception mechanism. The frame pointer is valuable but it is usually not necessary. By default, Fortran-386 does not set up a frame pointer in each subroutine or function. Fortran-386 will generate a frame pointer if the code is the same size or smaller with a frame pointer, but otherwise it will not create a frame pointer and it will access all local variables by using the stack pointer instead.

If it is necessary to have a frame pointer in every subroutine or function the "-ga" compile time option can be specified on the command line. This compile time option will guarantee that there will always be a frame pointer, but it will increase the size of the program.

If a subroutine or function is very short (a common occurrence in structured programming), the entry and exit code may take a large fraction of the space and execu-

tion time of the subroutine or function. If all of the parameters and local variables of a subroutine or function are allocated in registers (usually for a subroutine or function of 20 lines or less), the compiler can often eliminate the subroutine entry and exit code entirely. This optimization generates code much like the best assembly language implementation.

See the example under Register Allocation by Coloring for improvements to the entry and exit code.

22.2.4. Static Address Elimination

A valuable optimization performed by Fortran-386 is to maintain frequently used static addresses in registers. Since static addresses are 4 bytes long, if a static address is used just twice in a subroutine or function, it is faster and smaller to load the address into a register just once at the beginning of the subroutine or function and always use "register indirect" addressing to access it. In this way, most static references are reduced to one-third of the space and less execution time.

```
subroutine p
call f(%val(1))
call f(%val(2))
call f(%val(3))
call f(%val(4))
end
```

Fortran-386

```
p:      pushl   %esi
        movl    $f,%esi
        pushl   $1
        call    *%esi
        popl    %ecx
        pushl   $2
        call    *%esi
        popl    %ecx
        pushl   $3
        call    *%esi
        popl    %ecx
        pushl   $4
        call    *%esi
        popl    %ecx
        popl    %esi
        ret
        ----------
        28 bytes
```

UNIX f77

```
p:      pushl   %ebp
        movl    %esp,%ebp
        pushl   $1
        call    f
        popl    %ecx
        pushl   $2
        call    f
        popl    %ecx
        pushl   $3
        call    f
        popl    %ecx
        pushl   $4
        call    f
        popl    %ecx
        leave
        ret
        ----------
        37 bytes
```

The savings by Fortran-386 can be summarized as:

Static Address Elimination	7 bytes
Simplified entry code	2 bytes

22.2.5. Register Coalescing

Register Coalescing organizes the computation of expressions to ensure that values end up in the registers where they will be needed. This eliminates shuffling the values in registers to get them set up as needed. Most microprocessor compilers will copy the arguments of a computation into scratch registers; do the computation in the scratch registers; then copy the result to the destination. Fortran-386 will use the destination register in the computation so as to save unnecessary copies of the source registers into scratch registers.

For example the Fortran-386 compiler will compile the statement "i = i*100+ j" as follows (i is in %edi and j is in %esi):

Fortran-386		UNIX f77	
imull	$100,%edi,%edi	imull	$100,%edi,%eax
addl	%esi,%edx	addl	%esi,%eax
		movl	%eax,%edi

22.2.6. Loop Rotation

In Fortran, the DO loop specifies the loop termination conditions at the top of the loop. Therefore, many Fortran compilers generate a termination test at the top of the loop and an unconditional branch from the bottom of the loop to the top of the loop. The loop will execute two branch instructions on each iteration of the loop.

A better way to generate code for loops is to place the test at the bottom of the loop. This is called "Loop Rotation". If it can be determined at compile time that the loop will always execute at least once then the loop is entered from the top. If it cannot be determined that the loop will be executed at least once, then an unconditional branch to the termination test is placed before the loop entry. With the test at the bottom only one branch is executed on each iteration of the loop.

22.2.7. Peephole Optimizations

Peephole optimizations are local improvements to the code which are certain to be correct without further analysis of the surrounding code. An example would be two machine instructions where the first moves the contents of register A to register B, and the second instruction moves the contents of register B to register A. If the program code never branches to the second instruction (i.e. both instructions are always executed together), the second instruction can be safely eliminated.

All of the peephole optimizations which have been implemented are safe for device driver code. Should there be any reason to suppress these optimizations, it can be done with the -X9 compile time option.

22.3. Loop Optimizations

Programs which execute for long periods of time execute millions or billions of instructions. Since most programs consist of tens or hundreds of thousands of instructions, some instructions must be executed many times. To increase the speed of a program it is necessary to identify which instructions are executed the most often and concentrate the optimizations in these areas. Computer languages have two main constructs for repeating the execution of instructions: loops and subroutines. By making specific optimizations for each of these constructs it is possible to significantly improve the performance of most programs.

The loop optimizer is selected by the -OL compile time option. This compile time options informs Fortran-386 that most computation is performed in inner loops. When this compile time option is specified, Fortran-386 assigns most of the machines resources, registers in particular, to uses in the innermost loop. This can result in significant performance increases in programs which do most of their computation in loops.

The loop optimizer draws resources away from other useful optimizations. If -OL is specified for a program in which very little computation is done in inner loops, most of the machine's resources will be misdirected in attempting to optimize infrequently executed loops. This can result in decreasing the total performance of the program. The -OL compile time option should only be used on modules for which the programmer is certain most processing occurs in loops.

22.3.1. Loop Invariant Analysis

"Loop Invariant Analysis" is used to speed up loops. Each loop is examined for expressions and address calculations which do not change in the loop. These computations are moved out of the loop and the value is stored into a register. This optimization is particularly valuable for removing array subscripts from a loop when the subscript is a variable or expression which is not modified in the loop. In a small loop, all invariant expressions will be accessed with "register mode" and all invariant addresses will be accessed with "register indirect modes." This optimization usually eliminates all computations of invariant expressions and addresses in loops.

22.3.2. Strength Reduction

Strength reduction is found only in the most advanced compilers. It applies to loops which have an index variable which is incremented by a constant on each iteration of the loop (such as a DO loop). When a loop index variable is used as the subscript for an array, most compilers will multiply the loop index by the size of the array elements and add this offset to the base of the array. Each such reference will typically require at least three instructions. After the application of strength reduction, outside of the loop, a register is loaded with the address of the array element to be accessed on the first iteration of the loop. The array access is replaced by an indirect register addressing mode. On each iteration, the element size is added to the register so that it contains the address of the element to be accessed on the next iteration of the loop. This optimization results in a four to twenty times speed improvement.

Strength reduction also reduces multiplication of the loop index by a loop invariant value to addition of a constant to a register.

22.3.2.1. Fortran Applications

Strength reduction and loop invariant analysis are particularly important to Fortran programmers. Repetitive array indexing in DO loops is very common in Fortran application programs. Since most mainframe compilers do strength reduction, many Fortran programs have been written to depend on it. Fortran programmers take it for granted that an ordinary dot product or matrix operation will generate strength reduced code. If a Fortran compiler does not do strength reduction many popular application programs will run many times slower than expected based on a simple performance comparison of two machines. This is especially true of some Fortran applications programs being moved from mainframes or VAX/VMS to 386 UNIX systems. While the apparent performance ratio of the hardware might be 3 to 1, often the performance of Fortran programs is 10 to 1.

The inner loop of a typical numerical application, a matrix multiplication, is shown below.

```
      SUBROUTINE MATMUL(A,B,C)
      REAL A(100,100),B(100,100),C(100,100)
      DO 10 I = 1,100
      DO 10 J = 1,100
      A(I,J) = 0
      DO 10 K = 1,100
10    A(I,J) = A(I,J) + B(I,K) * C(K,J)
      END
```

Matrix Multiply inner loop by Fortran-386 (Weitek 1167 floating point)

```
.L5:
        wloadl  (%ecx),%fp4
        wmull   (%edx),%fp4
        wloadl  (%esi),%fp30
        waddl   %fp4,%fp30
        wstorl  %fp30,(%esi)
        addl    $4,%ecx
        addl    $400,%edx
        decl    %edi
        jne     .L5
```

CHAPTER 23

Porting Programs to Fortran-386

Some programs which appear to compile and operate correctly when compiled with other Fortran compilers, may not operate correctly when compiled with Fortran-386. The Fortran Language specifications define legal programs in such a way that legal programs will always work with all Fortran compilers, including Fortran-386. The problem is that many programmers make illegal assumptions about the machine or compiler that they are using. This chapter discusses many illegal assumptions which can cause programs to fail when compiled with Fortran-386.

23.1. Compatibility with other Green Hills Compilers

All Green Hills Languages use the same calling conventions for all subroutines, routines, procedures, and functions. Therefore, code from other Green Hills Languages can be freely used with in your Fortran-386 program.

The implementation of each Green Hills Fortran Compiler is the same for each Green Hills Target. Therefore, legal programs written in Fortran-386 can be moved to any other Green Hills Fortran Compiler.

Fortran-386 can be obtained on any Green Hills Host. It is exactly the same on every Host. Therefore, program development can be done on more than one Host, and moving your development to a new Host system is easy.

23.2. Word Size Problems

Some machines are byte addressable. That is, they have addresses which refer to 8 bit bytes. They typically have operations which operate on 8, 16, 32, 64 and 128 bit quantities. Other machines are word addressable. That is, they have addresses which refer to words of a standard size varying from 16 to 64 bits. They typically have operations which operate on multiples of the word size.

If two different machines have different word sizes or if one is word addressable and the other is byte addressable, a program which operates on one machine may not operate on the other machine for several reasons. The word size affects the range of numbers implemented by the INTEGER data type. The word size also affects the precision and range of the REAL and DOUBLE PRECISION data types.

The most common word size problems are (often undetected) integer overflows and floating point underflows, overflows, and loss of precision. The layout of bit aligned data structures will vary with the word size, so overlaying structures in memory makes programs difficult to port to another compile. Doing address arithmetic in integer variables is often not portable.

23.3. Byte Order Problems

Since the success of the IBM/360, byte machines have been more popular than word machines. The advantage of byte machines is their efficient processing of character data. The general acceptance of byte machines has led to easier program portability between machines.

There is, however, one major portability problem between byte machines. The first successful byte machine, the IBM/360, placed the most significant byte of a multiple byte integer value at the lowest address. Many byte machines such as the MC68000 and Z8000 have followed the IBM convention. The second successful byte machine, the PDP-11, placed the least significant byte of a multiple byte integer value at the lowest address. Intellectual decendents of the PDP-11, such as the VAX, 8086/88/286/386, NS32000, and Clipper have followed the DEC convention. These two groups seem to be so well entrenched that no agreement on byte ordering is possible.

Between machines with different byte ordering, programs which overlay characters and integers in memory or which use character pointers to integer variables and vice versa are often not portable. and as type "char" in another, may not work.

23.4. Alignment Requirements

Fortran-386 always aligns multiple byte data items on appropriate address multiples so that all accesses will be legal and efficient. The optimal alignment is the largest alignment required by any data type for optimal access. It is typically the word size or the external bus width. The alignment conventions for Fortran-386 are defined in the 80386 Target chapter. It is possible for the compiler to guarantee that there will be no illegal or inefficient references if the programmer follows simple rules.

The compiler always aligns parameters and local variables within the stack at an optimal offset from the beginning of the frame. The compiler always rounds up the size of the frame to a boundary of the largest optimal alignment of any data type on the 80386. If the stack pointer is initially aligned to this boundary, and the program involves no explicit manipulation of the stack pointer, all stack references will be optimal.

All variables within the global frame are allocated at an optimal offset from the base of the global frame. If the assembler and/or linker allocates the global frame with the maximum optimal alignment of the 80386, all global data references will be optimal.

Variables within a frame are optimally packed together in memory. When a data type has an alignment requirement, the least possible unused space is left between the end of the previous item and the next item so that the next item can be optimally aligned.

In satisfying different alignment requirements, complex data types may be allocated differently on different machines. This will lead to the usual problems with programs which rely on memory overlays. It will also lead to problems with programs which make implicit assumptions about the size and offset of objects.

23.5. Character Set Dependencies

Not all computer systems use the same characters. All computer systems recognize letters, digits, and the standard punctuation characters. But there is considerable variation among the less commonly used characters. Therefore programs which use the less common characters may not be portable.

Fortran-386 uses the ASCII character set and the ASCII collating sequence. Some implementations of Fortran use a different collating sequence such as EBCDIC.

Programs which manipulate character data, especially string sorting algorithms may be dependent on a particular character collating sequence. The collating sequence is the order in which characters are defined by the implementation. If one character appears before a second character in the collating sequence, then the first character will

be "less than" the second character when they are compared. In the ASCII collating sequence, the lower-case letters "a" to "z" appear as the contiguous values 97 to 122. In other collating sequences the lower-case letters are not contiguous.

To make character and string sorting programs portable, care must be taken to avoid depence on the character collating sequence. If a program is designed to operate with a collating sequence other that ASCII it may be necessary to modify string and character comparison code to operate with ASCII.

You can use the intrinsic functions LGE, LGT, LLE, and LLT, to improve the portability of character-entity comparisons.

The dollar sign "$" used in Fortran subroutine and function names are ignored.

23.6. Floating Point Range and Accuracy

One of the most variable aspects of different machines is floating point. The range, precision, accuracy and base vary widely. This can lead to many portability problems which can only be addressed numerically.

23.7. Operating System Dependencies

Programs which access operating system resources, such as files, by their system names are often not portable. The file and I/O device naming conventions vary greatly among computer systems. In order to write portable programs it is necessary to minimize the use of explicit file names in the program. It is best if these names can be input to the program when the program is run.

If a program contains explicit file names it may be necessary to change the names to names acceptable to the target system in order to get them to operate with Fortran-386. Refer to your target operating system documentation for a description of legal file names for your environment.

23.8. Assembly Language Interfaces

Programs which use embedded assembly code or interface to external assembly will require all of the assembly code to be redone when the program is transported to a new machine.

23.9. Evaluation Order

The Fortran Language specification does not fully specify the order in which the various components of an expression or statement must be evaluated, but it disallows computations whose results depend on which permitted evaluation order is used. Many illegal programs have gone undetected for years because they have only been compiled with one compiler. Since the Fortran-386 evaluation order is not identical to the evaluation order of other Fortran compilers, some of these illegal programs which operate as expected with another Fortran compiler may not operate the same way when compiled with Fortran-386.

Some implementations of the Fortran Language evaluate the arguments to a subroutine or function from right to left, others from left to right. See the 80386 Target chapter for details of the Fortran-386 calling conventions.

Expressions with side effects, such as subroutine or function calls may be executed in a different order by Fortran-386 and other Fortran compilers. When a variable is modified as a side effect of an expression and its value is also used at another point in the expression, it is not defined whether the value used at each point in the expression is the value before or after modification. Potentially, different values for the same variable could be used at different places in the expression depending on the order the

compiler chose for evaluation.

23.10. Illegal Assumptions about Compiler Optimizations

Some programs illegally depend on the exact code that some particular compiler generates. Such programs are particularly difficult to port to an advanced optimizing compiler, such as Fortran-386, because the optimizer makes major changes in the code in order to make the program smaller and/or faster. Described below are some of the most common illegal assumptions about code generation that some programs depend on to work. Please familiarize yourself with the optimizations described in the "Optimization" chapter before reading this section.

23.10.1. Implied register usage

Some programs rely on the exact register allocation scheme used by the compiler. Such programs are completely illegal, and will never transport without modification.

23.10.2. Dummy Assigned Goto Label List

The list in an assigned goto statement must be correct. The optimizer makes use of the list to determine data flow. Programmers have been known to put in dummy lists because many compilers ignore the list; this could cause erroneous results. If no list is present the optimizer assumes that the goto could branch to any label which appears in any ASSIGN statement in the program unit containing the assigned goto. The assigned goto may not be used to jump from one program unit to another program unit. Jumps from assembly code to assigned labels will only work if extreme care is taken.

23.10.3. Memory Allocation Assumptions

Memory is allocated by Fortran-386 in a different way than by f77 and other Fortran compilers. Therefore, there can be problems in porting programs which illegally depend on the memory allocation peculiarities of other compilers. Some programs depend on the compiler allocating variables in memory in the order that they are declared. Fortran-386 will not necessarily allocate variables in the order of declaration. Some programs depend on knowing that the compiler will allocate all variables even if they are not used. Fortran-386 may not allocate unused variables. Some programs depend on knowing that certain variables will be allocated in memory. Fortran-386 will allocate certain variables to registers that f77 and other compilers would always allocate to memory. Programs compiled with Fortran-386 must not make assumptions regarding the order of allocation of variables in memory (except where the Fortran language standard specifies it).

23.10.4. -OM Restrictions

The -OM and -OLM compile time options should only be used in algorithmic programs, that is, programs in which memory cannot change except under control of the compiler. The -OM and -OLM compile time options tell the compiler that memory locations do not change asynchronously with respect to the running program. In particular, if the compiler reads or writes some memory location, three instructions later it can assume that the same value is still in the memory location.

This simple assumption is not true for many parts of operating systems, device drivers, memory mapped I/O locations, shared memory environments, multiple process environments, interrupt driven routines, and when UNIX style signals are enabled. The -OM and -OLM compile time options MUST NOT be used in these cases.

23.10.5. Problems with Source Level Debuggers

Once a variable is allocated to a register it will always reside in that register. However, since other variables may share the register, the register may not always contain the value of that variable. This may cause a source level debugger to give incorrect results. If you ask for the value of a variable at a point at which that variable is about to be assigned into, the compiler may have temporarily allocated that register for some other purpose. Always check results just after they are assigned, or when the current value is going to be used later. Near the end of a subroutine or function most of the local variables are no longer going to be used, so the chance that the register has been reallocated is much higher.

23.11. Problems with Compiler Memory Size

Fortran-386 is an advanced optimizing compiler. It is much better that the current generation of "optimizing" microprocessor Fortran compilers. In accordance with its greater capability it requires more memory. Fortran-386 requires 300 Kbytes just for the program. It is designed to work best when it has at 1 Mbyte or more of memory available. It will run in less memory but with some degradation of performance or capability.

The compiler's primary use of memory is for the program, static data structures, global declarations, parse trees, and generated machine code. Global declarations consist of the global constant, type, variable, and subroutine or function declarations. This is a major use of memory when large numbers of declarations are included into a compilation. Even unused global declarations must be stored throughout the compilation. If memory size problems exist try to reduce the size of the include files by including just the declarations that are needed.

Fortran-386 is a one pass compiler. That is, it reads the source program only once. Each subroutine or function is converted into a parse tree as it is read. When the end of the subroutine or function is reached the optimizer is called with the parse tree as input. The optimizer modifies the parse tree and then passes it on to the 80386 code generator. The code generator produces an internal representation of the 80386 machine code to be output for the subroutine or function. Another optimization phase is then called to modify this machine code. Finally the optimized machine code for the subroutine or function is output. After the machine code is output, the memory being used for the parse tree and machine code is reclaimed for use in compiling the next subroutine or function.

The memory usage for parse trees and machine code is determined by the size of the largest subroutine or function in the program. If memory size problems exist, turn off the optimizer and reduce the size of the largest subroutine or function. Simple subroutine or functions of less than 100 lines should not cause memory size problems. Procedures which are more than 1000 lines or contain very complex statements can require more than a megabyte of memory to compile.

COMPLEX multiply, add and subtract are expanded in line for maximum speed. Each COMPLEX multiply generates at least four floating point multiplies and two floating point adds, and uses two temporary variables. Routines containing in excess of one hundred COMPLEX operations may require more than a megabyte of memory to compile.

23.12. Additional Undetected Errors

Many common errors can slip by the compiler and produce erroneous programs. Listed below are some of the problems to watch out for.

It is illegal (according to the Fortran-77 Standard) to pass some intrinsics as arguments to other procedures. Fortran-386 will not detect this error and an undefined reference will occur at link time.

No check is made for recursive statement functions.

Error messages follow the style of the UNIX f77 compiler. There is no indication of the position within the line of an error.

CHAPTER 24

Compile Time Options

This chapter describes the command line switches and compile time configuration options that may be used with the Fortran-386 compiler. The compiler has been preconfigured to easily generate programs for execution in the following four environments:

- Standalone program running on a system resource manager.
- Host program running on a system resource manager.
- Node program using the 80387 floating point coprocessor.
- Node program using the SX scalar floating point option.

You may want to modify the default configuration options if you want to use the compiler for an environment other than these or if you have unusual requirements. Modifications to the default configuration options may produce undefined and/or undesirable results, as Intel Scientific Computers has tested the compiler only for the environments listed above. The default configurations are changed by enabling and/or disabling one or more of the configuration options.

The compiler generates 80387 floating point coprocessor code for system resource manager programs. If your system contains both the VX vector option and SX scalar option, you can generate your node programs for the SX environment to take advantage of both options at the same time.

Use the following command line switches to generate executable code for the four supported environments.

Standalone system resource manager program: (no switches)
System resource manager host program: -host
Node program using 80387: -node
Node program using SX option: -node -sx

These and the other command line switches are explained below.

Command Line Switches

-both

Links in single and double precision routines for vectorization.(Only for VX Option)

-B

Implement a general 'debug enviroment' with all default compile optimizations disabled and code generation for iSC's debugger DECON enabled.

-c

Do not produce executable files. Produce only object files. For each source language file specified, compile the source language file into object code output. Put the object code output into a file whose name ends in ".o".

-C

Turn on runtime checking of subranges and array bounds. The code will execute slower when this option is used.

-double

Links in double precision routines for vectorization.(Only for VX Option)

-Dname

For file names ending with the ".F" extension, define "name" to the C preprocessor with the value 1. This is equivalent to putting "#define name 1" at the beginning of the source file.

-Dname=string

For file names ending with the ".F" extension, define "name" to the C preprocessor with the value "string". This is equivalent to putting "#define name string" at the beginning of the source file.

-F

Do not produce assembly, object, or executable files. Produce only Fortran source files. For each source language file name with the ".F" extension, preprocess the source language file with the C preprocessor and leave the preprocessor output on a file whose name ends in ".f".

-host

Link in the libraries required for host programs running on a system resource manager.

-g

Generate source level symbolic debug information and a frame pointer for stack traces.

-ga

Generate a frame pointer for stack traces. The default compiler setting is to optimize the program to the point that stack traces may become impossible, making program debugging difficult. When debugging a program this option should be used. This option does not imply "-g".

-i2

Make the type INTEGER be INTEGER*2.

-Istring

For file names ending with the ".F" extension, search for include files which are not specified with absolute path names (do not start with "/") in the directory "string", before searching the standard include file directories. Multiple -I options can be specified, and directories will be searched in the order that the -I options are listed. This option applies only to files that are included by the C preprocessor, using a '#include "filename" ' directive. This option does not affect the Fortran INCLUDE statement.

-lname

Include the specified library at link time. The normal UNIX library search sequence is used.

-o filename

Place the executable file output into the file named "filename". If this option is not specified the executable file will be named "a.out". This option is ignored if "-c", "-S", or "-F" is present.

-O

The -O option activates the Green Hills optimizers which are safe for use on all programs, except for the loop optimizer.

-OM

This option is eqivalent to -O except that it also allows the optimizer to assume that memory locations do not change except by explicit stores. That is, the optimizer is guaranteed that no memory locations are I/O device registers that can be changed by external hardware and no memory locations are being shared with other processes which can change them asynchronously with respect to the current process. This compile time option must be used with extreme caution (or not at all) in device drivers, operating systems, shared memory environments, and when interrupts (or UNIX signals) are present.

-OL

Optimize the program to be as fast as possible even if it is necessary to make the program bigger. In particular, most of the available resources are allocated to optimizations of the innermost loops. The -OL compile time option will perform optimizations which may make the program faster but larger. It is counter-productive to specify -OL on code which contains no loops or that is rarely executed as it will make the whole program larger but no faster. After experimenting with a program it is possible to discover which modules benefit from -OL and which ones do not.

-OLM

This option is equivalent to -OL and -OM.

-OML

This option is equivalent to -OLM.

-onetrip

Execute at least one iteration of every DO loop. The default is that if the lower bound of a DO loop is greater that the upper bound, then no iterations are made (this is the ANSI Fortran-77 standard). This was unspecified under the ANSI Fortran-66 standard and some important implementations (especially IBM) chose to always execute the loop at least once. The use of this option makes the compiler incompatible with the ANSI Fortran-77 standard, but it may be necessary to use it to get certain old Fortran-66 programs to operate correctly.

-node

Link in libraries required for node programs.

-p

Generate calls for execution profiling. The UNIX profiler must be available; a profiler is not part of the library provided by Green Hills.

-R

Put all data in the text section.

-S

Do not produce object files or executable files. Produce only assembly language files. For each source language file specified, compile the source language file into assembly language output. Put the assembly language output into a file whose name ends in ".s".

-single

Links in single precision routines for vectorization.(Only for VX Option)

-sx

Generate code for the SX scalar option and substitute the appropriate libraries at link time.

-u

Make the default data type for undeclared variables be "undefined", as if "IMPLICIT UNDEFINED(A-Z)" was placed at the beginning of the file.

-U

Do not convert upper case names in Fortran to lower case. By default, Fortran is not case sensitive and all Fortran names which are externally visible are in the object file in lower case. Use this option to gain access to names defined in C in upper case. Using this option makes the compiler incompatible with the ANSI Fortran-77 standard.

-v

Have the compiler driver print out the program name and command line arguments as it runs each subprocess.

-vec

Links in the appropriate Veclib library.(Only for VX Option)

-vecdbg

Links in the appropriate debug version of Veclib library.(Only for VX Option)

-vms

During compilation VAX/VMS conventions will be used where they conflict with f77(same as -X181). Linking will use a VAX/VMS version of libf.a.

-vx

Directs the compiler to align double precision variables on a 8 byte boundaries(same as -X302 -Z85). Directs the linker to allocate data to VX memory and link in appropriate Veclib routines.(Only for VX Option, see iPSC®/2 VX User's Guide for essential details.)

-w

Suppress warning diagnostics.

The remainder of the chapter lists the compiler configuration options. To see how the options are set when you select one of the four supported environments (no switch, -host, -node, or -sx), add the -v switch to the command line when compiling a program. The general form of these options is as follows:

-Xnnn

Turn on configuration option number nnn, where nnn is an unsigned integer constant.

-Znnn

Turn off configuration option number nnn, where nnn is an unsigned integer constant. This is the reverse of the X option, and is useful if you want to turn off an option that is enabled by default.

Configuration Options

-X9 Disable local (peephole) optimizer.

-X13 Suppress code generation. An empty output file will be created.

-X18 Do not allocate programmer-defined local variables to a register unless they are declared register.

-X32 Display the names of files as they are opened. This is useful for finding out why the compiler cannot find an include file.

-X37 Emit a warning when dead code is eliminated.

-X39 Do not move frequently used procedure and data addresses to registers.

-X58 Do not put an underscore in front of the names of global variables and procedures.

-X68 This makes characters unsigned as they are in some implementations of Fortran. The default is signed characters as in UNIX f77.

-X71 Use the Green Hills single precision math libraries.

-X74 The target system is UNIX System V.

-X77 Turn off compile time checking of FORMAT statements. Use this option if your runtime library supports FORMAT statement features that the Fortran compiler doesn't know about.

-X79 Pad hollerith constants on the right with blanks. The default is compatible with UNIX f77: only the first byte of the hollerith is significant and the constant is zero padded on the left.

-X80 Turn off the branch tail merging optimization. This can speed up compilation in some cases.

-X82 Compile lines starting with "x", "X", "d", "D". The default is to treat them as comments. Used for enabling debugging statements.

-X85 (UNIX Target only) Generate ".lcomm" (BSD 4.2) or ".bss" (UNIX System V) for zero initialized statics. The default is to allocate initialized data.

-X89 Pack structures with no space between members (even if it makes them impossible to access!)

-X105 Allow redefinition of #define symbols to the preprocessor.

-X143 Generate Weitek floating point code instead of 80387.

-X151 Do not allow dollar signs in names. The default allows dollars signs for VMS compatibility.

-X164 Do not stop in the event of a code generator abort or "Internal Compiler Error" error message. This option is occasionally useful for determining the cause of a compiler failure. If this option is used, the compiler may crash or otherwise terminate abnormally.

-X168 Do not move invariant floating point expressions out of loops.

-X181 VMS Fortran compatibility.

-X202 Don't output "." before assembler directives.

-X211 Suppress optimizations that generate inline code for external calls.

-X214 Eliminate branches by copying code at branch destination in place of branch.

-X219 Suppress elimination of jumps to jumps.

-X226 This is the compiler switch to indicate an 80387 rather than an 80287.

-X229 Use a non-standard convention whereby 4 of the 80X87 registers are used as registers.

-X230 Suppress common subexpression elimination and value propogation, except for trivial cases.

-X237 Apply associative rules in common subexpression elimination.

-X252 Pure Intel asm386.

-X255 Print a brief description of -X switches on the terminal.

-X264 Suppress phase that removes useless sign and zero extend instructions.

-X265 Suppress register database phase.

-X266 Repeat the peephole phase until the code fails to improve.

-X268 Suppress the lastload phase.

-X271 Suppress the phase that merges and removes excess move instructions.

-X272 Suppress the realvar code in database phase.

-X278 Don't merge index calculation into load instruction.

-X285 Suppress block merge phase.

-X297 Always load NARGS before a call.

-X302 Align double precision variables on 8-byte boundaries, subject to certain limitations.

-X308 Perform tail recursion optimizations.

-X311 Don't make multiple copies of blocks in merge blocks phase.

-X312 Suppress recognition of ?: operators as absolute value and min/max.

-X313 Generate code for multiple initializations of common variables. Only works with Avalon assemblers.

-X325 Return large items in a reentrant fashion, rather than following old UNIX customs.

-X326 Allocate gettarget temporaries as a round robin instead of a stack.

-X329 Generate "stabd" pseudo-ops for line numbers instead of stabn line numbers.

-X331 Allocate unused variables if symbolic debugging enabled (-g).

-X332 Try to avoid generating floating divides if a multiply can be used instead.

-X333 Suppress passing of front end information to the peephole optimizer and instruction scheduler.

-X337 Suppress most multiple initialization error messages. Some program units initialize the same variable more than once. This is illegal, but it is supported by some compilers. If a variable is initialized more than once with the same value, this switch may make the program work. If the variable is initialized with different values, this switch may mask the detection of a serious program error.

-X344 Suppress adrconst optimizations. Do not try to undo ineffective allocation of constants to temporaries.

-X353 Perform common subexpression analysis twice. Rarely useful.

-X356 Make sure DO loop variable is always up to date (never cache it).

-X370 Output line numbers in the assembly file.

CHAPTER 25

The iPSC®/2 Common Debug Enviroment

25.1. -B Compiler Flag

The purpose of the iPSC/2 debugger, DECON is to find user bugs, not compiler bugs. To this end, a common debug compiler can be invoked, using the -B switch, to provide the 'safest' compilation from the available switches. In particular, all those optimizations that might be unsafe are turned off.

Compiling objects for use with DECON can be done by:

f77 -c -B host.f or cc -c -B host.c

Listing -B last in the sequence of compiler flags gives it precedence over the previously listed and invoked flags. Optimization flags -O, -OLM ... should be removed for debugging user code.

The -B flag sets the following flags:

-g
: Generate source level symbolic debug information and a frame pointer for stack traces.

-ga
: Generate a frame pointer for stack traces. The default compiler setting is to optimize the program to the point that stack traces may become impossible, making program debugging difficult.

-X9
: Disable local (peephole) optimizer

-X18
: Do not allocate programmer-defined variables to registers

-X39
: Do not move frequently used procedures and data addresses to registers

-X168
: Do not move invariant floating point expressions out of loops

-X211
: Suppress optimization that generates inline code for external calls

-X219

Suppress eliminations of jumps to jumps

-X230

Suppress common subexpression elimination and value propagation

-X307

Turn off instruction reordering

-X356

Make sure the DO loop variable is always uptodate(never cache it)

Programs that work correctly on this debug compiler, but not in more optimized versions of the compiler should be reported to iSC as compiler bugs.