intel®

FORTRAN-86 USER'S GUIDE

Order Number: 121570-003

FORTRAN-86
USER'S GUIDE

Order Number: 121570-003

BITBUS	i$_m$	iSBC	Plug-A-Bubble
COMMputer	iMMX	iSBX	PROMPT
CREDIT	Insite	iSDM	Promware
Data Pipeline	int$_e$l	iSXM	QueX
Genius	int$_e$lBOS	KEPROM	QUEST
i	Intelevision	Library Manager	Ripplemode
$\hat{i}$	int$_e$ligent Identifier	MCS	RMX/80
I^2ICE	int$_e$ligent Programming	Megachassis	RUPI
ICE	Intellec	MICROMAINFRAME	Seamless
iCS	Intellink	MULTIBUS	SOLO
iDBP	iOSP	MULTICHANNEL	SYSTEM 2000
iDIS	iPDS	MULTIMODULE	UPI
iLBX	iRMX		

A1342 / 385 / 3K / DD / KH

SOFTWARE

REV.	REVISION HISTORY	DATE	APPD.
-001	Original issue.	8/81	
-002	Revised to support Version 1.5 of the Fortran-86 Compiler	7/82	
-003	Revised to support COMPLEX data and C language procedure calls.	12/84	D.N.

PREFACE

This manual provides language, compiler, and run-time information specific to Fortran-86.

It is designed to support new users as well as those already familiar with Fortran.

This manual contains fifteen chapters and nine appendixes:

- Chapter 1, "Overview," describes Fortran-86, the compiler, the run-time support, the operating environment, and program development.
- Chapter 2, "Program Structure," describes the parts of a Fortran program and their required order.
- Chapter 3, "Language Elements," describes Fortran's lexical structure.
- Chapter 4, "Program Delimitors and Comments," describes comment lines, procedure headings, and their use.
- Chapter 5, "Data and Specification Statements," describes data types, arrays, arguments, and specification statements.
- Chapter 6, "Subprograms," describes subroutines, external functions, intrinsic functions, statement functions, and BLOCK DATA subprograms.
- Chapter 7, "Expressions," describes the Fortran expressions and their use.
- Chapter 8, "Executable Statements," describes assignment statements, control statements, and data-transfer statements.
- Chapter 9, "Input and Output," describes the file-handling and I/O statements.
- Chapter 10, "Examples," describes sample Fortran-86 programs.
- Chapter 11, "Compiler Controls," describes the Fortran-86 controls with an indication of use.
- Chapter 12, "Compiler Operation," describes compiler invocation, input files, output files, overlay files, and compiler messages.
- Chapter 13, "Compiler Output," describes the listing output and the object module output.
- Chapter 14, "Linking, Relocating, and Executing Programs," describes how to run programs.
- Chapter 15, "Errors and Warnings," describes language and run-time errors and recovery.
- Appendix A, "Differences Between Fortran-86 and Other Versions of Fortran," lists how Fortran-86 differs from ANSI Fortran 77 and from Fortran-80.
- Appendix B, "Processor-Dependent Features of Fortran-86," lists the features dependent on the 8086, 8087, and 8088 processors.
- Appendix C, "Compiler Capacity," lists the upper limits imposed by the compiler or its environment.
- Apppendix D, "Language Summary," lists the Fortran statements, symbols, intrinsic functions and subroutines.
- Appendix E, "Character Set and Collating Sequence," gives the ASCII character set.
- Appendix F, "Hollerith Data Type," describes the Hollerith data type.
- Appendix G, "Run-Time Data Representations," describes the internal representations of Fortran data types.

- Appendix H, "Linking to Subprograms Written in Other Languages," describes parameter passing, returned values from functions, and sharing of data between Fortran-86 and other iAPX 86, 88 family languages.
- Appendix I, "Run-Time Interface," describes error handlers, interrupt processing, and the logical record interface for users not executing their programs on the Series-III.
- Appendix J, "Additional Information for Series III and Series IV Operating System Users," provides examples and information specific to the Series III and Series IV Operating Systems.
- Appendix K, "Additional Information for iRMX™ 86 Operating System Users," provides information and examples specific to the iRMX 86-based system.

"Microsystems 80" Nomenclature

Over the last several years, the increase in microcomputer system and software complexity has given birth to a new family of microprocessor products oriented towards solving these increasingly complex problems. This new generation of microprocessors is both powerful and flexible and includes many processor enhancements such as numeric floating point extensions, I/O processors, and operating system functionality in silicon.

As Intel's product line has grown and evolved, its microprocessor product numbering system has become inadequate to name VLSI solutions involving the above enhancements.

In order to accommodate these new VLSI systems, we've allowed the 8086 family name to evolve into a more comprehensive numbering scheme, while still including the basis of the previous 8086 nomenclature.

We've adopted the following prefixes to provide differentiation and consistency among our Microsystem 80 related product lines:

iAPX —	Processor Series
iRMX —	Operating Systems
iSBC —	Single Board Computers
iSBX —	MULTIMODULE Boards

Concentrating on the iAPX Series, two Processor Families are defined:

iAPX 86 —	8086 CPU based system
iAPX 88 —	8088 CPU based system

With additional suffix information, configuration options within each iAPX system can be identified, for example:

iAPX 86/10	CPU Alone (8086)
iAPX 86/11	CPU + IOP (8086 + 8089)
iAPX 88/20	CPU + Math Extension (8088 + 8087)
iAPX 88/21	CPU + Math Extension + IOP (8088 + 8087 + 8089)

This nomenclature is intended as an addition to, rather than a replacement for, Intel's current part numbers. These new series level descriptions are used to describe the functional capabilities provided by specific configurations of the processors in the

8086 Family. The hardware used to implement each functional configuration is still described by referring to the parts involved (as is the case for the majority of the 8086 information described in this manual).

This improved nomenclature provides a more meaningful view of system capability and performance within the evolving Microsystem 80 architecture.

Related Publications

For information on the Intellec Series-III Microcomputer Development System, see the following manuals:

- *A Guide to the Intellec Series III Microcomputer Development System*, 121632
- *Intellec Series III Microcomputer Development System Product Overview*, 121575
- *Intellec Series III Microcomputer Development System Console Operating Instructions*, 121609
- *Intellec Series III Microcomputer Development System Programmer's Reference Manual*, 121618
- *ISIS-II CREDIT CRT-Based Text Editor User's Guide*, 9800902
- *AEDIT Text Editor User's Guide*, 121756

For information on the iRMX 86 operating system, see the following manuals:

- *iRMX 86 Human Interface Reference Manual*, 9803202
- *iRMX 86 Nucleus Reference Manual*, 9803122
- *EDIT Reference Manual*, 143587

For information on auxiliary products, see the following manuals:

- *8086/8087/8088 Macro Assembly Language Reference Manual for 8086-Based Development Systems*, 121627
- *8086/8087/8088 Macro Assembler Operating Instructions for 8086-Based Development Systems*, 121628
- *iAPX 86,88 Family Utilities User's Guide for 8086-Based Development Systems*, 121616
- *Pascal-86 User's Guide*, 121539
- *PL/M-86 User's Guide*, 121662
- *ICE-86 In-Circuit Emulator Operating Instructions for ISIS-II Users*, 9800714
- *ICE-88 In-Circuit Emulator Operating Instructions for ISIS-II Users*, 9800949
- *The 8086 Family User's Manual*, 9800722
- *The 8086 Family User's Manual Numerics Supplement*, 121586
- *User's Guide for the iSBC 957B iAPX 86,88 Interface and Execution Package*, 143979

Notational Conventions

`UPPERCASE`	Characters shown in uppercase must be entered in the order shown. You may enter the characters in uppercase or lowercase.

italic	Italic indicates a meta symbol that may be replaced with an item that fulfills the rules for that symbol. The actual symbol may be any of the following:
directory-name	Is that portion of a *pathname* that acts as a file locator by identifying the device and/or directory containing the *filename*.
filename	Is a valid name for the part of a *pathname* that names a file.
pathname	Is a valid designation for a file; in its entirety, it consists of a *directory* and a *filename*.
pathname1, pathname2, ...	Are generic labels placed on sample listings where one or more user-specified pathnames would actually be printed.
system-id	Is a generic label placed on sample listings where an operating system-dependent name would actually be printed.
V*x.y*	Is a generic label placed on sample listings where the version number of the product that produced the listing would actually be printed.
[ ]	Brackets indicate optional arguments or parameters.
{ }	One and only one of the enclosed entries must be selected unless the field is also surrounded by brackets, in which case it is optional.
{ }...	At least one of the enclosed items must be selected unless the field is also surrounded by brackets, in which case it is optional. The items may be used in any order unless otherwise noted.
|	The vertical bar separates options within brackets [ ] or braces { }.
...	Ellipses indicate that the preceding argument or parameter may be repeated.
[,...]	The preceding item may be repeated, but each repetition must be separated by a comma.
punctuation	Punctuation other than ellipses, braces, and brackets must be entered as shown. For example, the punctuation shown in the following command must be entered: `SUBMIT PLM86(PROGA,SRC,'9 SEPT 81')`
`input lines`	In interactive examples, user input lines are printed in white on black to differentiate them from system output.
`<cr>`	Indicates a carriage return.

CONTENTS

FIGURES

TABLES

CHAPTER 1 OVERVIEW

This chapter introduces Fortran-86 and explains how it fits into the process of developing software for an iAPX 86 or iAPX 88 application system.

Fortran-86 is a high-level language designed for programming the 8086 and 8088 microprocessors. Fortran-86 is a superset of the Fortran 77 subset defined by the American National Standards Institute (ANSI), and is compatible with Fortran-80. Fortran-86 also includes additional features that facilitate the programmer's task of developing software.

The Fortran-86 compiler translates Fortran-86 source programs into relocatable 8086 object modules, which you can then link to other such modules, coded in Fortran or in other 8086/8088 languages. The compiler provides listing output, error messages, and a number of compiler controls that aid in developing and debugging programs.

The compiler provides a set of relocatable object libraries to be linked with your own code; these provide complete run-time support for input/output, arithmetic functions, and in-line code execution by using the optional 8087 Numeric Data Processor. After linking your own modules together with these Intel-supplied library modules, you can locate your final linked program in order to run it on an Intel development system, or in RAM, PROM, or ROM on 8086-based or 8088-based custom hardware.

To perform the steps following compilation, you use the standard 8086 Family software development utilities—LINK86, LOC86, LIB86, and OH86. Next, debug your programs by using the resident monitor program, the ICE-86 In-Circuit Emulator, or an Intel debugger (such as PSCOPE). For firmware systems, you may then use the Universal Prom Programmer (iUP) with its Universal Prom Mapper (UPM) software to burn your programs into PROM.

1.1 The Compiler and Run-Time System

The following sections describe the advantages offered by the Fortran-86 compiler controls and run-time libraries.

1.1.1 Compiler Features

The Fortran-86 compiler includes a number of features that make software programming and debugging easier. Compiler controls allow you to specify the form and content of your source code, object code, and output listing.

Controls are provided to copy (INCLUDE) source code from other files in addition to the main source file, to output debug information in the object file for use by LINK86 and the ICE-86 emulator, and to specify interrupt procedures. The compiler also provides an optional symbol listing control, as well as controls that format the output listing according to your specifications.

1.1.2 Run-Time Support Libraries

The run-time support libraries, provided in relocatable object code form to be linked to your compiled object program, allow you to run your program in a number of hardware environments. You simply choose the run-time libraries that match the hardware/software configuration you are using.

These libraries provide all I/O support, including device drivers, needed to run programs on the system. You may also choose to have floating-point arithmetic operations performed either by the floating-point software routines on an 8086 processor, or using the on-chip capabilities of an 8087 Numeric Data Processor (for higher performance). Both options include all required arithmetic and interface software in the run-time libraries.

In addition, the modular structure of the I/O libraries allows you to substitute your own device drivers for non-standard I/O devices. For instructions, see Appendix I.

1.2 Hardware and Software Environments

The following sections describe the appropriate environments for developing and executing Fortran-86 programs.

1.2.1 Program Development Environment

To run the compiler, you must have certain hardware and software. The system dependent appendixes (Appendix J for Series III and Appendix K for iRMX-86) list these requirements.

A system with a printer is also recommended for producing hard-copy output listings, but may be separate from the one used to compile programs.

To link and relocate programs after you have compiled them, and to prepare them for loading (or PROM programming) and execution, you need the following software:

- LINK86
- LOC86
- LIB86
- OH86

Instructions for using these utility programs are given in the *iAPX 86, 88 Family Utilities User's Guide*, order number 121616.

Depending on your development environment and your final run-time environment, you also may choose to use the following hardware and software:

- The ICE-86 In-Circuit Emulator
- The SDK-86 System Design Kit, optionally with the SDK-C86 Software and Cable Interface
- The iSBC 957B iAPX 86,88 Interface and Execution Package
- The Universal PROM Programmer (iUP) with the Universal PROM Mapper (UPM) software

1.2.2 Run-Time Environment

Your compiled, linked, and located program code may run in any of the following environments:

- A Series-III development system under the Series-III resident operating system
- An iSBC system with an iAPX 86,88 CPU board and the iRMX 86 operating system
- A custom-designed 8086- or 8088-based microcomputer system

In an environment without Intel operating system support, you will need to write your own I/O drivers (as described in Appendix I) and provide a software interface to the operating system.

The amount of memory required at run time will depend on the size of your application program.

You may increase the speed of floating-point arithmetic operations and reduce code size in your programs by including an 8087 Numeric Data Processor in your system. Detailed specifications are provided in the *iAPX 86, 88 User's Manual*, 210201.

1.3 Compiler Installation

The Fortran-86 software package includes this manual (the *Fortran-86 User's Guide*), the *Fortran-86 Pocket Reference*, supplementary literature including a customer letter and Software Problem Report forms, and two single-density diskettes and one double-density program diskette. Series IV users also receive the Fortran compiler on one 5¼-inch, double-sided, double-density diskette. The contents of the disks are listed in Appendixes J and K.

When your compile-time environment is configured, copy the compiler and run-time library files from the product diskette to the single-density diskettes or the double-density diskette, or to the hard disk if you are using one on your system. Copying for diskette systems is only necessary for backing-up the files or for storing the compiler and libraries on other diskettes.

1.4 The Program Development Process

The Fortran-86 compiler and run-time libraries are part of the integrated set of tools that make up the total 8086 development solution for your microcomputer system. Figure 1-1 shows how you use these tools to develop programs using Fortran-86. The shaded boxes represent Intel products.

The steps in the software development process are as follows:

1. Define the problem completely.
2. Outline the proposed solution in terms of hardware and software.
3. Design the software for your system. This important step may consist of several sub-steps, including breaking the task into modules, choosing the programming language, and selecting the appropriate algorithms. You may decide to code some modules in languages other than Fortran, such as 8086/8087/8088 Macro Assembly Language, PL/M-86, or Pascal-86.
4. Code your programs and enter them on the system by using a CRT-based text editor, such as CREDIT or AEDIT.
5. Use the Fortran-86 compiler to translate your Fortran program code.
6. Use the text editor to correct any compile-time errors reported by error messages, and retranslate the program.
7. Using LINK86 (and LOC86 if needed), link the resulting relocatable object module to the necessary run-time libraries supplied with Fortran-86 and the operating system. The use of LINK86 and LOC86 depends on your application; for detailed instructions, see the *iAPX 86, 88 Family Utilities User's Guide*, order number 121616.

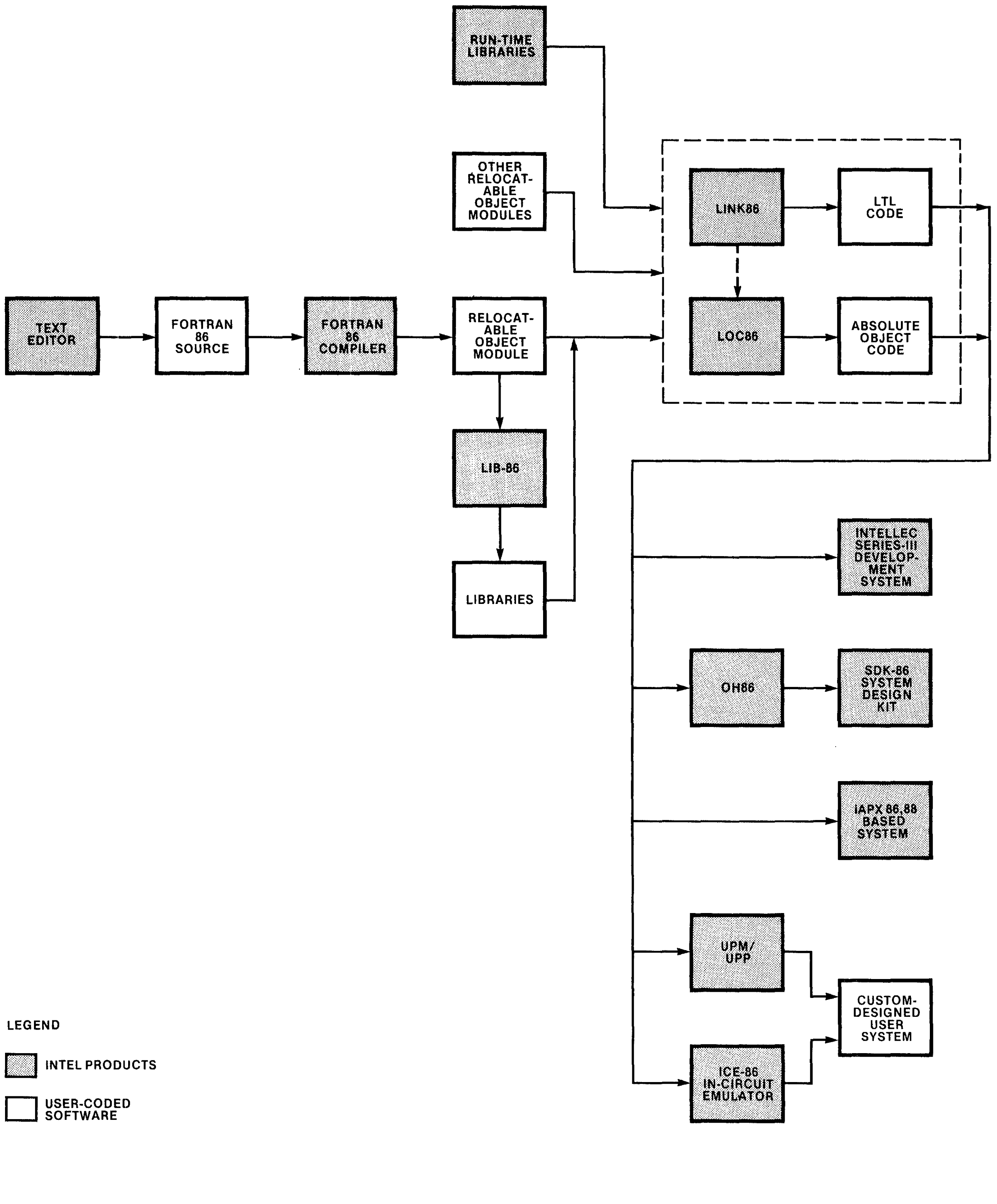


Figure 1-1. Fortran-86 Program Development Process 121570-1

8. You now can run and debug your programs with the aid of Fortran's run-time error messages. Your execution vehicle for debugging can be any of the following: a Series-III system with its resident monitor and an ICE-86 or ICE-88 In-Circuit Emulator (optional) or PSCOPE, an iRMX-based system, RAM on an SDK-86 System Design Kit, or RAM on an iAPX 86,88 Single Board Computer with a resident monitor.
9. Translate and debug your other system modules, including those coded in other languages. Once you have performed the desired amount of testing on each module, you can link the modules and optionally locate them by using LINK86 and LOC86.
10. Test and debug your software in the selected debug environment.
11. Produce a final debugged object module and transfer it to the run-time environment. This step is dependent on the environment and on the tools you are using.
    - When the environment is a development system, use the execution command to load and run your program.
    - When the environment is RAM on an SDK-86 kit or an iAPX 86,88 Single Board Computer system, use OH86 to obtain a hexadecimal object code file. Then, if you are developing your programs on a Series-III, use an appropriate tool for downloading them into the execution board (the ICE-86 In-Circuit Emulator, the SDK-C86 Software and Cable Interface, or the iSBC 957B Interface and Execution Package).
    - When the environment is ROM on an SDK-86, iAPX 86,88 Single Board Computer system, or your own custom-designed hardware, use the Universal PROM Programmer (iUP) with its Universal PROM Mapper (UPM) software to burn your program into PROM.

Note that you can perform hardware and software development in parallel, and that you can take intermediate hardware/software integration steps by using the ICE-86 In-Circuit Emulator.

For instructions on the use of other Intel products discussed in this section, refer to the manuals listed in the preface to this book.

CHAPTER 2 PROGRAM STRUCTURE

2.1 Basic Structure

You can divide a Fortran program into distinct program units. Each unit can be thought of as a sequence of statements and comments. The first statement of a program unit determines whether the compiler will treat the unit as a main program or a subprogram. Although it is optional, a main program usually has a PROGRAM statement as its first statement. A main program may contain any statements except BLOCK DATA, FUNCTION, or SUBROUTINE statements because these are used only to define subprograms. A main program cannot be referenced by a subprogram or by itself. A Fortran program can have only one main program, but it may contain any number of subprograms.

There are three kinds of subprograms: BLOCK DATA, FUNCTION, and SUBROUTINE. A BLOCK DATA subprogram begins with a BLOCK DATA statement, and provides initial values for variables and array elements in named COMMON blocks. A detailed description is in Chapter 6, "Subprograms."

Any executable program is called a procedure. FUNCTION and SUBROUTINE subprograms are external procedures. Either the main program or programs written in other iAPX 86,88 languages can call these procedures. A FUNCTION subprogram begins with a FUNCTION statement and returns a value when referenced. A SUBROUTINE subprogram begins with a SUBROUTINE statement. See Chapter 6 for a complete explanation of FUNCTION and SUBROUTINE subprograms.

2.2 Fortran Statements

In Fortran there are two kinds of statements: executable and nonexecutable. Executable statements do calculations, read or write data from external media, and control program execution. Nonexecutable statements define the characteristics or values of data and define program units. The following list classifies Fortran statements as executable or nonexecutable. You can find complete definitions in the chapters indicated below.

EXECUTABLE STATEMENTS:

- Arithmetic, logical, and character assignment statements (Chapter 8)
- ASSIGN statement (Chapter 8)
- Unconditional, assigned, and computed GOTO statements (Chapter 8)
- Arithmetic and logical IF statements (Chapter 8)
- Block IF, ELSE IF, ELSE, and END IF statements (Chapter 8)
- CONTINUE statement (Chapter 8)
- STOP and PAUSE statements (Chapter 8)
- DO statement (Chapter 8)
- READ, WRITE, and PRINT statements (Chapter 9)
- REWIND, BACKSPACE, ENDFILE, OPEN, and CLOSE statements (Chapter 9)
- CALL and RETURN statements (Chapter 8)
- END statement (Chapter 4)

NONEXECUTABLE STATEMENTS:

- PROGRAM, BLOCK DATA, FUNCTION, and SUBROUTINE statements (Chapter 4)

- DIMENSION, COMMON, EQUIVALENCE, IMPLICIT, EXTERNAL, INTRINSIC, and SAVE statements (Chapter 5)
- INTEGER, REAL, DOUBLE PRECISION, TEMPREAL, COMPLEX, DOUBLE COMPLEX, LOGICAL, and CHARACTER type statements (Chapter 5)
- DATA statement (Chapter 5)
- PARAMETER statement (Chapter 5)
- FORMAT statement (Chapter 9)
- Statement-function statement (Chapter 6)

2.2.1 Statement Order

Fortran program units must follow this standard order:

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

Figure 2-1 summarizes the rules for ordering Fortran statements in a program unit. In this figure, vertical lines separate statement types that can be mixed, and horizontal lines separate those that cannot.

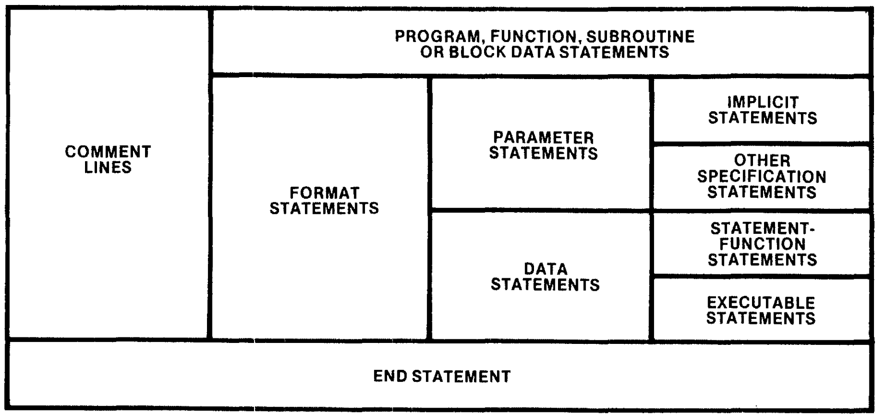


Figure 2-1. Order of Fortran Statements 121570-2

CHAPTER 3
LANGUAGE ELEMENTS

3.1 Basic Alphabet

The character set for Fortran-86 is the set of all uppercase and lowercase letters, the digits 0 through 9, and the following special characters:

	Blank
=	Equal Sign
+	Plus
–	Minus
*	Asterisk
/	Slash
(	Left Parenthesis
)	Right Parenthesis
,	Comma
'	Single Quote
$	Dollar Sign
#	Pound Sign
:	Colon
_	Underscore
%	Percent Sign

Blanks are significant only in character strings. They can be used to improve program readability. For example,

```
A=B*C+(D**2/E)
```

and

```
A  =  B*C  +  (D**2/E)
```

are equivalent statements.

However, blanks are counted in the total number of characters in a Fortran line. Blanks are also significant in character strings and in column six of the standard line format. Blanks have no effect on the total memory space allocated for the object code.

3.2 Statement Elements

The letters, digits and special characters of the Fortran-86 character set form the basic elements of a Fortran statement. These basic elements are constants, symbolic names, statement labels, keywords, and operators. There are no reserved words in Fortran; any combination of the character set is acceptable as long as it complies with certain rules outlined in the next sections.

3.2.1 Constants

A constant is a value that does not change. In Fortran, there are arithmetic, logical, and character constants. Each constant has a data type and a length. See Chapter 5, "Data and Specification Statements," for details on constants.

3.2.2 Symbolic Names

Each variable in Fortran must have a symbolic name. A symbolic name consists of 1 to 6 alphanumeric characters in standard Fortran, and 1 to 31 in Fortran-86. The first character must be a letter or an underscore. For example, the symbolic names,

```
A
a
C3PQ
ZA82B
BEF974
_A8
```

is not correct because the first character is not a letter or an underscore.

The symbolic name,

```
1ACG
```

is not correct because the first character is not a letter.

The compiler does not distinguish between uppercase and lowercase characters.

A symbolic name can be either global or local. Any global symbolic name applies throughout the entire program. The following is a list that catagorizes global symbolic names:

- Main program name
- Subroutine names
- External function names
- BLOCK DATA subprogram names
- Named COMMON names

A local symbolic name can represent different entities in different program units or statement functions. The following is a list of local symbolic names:

- Array names
- Variable names
- Statement-function names
- Intrinsic-function names
- Dummy procedure names

Variables that appear as dummy arguments in a statement function have a scope of that statement only.

3.2.3 Statement Labels

Any statement can be labeled; any statement referenced from elsewhere in the program must be labeled. This label is a 1-5 digit unsigned, nonzero integer constant placed in columns 1-5 of the statement's initial line.

3.2.4 Keywords

Fortran keywords are very important. All but two types of statements begin with a keyword, and the compiler uses the keyword to identify the statement. Most keywords fulfill the requirements of a symbolic name. Since there are no reserved words, however, the compiler distinguishes between keywords and symbolic names by the context.

3.3 Statements and Lines

Each Fortran statement is made up of lines. The first line is the initial line, and each subsequent line is a continuation line. Fortran-86 can have up to 19 continuation lines per statement.

3.3.1 Line Format

Fortran-86 lines must follow a specified order. Figure 3-1 shows this order.

Each line has a maximum of 132 characters. The first 5 positions may contain the statement label. If there is no statement label for a line, or if this line is a continuation line, these positions must be left blank. Position 6 is the continuation field. If this position contains a 0 or a blank, the line is an initial line. If this line contains any other Fortran character, it is a continuation line. The actual statement does not begin until column 7.

You can deviate from the standard Fortran line format by using the FREEFORM control. See Section 11.4.6, "FREEFORM CONTROL" for details.

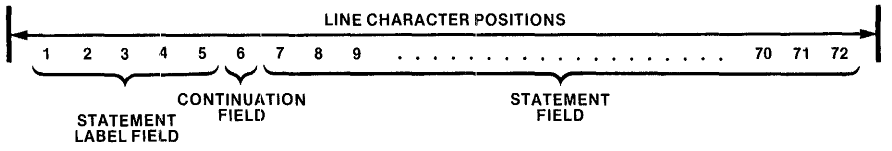


Figure 3-1. Fortran Line Format 121570-3

CHAPTER 4 PROGRAM DELIMITERS AND COMMENTS

4.1 Comments

Comments in Fortran are lines that document the program. Comment lines are useful for describing the intent of the program between the lines. Each comment line must begin with either the letter C or an asterisk (*) in position 1. A completely blank line is treated as a comment. Comment lines can appear anywhere before the END statement, including between an initial line and its continuation lines or between any two continuation lines. Comment lines have no effect on program execution or memory requirements.

4.2 Headings

As described in Chapter 2, "Program Structure," you can divide a Fortran program into a main program and any number of subprograms. Each unit begins with a different statement that defines the unit. The following sections describe these initial statements.

4.2.1 PROGRAM Statement

The PROGRAM statement names the main program. This statement is optional, but if present, it must be the first statement in the main program. Its syntax is as follows:

`PROGRAM` *name*

where

name	is the symbolic *name* of the main program. This *name* applies to the entire executable program and cannot be the same as the *name* of any function, subroutine, BLOCK DATA subprogram, common block, or any local variable within the main program.

4.2.2 FUNCTION Statement

The FUNCTION statement introduces a FUNCTION subprogram, and must be the first statement in the subprogram. Its syntax is as follows:

[*type*] `FUNCTION` *name*`(`[*arg*[`,` *arg*]]`)`

where

type	is one of the specified data types INTEGER, REAL, DOUBLE PRECISION, TEMPREAL, COMPLEX, DOUBLE COMPLEX, LOGICAL, or CHARACTER (see Chapter 5, "Data and Specification Statements").
name	is the symbolic *name* of the subprogram.
arg	is the name of a dummy argument that is either a variable, an array, or a procedure.

The FUNCTION *name* can appear as a variable within the subprogram. This name is defined or redefined each time the program activates the function. The value of this variable at the end of the subprogram is the resulting, or return, value of the function. A function can change the values of its dummy arguments. If there are no dummy arguments, the parentheses still must be present. The uses of FUNCTION subprograms are described in Chapter 6, "Subprograms."

4.2.3 SUBROUTINE Statement

The SUBROUTINE statement introduces a SUBROUTINE subprogram, and must be the first statement in the subroutine. Its syntax is as follows:

```
SUBROUTINE name ([arg[,arg]])
```

where

name	is the symbolic *name* of the subroutine
arg	is a dummy argument that is either a variable, array, or procedure.

A subroutine can change the values of its dummy arguments. If there are no dummy arguments, either form, SUBROUTINE name, or SUBROUTINE name ( ) is acceptable. The uses of subroutines are described in Chapter 6, "Subprograms."

4.2.4 BLOCK DATA Statement

The BLOCK DATA statement introduces a BLOCK DATA subprogram, and must be the first statement in the subprogram. Its syntax is as follows:

```
BLOCK DATA [name]
```

where

name	is the optional *name* of the subprogram.

A BLOCK DATA subprogram initializes global data and contains no executable statements. See Section 6.2, "BLOCK DATA Subprograms."

4.3 END Statement

The END statement indicates the end of a program unit. This unit may be either a main program or a subprogram. Its syntax is as follows:

```
END
```

The END statement must be the last statement in a program unit. When executed in a main program, END terminates the program. When executed in a subprogram, END acts as a RETURN statement and restores control to the main program.

You must enter an END statement only in positions 7 through 132 of an initial line, and the END statement cannot extend to a continuation line. No other statement can have an initial line with the same characteristics as an END statement.

CHAPTER 5 DATA AND SPECIFICATION STATEMENTS

Fortran-86 supports a range of data types. Each data type has a specification statement that indicates the data type for a given argument, and directs the compiler to allocate the appropriate amount of storage. This chapter describes the each of the data types supported by Fortran-86 and provides the corresponding specification statement for each data type.

5.1 Data Types

Fortran-86 supports the following data types: integer, floating-point, logical, and character. Floating-point data types include the following: REAL, DOUBLE PRECISION, COMPLEX, COMPLEX*16, and TEMPREAL. A symbolic name representing a constant, variable, array, or function indicates data type.

You can specify the type of a named constant, variable, array, external function, or statement function with a type statement. In the absence of a specific declaration, the Fortran default typing convention takes effect. According to this convention, the first letter of the name indicates the particular type. A first letter of I, J, K, L, M, or N indicates type INTEGER; any other letter or an underscore indicates type REAL. An IMPLICIT statement can change this convention (see Section 5.2).

Type statements can also specify data length or array dimension information. You cannot specify the type of a name explicitly more than once in a program unit. PROGRAM, SUBROUTINE, and BLOCK DATA names cannot appear in type statements.

In Fortran, there are four levels at which data lengths can be set: compiler default, the STORAGE control, the IMPLICIT statement, and type specification statements. If you do not specifically declare any data lengths, the following compiler defaults are in effect:

INTEGER	2 bytes
LOGICAL	1 byte
REAL	4 bytes
DOUBLE PRECISION	8 bytes
CHARACTER	1 byte
COMPLEX	8 bytes
TEMPREAL	10 bytes

If you use the STORAGE control when compiling your program, (see Section 11.4.19), you can change the default length specification for INTEGER and LOGICAL data only.

If you specify a length in an IMPLICIT statement, this specification overrides both the STORAGE control and the compiler defaults for the given class of names.

If you specify a length in a type statement, it overrides the IMPLICIT statement, STORAGE control, and the compiler default for the given names.

5.1.1 Integer Data

An item of integer data always comprises the exact representation of an integer value. The value can be positive, negative, or zero. An item of integer data requires one one-byte, two-byte, or four-byte numeric storage unit, depending on the default or explicit length specification for the constant, variable, or function. Table 5-1 provides the value ranges for integer data.

5.1.1.1 Integer Constants

The forms of an unnamed integer constant are as follows:

[*sign*] *diglet* [*diglet*]...

or

[*sign*] # *diglet* [*diglet*]...*base*

where

sign	is the optional plus (+) or minus (−) sign.
diglet	is one of the 10 digits (0 through 9) or one of six letters (A through F).
base	is a base specifier that is one of the letters D, B, O, Q, or H.

The base specifier indicates to the compiler what base the integer constant has. The letter D indicates a decimal number, B indicates a binary number, O or Q indicates an octal number, and H indicates a hexadecimal number.

If the base specifier is D, indicating a decimal number, each diglet must be one of the digits 0 through 9. By default, an integer without a base specifier integer is a decimal number.

If the base specifier is B, indicating a binary number, each diglet must be one of the digits 0 or 1.

If the base specifier is either O or Q, indicating an octal number, each diglet must be one of the digits 0 through 7.

If the base specifier is H, indicating a hexadecimal number, each diglet must be one of the digits 0 through 9 and the letters A through F. The first diglet must be one of the digits 0 through 9.

Fortran-86 gives a storage length equivalent to INTEGER*4 to the results of some integer constant expressions. An integer constant in a constant list of a DATA statement is given a length that matches the length of the corresponding data item. Other integer constants will be allocated to the smallest unit capable of holding their value, up to a maximum of four bytes. An exception to this rule occurs when an integer constant is used as an actual argument. In this case, the constant is assigned the default length (see Section 11.4.19, Storage Control).

Table 5-1. Value Ranges of INTEGER Data

Type and Length	Value
INTEGER*1	−128 TO +127
INTEGER*2	−32,768 TO +32,767
INTEGER*4	−2,147,483,648 TO +2,147,483,647

A value that exceeds the range of presentable values for the particular type of data is undefined.

5.1.1.2 INTEGER Type Statement

An INTEGER type statement declares names to be of type INTEGER. Its syntax is as follows:

INTEGER[* *len*]*name* [, *name*]...

where

len	has one of the numbers 1, 2, or 4.
name	is one of the following forms:
	var [* *len*]
	or
	array [(*d*)][* *len*]

where

var	is the name of an integer constant, variable, function, or dummy procedure.
array	is an array name.
array(*d*)	is an array declarator, (see Section 5.4.1, "DIMENSION Statement").
len	is the length in bytes of the integer variable or each integer array element. The value len must be 1, 2, or 4.

The length specification immediately following the keyword INTEGER applies to each item in the statement not having its own length specification. A length specification immediately following an item is for that item only. For an array, the length applies to each array element. If no length is specified, the compiler assigns a length (see Sections 5.2, "IMPLICIT Statement" and 11.4.19, "STORAGE Control").

5.1.2 Floating-Point Data

An item of floating-point data represents a processor approximation to the value of a floating-point number. Floating-point data values can be positive, negative, or zero. The internal representation, the precision, and the range of floating-point values conforms to the floating-point conventions established by the IEEE *Proposed Standard for Binary Floating-Point Arithmetic*, Draft 8.0. For more information on floating-point arithmetic, see the *iAPX 86, 88 User's Manual*, order number 210201.

Fortran-86 supports the following types of floating-point data: REAL, DOUBLE PRECISION, TEMPREAL, COMPLEX, and COMPLEX*16. REAL data is stored in one or two four-byte numeric storage units in a sequence depending on the explicit or implicit length specification. DOUBLE PRECISION data is stored in two four-byte numeric storage units and TEMPREAL data in 10 bytes. COMPLEX data is stored in two or four four-byte storage units. The first half stores the real part and the second half stores the imaginary part.

Note that the internal representation of the REAL*8 data type is the same as that of the DOUBLE PRECISION data type.

5.1.2.1 Floating-Point Constants

The basic form of a floating-point constant is as follows:

[*sign*] *digit*. *digit* [*exponent*]

where

sign	is an optional plus (+) or minus (−) sign.
digit. *digit*	is the integer and fractional part of the constant. Both the integer part and the fractional part are strings of decimal digits. You can omit either of these parts but not both. You can write a floating-point constant with more digits than the processor will use to approximate the value of the constant. The compiler interprets a floating-point constant as a decimal number.

There are three floating-point exponent forms that correspond to the floating-point data types. The syntax is as follows:

letter [*sign*]*digit*

where

letter	is the letter E for REAL exponents, D for DOUBLE PRECISION, and T for TEMPREAL.
sign	is the optional plus (+) or minus (−) sign.
digit	is a decimal integer constant.

The internal representation of the REAL*8 data type is the same as that of DOUBLE PRECISION. Therefore, you must write REAL*8 constants with the D exponent. The compiler will allocate any constant with the E exponent only one four-byte numeric storage unit. The exponential form of COMPLEX data is based on its component elements. A complex constant is represented by an ordered pair of REAL, INTEGER, or DOUBLE PRECISION constants, separated by a comma, enclosed in parenthesis.

Table 5-2 shows the approximate ranges for floating-point data.

5.1.2.2 REAL Type Statement

A REAL type statement declares names to be of type REAL. Its syntax is as follows:

REAL[* *len*]*name*[, *name*]...

Table 5-2. Value Ranges for Floating-Point Data

Type	Value
REAL	I 1.2*10**(−38) I TO I 3.4*10**(38) I
DOUBLE PRECISION	I 3.4*10**(−308) I TO I 1.8*10**(308) I
TEMPREAL	I 3.4*10**(−4932) I TO I 1.2*10**(4932) I
COMPLEX	*See note
COMPLEX*16	*See note

*Each component of a COMPLEX*8 number has the same value range as a REAL number. Each component of a COMPLEX*16 number has the same value range as a DOUBLE PRECISION number.

where

len	is one of the numbers 4 or 8.
name	has the following form:

var [* *len*]

or

array [(*d*)][* *len*]

where

var	is the name of a real constant, variable, function, or dummy procedure.
array	is an array name.
array (*d*)	is an array declarator (see Section 5.4.1, "DIMENSION Statement").
len	is the length in bytes of the real variable or each real array element. The value len must be 4 or 8.

The length specification immediately following the keyword REAL applies to each item in the statement not having its own length specification. A length specification immediately following an item is for that item only. For an array, the length applies to each array element. If no length is specified, the compiler assumes the default length of four bytes, or the default length specified by the IMPLICIT statement (see Section 5.2).

5.1.2.3 DOUBLE PRECISION Type Statement

The DOUBLE PRECISION type statement declares names to be of type DOUBLE PRECISION. Its syntax is as follows:

DOUBLE PRECISION *name* [, *name*]...

where

name	is a constant name, variable name, function name, dummy procedure name, array name, or array declarator (see Section 5.4.1, "DIMENSION Statement"). The compiler assigns a length of two four-byte numeric storage units to each name.

5.1.2.4 TEMPREAL Type Statement

The TEMPREAL type statement declares names to be of type TEMPREAL. Its syntax is as follows:

TEMPREAL *name* [, *name*]...

where

name	is a constant name, variable name, function name, dummy procedure name, array name, or array declarator (see Section 5.4.1, "DIMENSION Statement"). The compiler assigns a length of one 10-byte numeric storage unit to each name.

5.1.2.5 COMPLEX and COMPLEX*16 Type Statement

The COMPLEX and COMPLEX*16 type statement declares names to be of type COMPLEX, and has the following syntax:

COMPLEX[**len*], *name*[**len*], *name*[**len*], ...

where

name	is one of the forms:
	v[**len*]
	or
	a(*d*) [**len*]

where

v	represents a variable name, function name, or dummy procedure.
a	is an array name.
a(*d*)	is an array declarator.
len	is the storage unit length. Len may be 8 or 16 for complex data. Len may also be an integer constant expression that evaluates to one of the above values (8 or 16), enclosed in parenthesis. By default, the compiler assigns two or four four-byte storage units to each name.

5.1.3 Logical Data

Logical data can assume only the values true or false. Logical data may have one, two, or four-byte numeric storage units, depending on the explicit length specification or the implicit length for a LOGICAL variable or function (see Sections 5.2, "IMPLICIT Statement" and 11.4.19, "STORAGE Control"). Note that only the first byte of a two or four-byte data item is actually used.

5.1.3.1 Logical Constants

Table 5-3 shows the form and acceptable values of logical constants.

5.1.3.2 LOGICAL Type Statement

The LOGICAL type statement declares names to be of type LOGICAL. Its syntax is as follows:

LOGICAL[**len*] *name*[, *name*]...

Table 5-3. Value Ranges of LOGICAL Data

Type and Length	Value
LOGICAL*1	.TRUE. or .FALSE.
LOGICAL*2	.TRUE. or .FALSE.
LOGICAL*4	.TRUE. or .FALSE.

where

len	is one of the numbers 1, 2, or 4.
name	has the following form: *var* [* *len*] or *array* [(*d*)][* *len*]

where

var	is the name of a logical constant, variable, function, or dummy procedure.
array	is an array name.
array (*d*)	is an array declarator (see Section 5.4.1, "DIMENSION Statement").
len	is the length in bytes of the logical variable or each logical array element. The value len must be 1, 2, or 4.

The length specification immediately following the keyword LOGICAL applies to each item in the statement not having its own length specification. A length specification immediately following an item applies to that item only. For an array, the length applies to each array element. If no length is specified, the compiler assumes the default length (see Sections 5.2, "IMPLICIT Statement", and 11.4.19, "STORAGE Control").

5.1.4 Character Data

Character data are strings of ASCII characters. Each character in the string has a character position numbered consecutively from left to right beginning with 1. The blank character is valid and significant in character data.

5.1.4.1 Character Constants

A character constant has the following form:

```
'CHARACTERS'
```

The apostrophe (') is not part of the character constant, but must be entered to delineate the constant. Two consecutive apostrophes ('') represent a single apostrophe within the string. For example:

```
''MURPHY''S LAW'
```

The length of a character string is the number of characters in the string. Each pair of consecutive apostrophes counts as one character. The length of a character constant must be greater than zero.

5.1.4.2 CHARACTER Type Statement

A CHARACTER type statement declares names to be of type CHARACTER. Its syntax is as follows:

CHARACTER[* *len*]*name* [, *name*]...

where

len	is any unsigned, non-zero, integer constant expression enclosed in parentheses, or is an asterisk (*) enclosed in parentheses.
name	has one of the following forms:
	var[* *len*]
	or
	array[(*d*)][* *len*]

where

var	is a character variable.
array	is an array name.
array(*d*)	is an array declarator (see Section 5.4.1, DIMENSION Statement).
len	is the number of characters in the character variable, character array element, character constant with a symbolic name, or character function.

The length specification immediately following the keyword CHARACTER applies to each item in the statement not having its own length specification. A length specification immediately following an item applies to that item only. For an array, the length applies to each array element. If no length is specified, the compiler assumes the standard default length for CHARACTER data (one byte). If a length has been specified by an IMPLICIT statement (see Section 5.2), that length will be assigned to the item.

5.1.5 Hollerith Data

Fortran-86 supports Hollerith data types. See Appendix F for details.

5.2 IMPLICIT Statement

The IMPLICIT statement defines the default type and length for symbolic names that begin with the letter or letters specified by IMPLICIT. IMPLICIT overrides the standard Fortran typing convention (see Section 5.1). Any type statement or explicit type specification in a FUNCTION statement can override an IMPLICIT statement.

The syntax of the IMPLICIT statement is as follows:

IMPLICIT *type*(*let*[, *let*]...)[*type*(*let*[, *let*]..)]...

where

type	is one of the Fortran-86 data types:
	INTEGER[* *len*], REAL[* *len*], LOGICAL[* *len*], CHARACTER[* *len*], DOUBLE PRECISION, TEMPREAL, COMPLEX or COMPLEX*16.
let	is a single letter or range of letters in alphabetical order in the form *let-let*. For this range specification an underscore is considered to immediately follow Z.

len	is the length of the item in bytes. The value of *len* must be 1, 2, or 4 for INTEGER or LOGICAL data, 4 or 8 for REAL data, 8 or 16 for COMPLEX data, and is the length of the character string for CHARACTER data. If no length is specified, the compiler assumes the following default lengths: 1 byte for CHARACTER data, 4 bytes for REAL data, and 8 bytes for COMPLEX data. The default lengths for INTEGER and LOGICAL data are specified as described in Section 11.4.19, "STORAGE Control".

The IMPLICIT statement applies only to the program unit in which it appears and must precede all other specification statements in that program unit. A program unit can have more than one IMPLICIT statement, but you can specify a particular letter only once.

5.3 PARAMETER Statement

The PARAMETER statement gives constants symbolic names. Its syntax is as follows:

```
PARAMETER(name = exp [,...])
```

where

name	is a symbolic name.
exp	is a constant expression.

If the name is of type INTEGER, the corresponding expression must be of type INTEGER. If the name is of type REAL, DOUBLE PRECISION, TEMPREAL, COMPLEX, or COMPLEX*16, the corresponding expression must also be a constant of the same type. If the name is of type LOGICAL, then *exp* must be a logical constant. If name is of type CHARACTER, *exp* must be a character constant.

Any symbolic name of a constant that appears as an expression in a PARAMETER statement must have been defined previously in a PARAMETER statement (including the same PARAMETER statement).

If the symbolic name of a constant is not of default implied type or length, you also must specify its type and length in either a type statement or an IMPLICIT statement before it appears in the PARAMETER statement. Subsequent statements, including an IMPLICIT statement, cannot change the type or length.

A symbolic name in a PARAMETER statement may identify only the corresponding constant in that program unit.

5.4 Arrays

An array is a sequence of data elements. You can refer to the sequence as a whole or to individual elements in the sequence.

An array name is the symbolic name of the entire array. An array element name is the symbolic name of one member of the array. An array element name is an array name qualified by one or more subscripts enclosed in parentheses. An array name not qualified by a subscript identifies the entire array with the exception: in an EQUIVALENCE statement or CALL assignment list, the array name with no subscript identifies the first element in the array.

You define an array by assigning a symbolic name to the array and specifying its dimensions. This definition can occur in type statements (Section 5.1), a COMMON statement (Section 5.5.2), or a DIMENSION statement (Section 5.4.1).

5.4.1 DIMENSION Statement

The DIMENSION statement defines an array. Its syntax is as follows:

DIMENSION *array*(*d*)[,*array*(*d*)]...

where

array(*d*)	is an array declarator that has the following form: *array* (*s*[,...])

where

array	is the symbolic name of the array.
s	is a dimension declarator. The number of dimension declarators indicates the number of dimensions in the array. Each dimension declarator indicates the number of elements of that dimension. The maximum number of dimensions is seven.

The form of a dimension declarator is as follows:

[*d1*:]*d2*

where

d1	is the lower dimension bound.
d2	is the upper dimension bound.

Both upper and lower bounds are arithmetic expressions; *d1* may include only integer constants and variables and *d2* may include integer constants, variables, or an asterisk (*). A dimension bound cannot contain a reference to either a function or an array element. The values of upper and lower bounds can be positive, negative, or zero. However, the upper bound must be greater than or equal to the lower bound. If you do not specify a lower bound, the default value is one.

5.4.2 Kinds of Array Declarators

An array declarator (*array*(*d*)) has three types: a constant, adjustable, or assumed-size array declarator.

In a constant array declarator, each of the dimension bounds is an integer constant or integer constant expression. For example:

```
ARRAY(3,3,-3:4)
```

In an adjustable array declarator, one or more of the dimension bounds is an INTEGER variable or expression. For example:

```
ARRAY(3,2:MIDDLE,THIRD:+8)
```

In an assumed-size array declarator, the upper bound of the last dimension is an asterisk (*), as follows:

```
ARRAY(3,MIDDLE,*)
```

You can use an array name as a dummy argument in a FUNCTION or SUBROUTINE subprogram. An actual array declarator must be a constant array declarator whereas dummy array declarators may be constant, adjustable, or assumed-size. Like actual array declarators, dummy declarators are permitted in DIMENSION or type statements, but unlike actual array declarators, they cannot appear in COMMON statements. Each variable name used in a dimension-bound expression must also appear in the subprogram's dummy argument list or in a COMMON block in the subprogram. You can avoid this requirement in the last dimension by using the asterisk (*) feature for the upper bound.

5.4.3 Properties of Arrays

The DIMENSION statement defines the following properties for arrays:

- The type of the array name
- The type of the array elements
- The length of the array elements
- The number of dimensions in the array
- The size of each dimension
- The total number of array elements

The number of dimensions equals the number of dimension declarators in the array declarator. For example, the following array:

```
TABLE(-6:4,4)
```

has two dimensions.

The size of a dimension declarator has the following value:

d2 − d1 + 1

where

d1	is the value of the lower dimension bound.
d2	is the value of the upper bound.

You can compute the size of an array as the product of the sizes of the dimensions specified by the array declarator. For example:

```
ARRAY(3,-1:1,3)
```

has 27 elements. To determine the number of elements in an assumed size array, do the following:

- If the actual argument corresponding to the dummy array is an array name, the size is that of the actual array.
- If the actual argument is an array element name with a subscript value of p in an array of size n, the size of the dummy array is n + 1 − p.

The compiler stores array elements sequentially. For example, in the following sequence:

```
DIMENSION TABLE(3,3)
TABLE(3,1)=2.9
TABLE(2,3)=7.3
```

2.9 is in the third storage location in the block whose low address is TABLE, and 7.3 is in the eighth location, as shown below.

(1,1)(2,1)2.9(1,2)(2,2)(3,2)(1,3)7.3(3,3)

To determine the total number of bytes in an array, multiply the number of elements by the number of bytes occupied by each element.

5.4.4 Referencing Array Elements

You reference an array element by qualifying the array name with subscripts. For example:

array (*s*[, *s*]...)

where

array	is the array name.
s	is the subscript. The number of subscripts must equal the number of dimensions in the array declarator. Each subscript must be an integer expression in the range lower bound < =s < = upper bound. If the upper dimension bound is an asterisk (*), the value of the corresponding subscript must not exceed the effective upper bound of the corresponding actual array. Table 5-4 shows how to calculate which element in the storage sequence of array elements you are referencing.

Table 5-4. Subscript Reference

n	Dimension Declarator	Subscript	Element Referenced
1 2	$([l_1:]u_1)$ $([l_1:]u_1,[l_2:]u_2)$	s_1 (s_1,s_2)	s_1-l_1+1 $(s_1-l_1+1)+$ $(s_2-l_2)*(u_1-l_1+1)$
3	$([l_1:]u_1,[l_2:]u_2,$ $[l_3:]u_3)$	(s_1,s_2,s_3)	$(s_1-l_1+1)+$ $(s_2-l_2)*(u_1-l_1+1)$ $(s_3-l_3)\text{-}(u_1-l_1+1)*(u_2-l_2+1)$
• • •	• • •	• • •	• • •
7	$([_1:]u_1,[l_2:]u_2,$ $...,[l_7:]u_7)$	$(s_1,s_2,...,s_7$	$(s_1-l_1+1)+$ $(s_2-l_2)*(u_1-l_1+1)$ $(s_3-l_3)\text{-}(u_1-l_1+1)*(u_2-l_2+1)'+$ $... +$ $(s_7-l_7)*(u_1-l_1+1)*(u_2-l_2+1)...*(u_6-l_6+1)$

where n = number of dimensions
l = value of lower bound
u = value of upper bound
s = subscript expression

5.5 Character Substrings

A character substring is a contiguous portion of a CHARACTER variable. It is identified by a substring name and can be assigned values and referenced.

5.5.1 Substring Specification

The syntax is as follows:

v ([*e1*] : [*e2*])
a (*s* [, *s*]...) ([*e1*] : [*e2*])

where

v	is a character variable name.
a (*s* [, *s*]...)	is a character array element.
e1 and *e2*	are integer expressions called substring expressions.

The value *e1* specifies the leftmost character position of the substring. The value *e2* specifies the rightmost character position. For example *d*(3:5) specifies the characters in positions three through five of the character variable *d*, and *m*(3,8) (2:4) specifies the characters in positions two through four of the character array element *m*(3,8).

The values of *e1* and *e2* must be such that:

$1 <= e1 <= e2 <= len$

where

len	equals the length of the character variable or array element. If omitted, the default value of *e1* is one and the default value of *e2* is *len*. Both *e1* and *e2* may be omitted: for example, *v* (:) is equal to *v*, and *a*(*s*[,*s*]...) (:) is equal to *a*(*s*[,*s*]...). The length of a character substring is $e2 - e1 + 1$.

5.5.2 Substring Expressions

A substring expression may be any integer expression and it may contain array element references and function references. Evaluation of a function must not alter the value of any variable within the same substring specification.

5.6 Memory Definition

Fortran includes two statements that control the location of memory areas: the EQUIVALENCE and COMMON statements.

5.6.1 EQUIVALENCE Statement

The EQUIVALENCE statement allows items in a program unit to share memory. Entities listed in the EQUIVALENCE statement share the same starting address in memory, even if they are of unequal length. The syntax for an EQUIVALENCE statement follows:

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

where

nlist	is a list of two or more variable names, array names, or array element names. An array name not qualified by a subscript refers to the first element of the array. You cannot use function names or dummy argument names in this list.

Entities listed in an EQUIVALENCE statement may be of different types. However, since the EQUIVALENCE statement implies no type conversion, it is not recommended.

The EQUIVALENCE statement must not cause the same storage item to occur more than once in a memory sequence. It cannot result in the splitting of a memory sequence already defined.

5.6.2 COMMON Statement

The COMMON statement associates memory among the same and different program units, allowing common use of data and memory throughout an entire program. The COMMON statement defines common blocks that are either named or unnamed (blank). Its syntax is as follows:

```
COMMON[/[name]/]nlist[[,]/[name]/nlist]...
```

where

name	is the optional name.
nlist	is a list of variable names, array names or array declarators. You cannot use function names or dummy argument names in this list.

If you omit the first name in a COMMON statement, you can omit the slashes (//). For any other name you omit in the sequence, the slashes (//) must be present.

The same common name (or blank name) can appear in other COMMON statements in the same program unit. The common block memory sequence consists of the memory of all items listed in the COMMON statement(s), in order of their appearance, for that common block name within a program unit.

An EQUIVALENCE statement can extend a common block. When this happens, the compiler adds memory beyond the highest location in the common block. An EQUIVALENCE statement cannot associate two different common blocks within a program. Common blocks with the same name but defined in different program units share the same starting address in memory.

A named COMMON block must be defined with the same length in every program unit that references it. Data statements in BLOCK DATA subprograms can initialize items only in named common blocks.

5.7 SAVE Statement

The SAVE statement ensures that specified variables within a FUNCTION or SUBROUTINE subprogram do not become undefined upon execution of a RETURN or END statement. Its syntax is as follows:

SAVE */name/*[, */name/*]...

where

name	is the *name* of a common block enclosed in slashes, a variable name, or an array name. Naming the common block in a SAVE statement saves all items in that block. Only one reference to a specific item can occur in a single SAVE statement. Local data names are not saved if the REENTRANT control is specified.

In an overlayed program, the SAVE statement does not ensure that variables will be saved across overlay, loading unless the subprogram is in the root. Values in common blocks will be saved only if the common block is declared in at least one program or subprogram in the root.

5.8 DATA Statement

The DATA statement gives the initial values of variables, arrays, and array elements. DATA statements cannot initialize dummy arguments or functions. DATA statements can initialize common memory only if the DATA statements are part of a BLOCK DATA subprogram. Its syntax is as follows:

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

where

nlist	is a list of variable names, array names, array element names and implied-DO lists.
clist	has the following form: [*r**]*c*[, [*r**]*c*]...

where

c	is any constant, even a Hollerith constant.
r	is a repeat specifier that is an unsigned, nonzero, integer constant.
r c*	is equivalent to *r* successive appearances of the constant *c*.

Both *nlist* and *clist* must have the same number of items since the lists correspond on a one-to-one basis. If *nlist* contains an array name without a subscript, *clist* must have one constant for each element in that array. All listed subscripts must be integer constant expressions.

The type of any name in *nlist* must agree with the type of the corresponding constant in *clist* except that you can initialize an item of any non-character type to a Hollerith constant.

If a variable or array element in *nlist* has a specific length, the length of its corresponding Hollerith constant in *clist* must be less than or equal to that length. If a length in *clist* is less than the corresponding length in *nlist*, the compiler pads the constant on the right with blanks until the two lengths are equal.

You can initialize a variable or an array element only once in a program. If an EQUIVALENCE statement associates two items, you can initialize only one of these items.

5.8.1 Implied-DO in a DATA Statement

The form of the implied-DO list is:

(*dlist*, *i* = m_1, m_2, [m_3])

where

dlist	is a list of array element names and implied-DO lists.
i	is the implied-DO variable.
m_1, m_2, and m_3	are integer constant expressions.

The range of an implied-DO list is *dlist*. The iteration count and the values of the implied-DO variable are established from m_1, m_2, and m_3 exactly as for a DO (see Section 8.3). When an implied-DO list appears in a DATA statement, items in *dlist* are specified once for each iteration of the list with the appropriate substitution of values for any occurrence of *r*. The presence of *i* in a DATA statement does not affect the status of any other variable with the same name in the same program unit. The following is an example of an implied-DO list in a data statement:

```
DIMENSION IARRAY (5)
DATA(IARRAY(I),I=1,5)/ 2, 4, 6, 8, 10/
```

5.9 INTRINSIC Statement

The INTRINSIC statement confirms that a symbolic name represents an intrinsic function and allows the use of that name as an actual argument. Its syntax is as follows:

INTRINSIC *name*[, *name*]...

where

name	is an intrinsic-function name. An intrinsic-function name can appear only once in any INTRINSIC statement in a program unit. The same intrinsic function name cannot appear in both an INTRINSIC statement and an EXTERNAL statement in the same program unit.

The following intrinsic-function names cannot appear in INTRINSIC statements: type conversion functions, lexical relationship functions, and functions for choosing the largest or smallest value.

5.10 EXTERNAL Statement

The EXTERNAL statement confirms that a symbolic name represents an external or dummy procedure and allows that name to appear as an actual argument. Its syntax is as follows:

EXTERNAL *name*[, *name*]...

where

name	is the name of an external or dummy procedure.

If an intrinsic function name appears in an EXTERNAL statement, that name no longer specifies an intrinsic function in the program unit, but instead becomes the name of an external procedure.

CHAPTER 6 SUBPROGRAMS

A Fortran program consists of a main program and any number of subprograms. This chapter describes the various kinds of subprograms, including subroutines, functions, and BLOCK DATA subprograms.

6.1 Types of Subprograms

Subroutines or functions organize programs into structures and enable multiple use of commonly used program units. A subprogram is largely self-contained, accepts arguments and, in the case of a function, returns a value to the invoking program unit.

Any other program unit can invoke a subroutine or function, but it cannot invoke itself unless compiled with the REENTRANT Control (see Section 11.4.15).

Programs that are created separately must be linked together before execution. For linking Fortran-86 procedures with procedures written in other programming languages, see Appendix H.

6.1.1 Arguments

You use dummy arguments in an argument list when you define a subprogram. You use actual arguments in a corresponding argument list when you reference that subprogram.

The actual arguments of subprograms must agree in order, number, type, and length with their corresponding dummy arguments. A dummy argument can be a variable, an array, a function, or a subroutine. The corresponding actual argument must be an expression, array, function, or subroutine, respectively.

The symbolic name of a dummy argument cannot appear in an EQUIVALENCE, SAVE, INTRINSIC, DATA, or COMMON statement.

All subscripts and expressions appearing in an actual argument list are evaluated before association of the actual and dummy arguments.

External and dummy procedures used as actual arguments must be defined in an EXTERNAL statement (see Section 5.9). Intrinsic functions used as actual arguments must be defined in an INTRINSIC statement (see Section 5.8). Statement functions are not permitted as actual arguments.

The following intrinsic functions can not be used as actual arguments:

INT	ICHAR	DMIN1
IFIX	CHAR	AMIN0
IDINT	MAX	MIN1
INT1	MAX0	LGE
INT2	AMAX1	LGT
INT4	DMAX1	LLE
REAL	AMAX0	LLT
FLOAT	MAX1	DCMPLX
SNGL	MIN	CMPLX
DBLE	MIN0	INTVL
TREAL	AMIN1	DINTVL

6.2 Subroutines

The first statement of any subroutine must be a SUBROUTINE statement (see Section 4.2.3, "SUBROUTINE Statement"). A subroutine can contain any type of statement except a FUNCTION, BLOCK DATA, or PROGRAM statement.

You reference a subroutine using a CALL statement (see Section 8.5). After the subroutine performs its operations, it restores control to the point of call with a RETURN or END statement (see Section 8.6).

6.2.1 Intrinsic Subroutines

Fortran-86 provides four intrinsic subroutines for handling input/output through eight-bit and sixteen-bit I/O ports.

For eight-bit ports, the forms of the subroutine call are as follows:

```
CALL INPUT(port, var)

CALL OUTPUT(port, exp)
```

where

port	is an integer constant in the range 0≤*port*≤65535.
var	is an integer variable.
exp	is an integer expression.

The value read or written for these intrinsic subroutines is a single-byte integer.

For sixteen-bit ports, the forms of the subroutine calls are as follows:

```
CALL INW(port, var)

CALL OUTW(port, exp)
```

where

port	is an integer constant in the range 0≤*port*≤65535.
var	is an integer variable.
exp	is an integer expression.

The value read or written for these intrinsic subroutines is a two-byte integer.

6.3 Functions

The types of functions available in Fortran-86 are as follows:

- FUNCTION subprograms—User-defined subprograms that the compiler identifies by their initial FUNCTION statements.
- Intrinsic functions—Predefined Fortran-86 functions which eliminate the coding of common mathematical functions.
- Statement functions—User-defined single statements which you define as functions.
- %VAL function—A non-standard function that enables parameter passing by value for program linkage with non-Fortran programs.

6.3.1 FUNCTION Subprograms

A FUNCTION subprogram is an external procedure that returns a value. The first statement of a FUNCTION subprogram must be a FUNCTION statement (see Section 4.2.2). The FUNCTION subprogram can contain any type of statement except a SUBROUTINE, BLOCK DATA, or PROGRAM statement.

The name of a FUNCTION subprogram is global and cannot be used within the subprogram except as a variable in the body of the subprogram which represents the return value. Within the subprogram, the name can appear in a type statement if the type has not already been specified in the FUNCTION statement. A type statement is the only nonexecutable statement where the name can appear.

The following is an example of a FUNCTION subprogram.

```
C THE FOLLOWING EXAMPLE TOTALS THE VALUES
C IN AN ARRAY OF LENGTH I
C THE TYPE OF THE FUNCTION IS REAL BY DEFAULT

      FUNCTION TOTAL(ARRAY,I)
        DIMENSION ARRAY(I)
        TOTAL = 0.0
        DO 100 K = 1,I
          TOTAL = TOTAL + ARRAY(K)
100     CONTINUE
        RETURN
      END
```

You reference a function by specifying its name in an expression, along with any necessary actual arguments.

The following function names are reserved for future implementation of interval arithmetic. If it is necessary to use any of these names in your program, clarify their use by including an EXTERNAL statement in your program. The EXTERNAL statement will confirm that the function name represents an external or dummy procedure and allows that name to appear in an actual argument.

DIST	ANMAX	ANSIN	ANATAN
WIDTH	DNMAX	DNSIN	DNATAN
SIGNUM	ANSQRT	ANCOS	ANATNZ
NTSCT	DNSQRT	DNCOS	DNATNZ
UNION	ANEXP	ANTAN	ANSINH
LEFT	DNEXP	DNTAN	DNSINH
RIGHT	ANLOG	ANASIN	ANCOSH
DRIGHT	DNLOG	DNASIN	DNCOSH
INTVL	ANLG10	ANACOS	ANTANH
DINTVL	DNLOG10	DNACOS	DNTANH

6.3.2 Intrinsic Functions

An intrinsic function is a predefined Fortran-86 function which performs common mathematical operations such as square root calculations and type conversions. The name of an intrinsic function can be either a generic name or a specific name. A generic name simplifies the referencing of an intrinsic function because you can use it with any of the data types defined for that function. You can use a specific name only with a data type defined for that name. You need to use the specific name if you are using the intrinsic function as an actual argument to a function or subroutine. For direct calls, the generic name is sufficient.

You reference an intrinsic function by specifying it in an expression. For example:

```
A = 33 + SQRT (B)
```

The resulting value (A in the example) depends on the value of the actual argument (B in the expression). There can be more than one argument in a list each separated by a comma, depending upon the particular function used. Each argument in the list must agree in type, number, and order with the specifications for the functions which are detailed later in this chapter. Restrictions on the range of arguments and results for intrinsic functions are given with the explanation of each function.

If the name of an intrinsic function appears in the dummy argument list of a FUNCTION or SUBROUTINE subprogram, the name has no relationship to that intrinsic function within the scope of the program unit. The data type associated with the symbolic name is specified either by default or by a type statement.

If you use the name of an intrinsic function as an actual argument in a FUNCTION or SUBROUTINE reference, you must specify it first in an INTRINSIC statement. The intrinsic functions for type conversion, lexical relationship, and for choosing the largest and smallest value cannot be actual arguments and can never appear in an INTRINSIC statement.

6.3.2.1 Intrinsic Type-Conversion Functions

Intrinsic type-conversion functions take an argument of one type and return a value of the type indicated by the particular function. The syntax is as follows:

name (*arg*)

where

name	is the generic or specific function name.
arg	is the value on which the function will be performed. Type-conversion functions can have only one argument.

Table 6-1 lists all the type-conversion functions by both generic and specific names with the acceptable types of their arguments and results.

The INT, INT1, INT2, INT4 functions take an argument of one type and return a value of type INTEGER. More specifically, INT(*arg*) returns an INTEGER value, INT1(*arg*) returns an INTEGER*1 value, INT2(*arg*) returns an INTEGER*2 value, and INT4(*arg*) returns an INTEGER*4 value.

Each of the names INT, INT1, INT2, and INT4 are generic names. Only the function INT has specific names associated with it as well. The function IFIX takes a REAL*4 argument and returns an INTEGER value. The results of IFIX(*arg*) and INT(*arg*) are the same. The function IDINT(*arg*) takes a DOUBLE PRECISION argument and returns an INTEGER value. See Table 6-1 for details.

For any real, double precision, or tempreal argument, two possible results exist. If $|arg| < 1$, then INT(*arg*) = 0; if $|arg| \geq 1$, then INT(*arg*) is the integer whose magnitude is the largest integer that does not exceed the magnitude of $|arg|$ and whose sign is the same as that of *arg*. For example:

```
INT(-12.8)=-12
```

Table 6-1. Type-Conversion Functions

Generic Name	Specific Name	Category	Function	Type	
				Arguments	Result
INT		Type Conversion	Convert to INTEGER	INTEGER	INTEGER
				INTEGER*1	INTEGER
				INTEGER*2	INTEGER
				INTEGER*4	INTEGER
	INT			REAL*4	INTEGER
	IFIX			REAL*4	INTEGER
				REAL*8	INTEGER
	IDINT			DOUBLE PRECISION	INTEGER
				TEMPREAL	INTEGER
				COMPLEX*8	INTEGER
				COMPLEX*16	INTEGER
INT1		Type Conversion	Convert to INTEGER*1	INTEGER	INTEGER*1
				INTEGER*1	INTEGER*1
				INTEGER*2	INTEGER*1
				INTEGER*4	INTEGER*1
				REAL*4	INTEGER*1
				REAL*8	INTEGER*1
				DOUBLE PRECISION	INTEGER*1
				TEMPREAL	INTEGER*1
				COMPLEX*8	INTEGER*1
				COMPLEX*16	INTEGER*1
INT2		Type Conversion	Convert to INTEGER*2	INTEGER	INTEGER*2
				INTEGER*1	INTEGER*2
				INTEGER*2	INTEGER*2
				INTEGER*4	INTEGER*2
				REAL*4	INTEGER*2
				REAL*8	INTEGER*2
				DOUBLE PRECISION	INTEGER*2
				TEMPREAL	INTEGER*2
				COMPLEX*8	INTEGER*2
				COMPLEX*16	INTEGER*2
INT4		Type Conversion	Convert to INTEGER*4	INTEGER	INTEGER*4
				INTEGER*1	INTEGER*4
				INTEGER*2	INTEGER*4
				INTEGER*4	INTEGER*4
				REAL*4	INTEGER*4
				REAL*8	INTEGER*4
				DOUBLE PRECISION	INTEGER*4
				TEMPREAL	INTEGER*4
				COMPLEX*8	INTEGER*4
				COMPLEX*16	INTEGER*4
REAL		Type Conversion	Convert to REAL	INTEGER	REAL*4
	FLOAT			INTEGER	REAL*4
				INTEGER*1	REAL*4
	FLOAT			INTEGER*1	REAL*4
				INTEGER*2	REAL*4
	FLOAT			INTEGER*2	REAL*4
				INTEGER*4	REAL*4
	FLOAT			INTEGER*4	REAL*4
				REAL*4	REAL*4
				REAL*8	REAL*4
	SNGL			DOUBLE PRECISION	REAL*4
				TEMPREAL	REAL*4
				COMPLEX*8	REAL*4
				COMPLEX*16	REAL*4

Table 6-1. Type-Conversion Functions (Cont'd.)

Generic Name	Specific Name	Category	Function	Type	
				Arguments	Result
CMPLX		Type Conversion	Convert to COMPLEX	INTEGER INTEGER*1 INTEGER*2 INTEGER*4 REAL REAL*4 REAL*8 DOUBLE PRECISION TEMPREAL COMPLEX*8 COMPLEX*16	COMPLEX COMPLEX COMPLEX COMPLEX COMPLEX COMPLEX COMPLEX COMPLEX COMPLEX COMPLEX COMPLEX COMPLEX
DCMPLX		Type Conversion	Convert to COMPLEX	INTEGER INTEGER*1 INTEGER*2 INTEGER*4 REAL REAL*4 REAL*8 DOUBLE TEMPREAL COMPLEX*8 COMPLEX*16	COMPLEX*16 COMPLEX*16 COMPLEX*16 COMPLEX*16 COMPLEX*16 COMPLEX*16 COMPLEX*16 COMPLEX*16 COMPLEX*16 COMPLEX*16 COMPLEX*16
DBLE		Type Conversion	Convert to DOUBLE PRECISION	INTEGER INTEGER*1 INTEGER*2 INTEGER*4 REAL*4 REAL*8 DOUBLE PRECISION TEMPREAL COMPLEX*8 COMPLEX*16	DOUBLE PRECISION DOUBLE PRECISION DOUBLE PRECISION DOUBLE PRECISION DOUBLE PRECISION DOUBLE PRECISION DOUBLE PRECISION DOUBLE PRECISION DOUBLE PRECISION DOUBLE PRECISION
TREAL		Type Conversion	Convert to TEMPREAL	INTEGER INTEGER*1 INTEGER*2 INTEGER*4 REAL*4 REAL*8 DOUBLE PRECISION TEMPREAL COMPLEX*8 COMPLEX*16	TEMPREAL TEMPREAL TEMPREAL TEMPREAL TEMPREAL TEMPREAL TEMPREAL TEMPREAL TEMPREAL TEMPREAL
	ICHAR	Type Conversion	Convert CHAR to INTEGER	CHARACTER	INTEGER
CHAR		Type Conversion	Convert INTEGER to CHARACTER	INTEGER INTEGER*1 INTEGER*2 INTEGER*4	CHARACTER CHARACTER CHARACTER CHARACTER

The REAL function takes an argument of any type and returns a corresponding REAL*4 value. For any integer argument (type INTEGER, INTEGER*1, INTEGER*2, INTEGER*4, or COMPLEX), you can use either of the names REAL or FLOAT. The results of REAL(*arg*) and FLOAT(*arg*) are the same. For a double precision argument, you can use the specific name SNGL in place of REAL. See Table 6-1 for details.

For any integer, double-precision, or tempreal argument, the result of a REAL(*arg*) function has as much precision of the significant part of *arg* that REAL data can contain.

The CMPLX function takes one or two arguments of type INTEGER, REAL, DOUBLE PRECISION, TEMPREAL, COMPLEX, or COMPLEX*16, and returns a value of type COMPLEX. If there are two arguments, they must both be the same type and must not be COMPLEX. If there are two arguments, the first argument will be used as the real part of the returned value and the second argument will be used as the imaginary part. If there is one argument of any type except COMPLEX or COMPLEX*16, the argument will be used as the real part of the returned value and the imaginary part wil be zero. If the argument is COMPLEX or COMPLEX*16, the real and imaginary parts of the argument will be used as the corresponding parts of the returned value.

The DCMPLX function takes one or two arguments of type INTEGER, REAL, DOUBLE PRECISION, TEMPREAL, COMPLEX, or COMPLEX*16, and returns a value of type COMPLEX*16. If there are two arguments they must both be the same type, and they must not be complex arguments. If there are two arguments, the first argument will be used as the real part of the returned value and the second argument will be used as the imaginary part. If there is one argument of any type except COMPLEX or COMPLEX*16, the argument will be used as the real part of the returned value and the imaginary part will be zero. If the argument is COMPLEX or COMPLEX*16, the real and imaginary parts of the argument will be used as the corresponding parts of the returned value.

The DBLE Function takes an argument of any type and returns a double precision value.

For any argument, integer, real, or tempreal, the result of a DBLE(*arg*) function has as much precision of the significant part of *arg* that DOUBLE PRECISION data can contain.

The TREAL function takes an argument of any type and returns a TEMPREAL value.

For any integer, real, or double-precision argument, the result of a TREAL(*arg*) function has as much precision of the significant part of *arg* as TEMPREAL data can contain.

The ICHAR and CHAR functions provide a way to convert from characters to integers, and vice versa, based on the position of the character in the ASCII collating sequence (See Appendix E, "Character Set and Collating Sequence"). The first character in the collating sequence corresponds to position 0 and the last to position $n - 1$, where n is the number of characters in the collating sequence.

The value of ICHAR(*arg*) is an integer in the range $0 \leq \text{ICHAR}(arg) \leq n - 1$,

where

arg	is an argument of type CHARACTER and length one.

For example:

```
ICHAR('%') is 25H
```

The function CHAR provides a way to convert integers to characters, based on the ASCII collating sequence (see Appendix E.) The value of CHAR(*arg*) is the character that appears in the *arg* position in the collating sequence. The argument must be an integer expression whose value is in the range of $0 \leq arg \leq n - 1$. For example:

```
CHAR(25H) is '%'
```

6.3.2.2 Truncation and Rounding Functions

Truncation and rounding functions perform numeric conversions. Both types of functions take only one argument. See Table 6-2 for details. The syntax is as follows:

name(*arg*)

where

name	is the intrinsic function name.
arg	is the value on which the function will be performed.

The AINT function truncates an argument leaving a result that is the integer part of the argument. For REAL*8 and DOUBLE PRECISION arguments, you can use the specific name DINT in place of AINT.

Table 6-2. Truncation and Rounding Functions

Generic Name	Specific Name	Category	Function	Type: Arguments	Type: Results
AINT		Truncation	Truncate Argument		
	AINT			REAL*4	REAL*4
	DINT			REAL*8	REAL*8
	DINT			DOUBLE PRECISION	DOUBLE PRECISION
				TEMPREAL	TEMPREAL
ANINT		Rounding	Round to Nearest Whole Number		
	ANINT			REAL*4	REAL*4
	DNINT			REAL*8	REAL*8
	DNINT			DOUBLE PRECISION	DOUBLE PRECISION
				TEMPREAL	TEMPREAL
NINT		Rounding	Round to Nearest Integer		
	NINT			REAL*4	INTEGER
	IDNINT			REAL*8	INTEGER
	IDNINT			DOUBLE PRECISION	INTEGER
				TEMPREAL	INTEGER
RINT		Rounding	Round to Even Whole Number		
	RINT			REAL*4	REAL*4
	DRINT			REAL*8	REAL*8
	DRINT			DOUBLE PRECISION	DOUBLE PRECISION
				TEMPREAL	TEMPREAL
IRINT		Rounding	Round to Even Integer		
	IRINT			REAL*4	INTEGER
	IDRINT			REAL*8	INTEGER
	IDRINT			DOUBLE PRECISION	INTEGER
				TEMPREAL	INTEGER

For any real, double-precision, or tempreal argument, AINT(*arg*) is the integer whose magnitude is the largest integer that does not exceed the magnitude of *arg* and whose sign is the same as *arg*. Note that for $-1.0 < \text{arg} < 0$, a negative zero is returned for AINT.

The ANINT function rounds an argument to the nearest whole number. For REAL*8 and DOUBLE PRECISION arguments, you can use the specific name DNINT in place of ANINT. The following formulas apply:

$$\begin{aligned}\text{ANINT}(x) = {} & \text{AINT}\ (x + .5), \text{ if } x \geq 0 \\ & \text{AINT}\ (x - .5), \text{ if } x < 0\end{aligned}$$

The NINT function rounds an argument to the nearest INTEGER. For REAL*8 and DOUBLE PRECISION arguments, you can use the specific name IDNINT in place of NINT. The following formula applies:

$$\text{NINT}(x) = \text{INT}\ (\text{ANINT}(x))$$

The RINT function rounds an argument to the nearest or even whole number. For REAL*8 or DOUBLE PRECISION arguments, you can use the specific name DRINT in place of RINT. The following formula applies:

$$\begin{aligned}\text{RINT}(x) &= \text{AINT}(x), \text{ if } |\text{AINT}(x) = |x| - .5 \text{ and is even} \\ &= \text{ANINT}(x), \text{ otherwise}\end{aligned}$$

The IRINT function rounds an argument to the *nearest* INTEGER, or to the nearest even INTEGER if the argument is at the midpoint between two whole INTEGERS. For REAL*8 or DOUBLE PRECISION arguments, you can use the specific name IDRINT in place of IRINT. The following formula applies:

$$\text{IRINT}(x) = \text{INT}(\text{RINT}(x))$$

6.3.2.3 The Remainder Functions

The remainder functions perform a division operation on two arguments and return the remainder. See Table 6-3 for details. The syntax is as follows:

name(*arg1* , *arg2*)

where

name	is the intrinsic function name.
arg1 and *arg2*	are the values on which the function will be performed.

The remainder functions will never incur a rounding error. In addition, the result of a valid remainder operation is never an unnormal number.

The MOD function is equivalent to the operation that follows:

argl − AINT(*argl*/*arg2*)**arg2*

For REAL*4 arguments, you can use the specific name AMOD in place of MOD. For REAL*8 or DOUBLE PRECISION arguments, you can use the specific name DMOD in place of MOD.

The RMD function is equivalent to the operation that follows:

argl − RINT(*argl*/*arg2*) **arg2*

Table 6-3. Remainder Functions

Generic Name	Specific Name	Category	Function	Type	
				Arguments	Results
MOD		Remainder	arg1-AINT (arg1/arg2) *arg2	INTEGER	INTEGER
				INTEGER*1	INTEGER*1
				INTEGER*2	INTEGER*2
				INTEGER*4	INTEGER*4
	AMOD			REAL*4	REAL*4
	DMOD			REAL*8	REAL*8
	DMOD			DOUBLE PRECISION	DOUBLE PRECISION
				TEMPREAL	TEMPREAL
RMD	IRMD	Remainder	arg1-RINT (arg1/arg2) *arg2	INTEGER	INTEGER
				INTEGER*1	INTEGER*1
				INTEGER*2	INTEGER*2
				INTEGER*4	INTEGER*4
				REAL*4	REAL*4
	DRMD			REAL*8	REAL*8
	DRMD			DOUBLE PRECISION	DOUBLE PRECISION
				TEMPREAL	TEMPREAL

For INTEGER arguments, you can use the specific name IRMD in place of RMD. For REAL*8 or DOUBLE PRECISION arguments, you can use the specific name DRMD in place of RMD.

The ABS function returns the absolute value of an argument. Its syntax is as follows:

ABS(*arg*)

where

arg is the value on which the function will be performed.

See Table 6-4 for details.

For INTEGER arguments, you can use the specific name iABS in place of ABS. For REAL*8 or DOUBLE PRECISION functions, you can use the symbolic name DABS in place of ABS. You can use the symbolic name CABS for COMPLEX*8 arguments. For COMPLEX*16 arguments, you can use the symbolic name CDABS.

The SIGN function takes the sign of the second argument and transfers this sign to the first argument. Specifically, it returns the following value:

$|arg1|$, if $arg2 \geq 0$
$-|arg1|$, if $arg2 < 0$

The syntax is as follows:

SIGN(*arg1*, *arg2*)

where

arg1 and *arg2* are the values on which the function will be performed.

See Table 6-4 for details.

Table 6-4. Absolute Value, Sign Transfer, Positive Difference, and Double Precision Product Functions

Generic Name	Specific Name	Category	Function	Type	
				Arguments	Results
ABS	IABS	Absolute	Return	INTEGER	INTEGER
		Value	Absolute	INTEGER*1	INTEGER*1
			Value	INTEGER*2	INTEGER*2
				INTEGER*4	INTEGER*4
	ABS			REAL*4	REAL*4
	DABS			REAL*8	REAL*8
	DABS			DOUBLE	DOUBLE
				PRECISION	PRECISION
				TEMPREAL	TEMPREAL
	CABS			COMPLEX*8	COMPLEX*8
	CDABS			COMPLEX*16	COMPLEX*16
SIGN	ISIGN	Sign	Transfer	INTEGER	INTEGER
		Transfer	Sign of	INTEGER*1	INTEGER*1
			arg2 to arg1	INTEGER*2	INTEGER*2
			sign(y,x)=	INTEGER*4	INTEGER*4
	SIGN		1y1.x≥0	REAL*4	REAL*4
	DSIGN		1y1.x≤0	REAL*8	REAL*8
	DSIGN			DOUBLE	DOUBLE
				PRECISION	PRECISION
				TEMPREAL	TEMPREAL
DIM	IDIM	Positive	Return	INTEGER	INTEGER
		Difference	arg1 − arg2	INTEGER*1	INTEGER*1
			if arg1>arg2	INTEGER*2	INTEGER*2
			else 0	INTEGER*4	INTEGER*4
	DIM			REAL*4	REAL*4
	DDIM			REAL*8	REAL*8
	DDIM			DOUBLE	DOUBLE
				PRECISION	PRECISION
	DDIM			TEMPREAL	TEMPREAL
	DPROD	Double	Multiply	REAL*4	DOUBLE
		Precision	arg1 by arg2		PRECISION
		Product			

For INTEGER arguments, you can use the specific name ISIGN in place of SIGN. For REAL*8 or DOUBLE PRECISION arguments, you can use the specific name DSIGN in place of SIGN.

If the value of the first argument is zero, the result is zero.

The DIM function returns the result of the operation *arg1* − *arg2* as long as this result is positive. If the operation gives a negative result, the result of the function is zero. Specifically, it returns the following value:

argl − *arg2*, if *argl* ≥ *arg2*
otherwise, 0.0

If the result of the subtraction is negative, an underflow exception cannot occur.

The syntax is as follows:

DIM(*arg1*, *arg2*)

where

arg1 and *arg2*	are the values on which the function will be performed.

See Table 6-4 for details.

For INTEGER arguments, you can use the specific name IDIM in place of DIM. For REAL*8 or DOUBLE PRECISION arguments, you can use the specific name DDIM in place of DIM.

The DPROD function takes two arguments of type REAL*4, multiplies them, and returns a DOUBLE PRECISION result. Its syntax is as follows:

DPROD(*arg1*, *arg2*)

where

arg1 and *arg2*	are the arguments on which the function will be performed.

See Table 6-4 for details. DPROD is sensitive to the precision mode on the 8087 processor. You can use this function meaningfully only if the precision is set to 53 or 64 bits. DPROD provides no advantage over the multiplication operation (*) in Fortran-86.

6.3.2.4 Choosing the Largest or Smallest Value Functions

The largest or smallest value functions evaluate a list of at least two arguments and choose the largest or smallest value, depending upon the function. The syntax is as follows:

name (*arg1*, *arg2...*)

where

name	is the intrinsic function name.
arg1, *arg2*, ...	are the arguments from which it chooses.

See Table 6-5 for details.

The MAX function evaluates a list of at least two arguments and chooses the largest value. For INTEGER arguments, you can use the specific name MAX0 in place of MAX. For REAL*4 arguments, you can use the specific name AMAX1 in place of MAX. For DOUBLE PRECISION arguments, you can use the specific name DMAX1 in place of MAX.

The AMAX0 function evaluates a list of at least two arguments and chooses the largest value. The arguments can be only of type INTEGER or type REAL*4. For REAL*4 arguments, you can use the specific name MAX1 in place of AMAX0.

The MIN function evaluates a list of at least two arguments and chooses the smallest value. For INTEGER arguments, you can use the specific name MIN0 in place of MIN. For REAL*4 arguments, you can use the specific name AMIN1 in place of MIN. For DOUBLE PRECISION arguments, you can use the specific name DMIN1 in place of MIN.

Table 6-5. Choosing the Largest or Smallest Value Functions

Generic Name	Specific Name	Category	Function	Type	
				Arguments	Results
MAX	MAX0	Largest Value	Choose Largest Value in List	INTEGER	INTEGER
				INTEGER*1	INTEGER*1
				INTEGER*2	INTEGER*2
				INTEGER*4	INTEGER*4
	AMAX1			REAL*4	REAL*4
	DMAX1			REAL*8	REAL*8
				DOUBLE PRECISION	DOUBLE PRECISION
				TEMPREAL	TEMPREAL
AMAX0		Largest Value	Choose Largest Value in List	INTEGER	REAL*4
				INTEGER*1	REAL*4
				INTEGER*2	REAL*4
				INTEGER*4	REAL*4
	MAX1			REAL*4	INTEGER
MIN	MIN0	Smallest Value	Choose Smallest Value in List	INTEGER	INTEGER
				INTEGER*1	INTEGER*1
				INTEGER*2	INTEGER*2
				INTEGER*4	INTEGER*4
	AMIN1			REAL*4	REAL*4
	DMIN1			REAL*8	REAL*8
				DOUBLE PRECISION	DOUBLE PRECISION
				TEMPREAL	TEMPREAL
AMIN0		Smallest Value	Choose Smallest Value in List	INTEGER	REAL*4
				INTEGER*1	REAL*4
				INTEGER*2	REAL*4
				INTEGER*4	REAL*4
	MIN1			REAL*4	INTEGER

The AMIN0 function evaluates a list of at least two arguments and chooses the smallest value. The arguments can be only of type INTEGER or type REAL*4.

For REAL*4 arguments, you can use the specific name MIN1 in place of AMIN0.

6.3.2.5 The LEN and INDEX Functions

The LEN function returns the length of a character string. Its syntax is as follows:

LEN(*arg*)

where

arg	is the character string.

When the program executes the LEN function, the value of the argument need not be defined.

The INDEX function determines whether a character string appears in a second character string. It returns an INTEGER value representing the starting character position of the first string in the second string if found; otherwise, a zero value is returned. Its syntax is as follows:

INDEX(*arg1*, *arg2*)

where

arg1 and *arg2* are character strings.

See Table 6-6 for details of both these functions.

6.3.2.6 The Arithmetic Functions

The Arithmetic functions perform specific arithmetic functions. The syntax is as follows:

name(*arg*)

where

name	is the intrinsic function name.
arg	is the argument on which the function will be performed. All functions expect and return floating-point values.

See Table 6-7 for details.

The PROJ Function provides a mechanism for uniformly handling complex infinities. A complex infinity is any complex number with at least one infinite component. Whenever users are performing algebraic computations in which they suspect that a complex infinity may arise as input to an algebraic operation, they should apply the PROJ function to such inputs.

The PROJ function is defined as follows:

PROJ((x.y))=(INFINITY, SIGN(0,Y)), if either X or Y is infinite;
=(x,y), otherwise.

The AIMAG function returns the imaginary part of a COMPLEX argument.

You can use the specific name AIMAG for COMPLEX*8 arguments, and the specific name DAIMAG for COMPLEX*16 arguments.

The CONJG function returns the conjugate of a COMPLEX argument.

You can use the specific name CONJG for COMPLEX*8 arguments, and the specific name DCONJG for COMPLEX*16 arguments.

The SQRT function returns the square root of an argument. The argument must be greater than or equal to zero.

For REAL*8 and DOUBLE PRECISION arguments, you can use the specific name DSQRT in place of SQRT. You can use the specific name CSQRT for COMPLEX*8 arguments. For COMPLEX*16 arguments, you can use the specific name CDSQRT.

The EXP function returns a value that is equal to e raised to the power of the argument.

Table 6-6. Length and Index Functions

Generic Name	Specific Name	Category	Function	Type	
				Arguments	Results
	LEN	Length	Determine the Length of Character Entity	CHARACTER	INTEGER
	INDEX	Index of Substring	Return Location of Substring arg2 in String arg1	CHARACTER	INTEGER

Table 6-7. Arithmetic Functions

Generic Name	Specific Name	Category	Function	Type	
				Arguments	Results
AIMAG	AIMAG DAIMAG	Arithmetic	Returns Imaginary Part of a Complex Argument	COMPLEX*8 COMPLEX*16	REAL DOUBLE PRECISION
CONJG	CONJG DCONJG	Arithmetic	Returns Conjugate of a Complex Argument	COMPLEX COMPLEX*8 COMPLEX*16	COMPLEX COMPLEX*8 COMPLEX*16
SQRT	SQRT DSQRT DSQRT CSQRT CDSQRT	Arithmetic	Return Square Root	REAL*4 REAL*8 DOUBLE PRECISION TEMPREAL COMPLEX*8 COMPLEX*16	REAL*4 REAL*8 DOUBLE PRECISION TEMPREAL COMPLEX*8 COMPLEX*16
EXP	EXP DEXP DEXP CEXP CDEXP	Arithmetic	Return e Raised to Power of Argument	REAL*4 REAL*8 DOUBLE PRECISION TEMPREAL COMPLEX*8 COMPLEX*16	REAL*4 REAL*8 DOUBLE PRECISION TEMPREAL COMPLEX*8 COMPLEX*16
LOG	ALOG DLOG DLOG CLOG CDLOG	Arithmetic	Return Natural Logarithm	REAL*4 REAL*8 DOUBLE PRECISION TEMPREAL COMPLEX*8 COMPLEX*16	REAL*4 REAL*8 DOUBLE PRECISION TEMPREAL COMPLEX*8 COMPLEX*16
LOG10	ALOG10 DLOG10 DLOG10	Arithmetic	Return Common Logarithm	REAL*4 REAL*8 DOUBLE PRECISION TEMPREAL	REAL*4 REAL*8 DOUBLE PRECISION TEMPREAL

For REAL*8 or DOUBLE PRECISION arguments, you can use the specific name DEXP in pla‹ › of EXP. For COMPLEX*8 arguments, you can use the specific name CEXP. You ca. use the specific name CDEXP for COMPLEX*16 arguments.

The LOG function .eturns the natural logarithm of an argument. The argument must be greater than or equal to zero.

For REAL*4 arguments, you can use the specific name ALOG in place of LOG. For REAL*8 or DOUBLE PRECISION arguments, you can use the specific name DLOG in place of LOG. You can use the specific name CLOG for COMPLEX*8 arguments. For COMPLEX*16 arguments, you can use the specific name CDLOG.

The ALOG10 function returns the common logarithm of an argument. The argument must be greater than zero.

For REAL*8 or DOUBLE PRECISION arguments, you can use the specific name DLOG10 in place of ALOG10.

6.3.2.7 The Trigonometric Functions

The trigonometric functions perform specified trigonometric functions. The syntax is as follows:

name (*arg*)

where

name	is the intrinsic function name.
arg	is the value on which the function will be performed. All trigonometric functions expect and return floating-point values.

See Table 6-8 for details.

The SIN function returns the sine of an argument. The absolute value of the argument is not restricted to be less than 2PI. The range of the result is $-1 \leq$ result ≤ 1.

For REAL*8 and DOUBLE PRECISION arguments, you can use the specific name DSIN in place of SIN. You can use the specific name CSINH for COMPLEX*8 arguments. For COMPLEX*16 arguments, you can use the specific name CDSINH.

The COS function returns the cosine of an argument. The absolute value of the argument is not restricted to be less than 2PI. The range of the result is $-1 \leq$ result ≤ 1.

For REAL*8 and DOUBLE PRECISION arguments, you can use the specific name DCOS in place of COS. You can use the specific name COSH for arguments that are COMPLEX*8. For arguments that are COMPLEX*16, you can use the specific name CDCOSH.

The TAN function returns the tangent of an argument. The absolute value of the argument is not restricted to be less than 2PI.

For REAL*8 and DOUBLE PRECISION arguments, you can use the specific name DTAN in place of TAN. You can use the specific name CTANH for COMPLEX*8 arguments. For COMPLEX*16 arguments, you can use the specific name CDTANH.

The ASIN function returns the arcsine of an argument. The absolute value of the argument must be ≤ 1. The range of the result is $-PI/2 \leq$ result $\leq PI/2$.

For REAL*8 or DOUBLE PRECISION arguments, you can use the specific name DASIN in place of ASIN. You can use the specific name CASIN for COMPLEX*8 arguments. For COMPLEX*16 arguments, you can use the specific name CDASIN.

The ACOS function returns the arccosine of an argument. The absolute value of the argument must be ≤ 1. The range of the result is $0 \leq$ result $\leq$ PI.

For REAL*8 or DOUBLE PRECISION arguments, you can use the specific name DACOS in place of ACOS. You can use the specific name CACOS for COMPLEX*8 arguments. For COMPLEX*16 arguments, you can use the specific name CDACOS.

The ATAN function returns the arctangent of an argument. The range of the result is $-PI/2 \leq$ result $\leq PI/2$. If the value of the argument is positive, the result is positive.

For REAL*8 or DOUBLE PRECISION arguments, you can use the specific name DATAN in place of ATAN. You can use the specific name CATAN for COMPLEX*8 arguments. For COMPLEX*16 arguments, you can use the specific name CDATAN.

The ATAN2 function computes the principal value of the angular component of the polar coordinates of a point whose rectangular coordinates are arg1 and arg2, its first and second arguments, respectively. The restriction to principal value means that the result satisfies $-PI \leq$ result $\leq$ PI, and guarantees that any finite point other than

Table 6-8. Trigonometric Functions

Generic Name	Specific Name	Category	Function	Type	
				Arguments	Results
SIN		Trigonometric	Return Sine	REAL*4	REAL*4
	DSIN			REAL*8	REAL*8
	DSIN			DOUBLE PRECISION	DOUBLE PRECISION
				TEMPREAL	TEMPREAL
	CSIN			COMPLEX*8	COMPLEX*8
	CDSIN			COMPLEX*16	COMPLEX*16
COS		Trigonometric	Return Cosine	REAL*4	REAL*4
	DCOS			REAL*8	REAL*8
	DCOS			DOUBLE PRECISION	DOUBLE PRECISION
				TEMPREAL	TEMPREAL
	CCOS			COMPLEX*8	COMPLEX*8
	CDCOS			COMPLEX*16	COMPLEX*16
TAN		Trigonometric	Return Tangent	REAL*4	REAL*4
	DTAN			REAL*8	REAL*8
	DTAN			DOUBLE PRECISION	DOUBLE PRECISION
				TEMPREAL	TEMPREAL
	CTAN			COMPLEX*8	COMPLEX*8
	CDTAN			COMPLEX*16	COMPLEX*16
ASIN		Trigonometric	Return Arcsine	REAL*4	REAL*4
	DASIN			REAL*8	REAL*8
	DASIN			DOUBLE PRECISION	DOUBLE PRECISION
				TEMPREAL	TEMPREAL
	CASIN			COMPLEX*8	COMPLEX*8
	CDASIN			COMPLEX*16	COMPLEX*16
ACOS		Trigonometric	Return Arccosine	REAL*4	REAL*4
	DACOS			REAL*8	REAL*8
	DACOS			DOUBLE PRECISION	DOUBLE PRECISION
				TEMPREAL	TEMPREAL
	CACOS			COMPLEX*8	COMPLEX*8
	CDACOS			COMPLEX*16	COMPLEX*16
ATAN		Trigonometric	Return Arctangent with one Argument	REAL*4	REAL*4
	DATAN			REAL*8	REAL*8
	DATAN			DOUBLE PRECISION	DOUBLE PRECISION
				TEMPREAL	TEMPREAL
	CATAN			COMPLEX*8	COMPLEX*8
	CDATAN			COMPLEX*16	COMPLEX*16
ATAN2		Trigonometric	Return Arctangent with two Arguments	REAL*4	REAL*4
	DATAN2			REAL*8	REAL*8
	DATAN2			DOUBLE PRECISION	DOUBLE PRECISION
				TEMPREAL	TEMPREAL

the origin has a unique angular component. The function assigns angular components to the origin (± 0, ± 0) also, namely $+0$ or $\pm \pi$, depending on the signs of the rectangular coordinates.

The results for other finite inputs are as follows:

ATAN(*arg2/arg1*), if *arg1* $>$ 0

SIN(PI/2,*arg2*), if *arg1*$=$0 and *arg2* $\neq$ 0

ATAN(*arg2/arg1*) $+$ SIGN(PI,*arg2*), if *arg1*$<$0

Note that a correct result is returned even if some of the arguments are infinite.

6.3.2.8 Hyperbolic Functions

Hyperbolic functions perform specified hyperbolic functions. The syntax is as follows:

name(*arg*)

where

name	is the intrinsic function name.
arg	is the value on which the function will be performed. All hyperbolic functions expect and return floating-point values.

See Table 6-9 for details.

Table 6-9. Hyperbolic Functions

Generic Name	Specific Name	Category	Function	Type	
				Arguments	Results
SINH		Hyperbolic	Return Hyperbolic Sine	REAL*4	REAL*4
	DSINH			REAL*8	REAL*8
	DSINH			DOUBLE PRECISION	DOUBLE PRECISION
				TEMPREAL	TEMPREAL
	CSINH			COMPLEX*8	COMPLEX*8
	CDSINH			COMPLEX*16	COMPLEX*16
COSH		Hyperbolic	Return Hyperbolic Cosine	REAL*4	REAL*4
	DCOSH			REAL*8	REAL*8
	DCOSH			DOUBLE PRECISION	DOUBLE PRECISION
				TEMPREAL	TEMPREAL
	CCOSH			COMPLEX*8	COMPLEX*8
	CDCOSH			COMPLEX*16	COMPLEX*16
TANH		Hyperbolic	Return Hyperbolic Tangent	REAL*4	REAL*4
	DTANH			REAL*8	REAL*8
	DTANH			DOUBLE PRECISION	DOUBLE PRECISION
				TEMPREAL	TEMPREAL
	CTANH			COMPLEX*8	COMPLEX*8
	CDTANH			COMPLEX*16	COMPLEX*16

The SINH function returns the hyperbolic sine of an argument. For REAL*8 or DOUBLE PRECISION arguments, you can use the specific name DSINH in place of SINH. You can use the specific name CSINH for COMPLEX*8 arguments. For COMPLEX*16 arguments, you can use the specific name CDSINH.

The COSH function returns the hyperbolic cosine of an argument. For REAL*8 or DOUBLE PRECISION arguments, you can use the specific name DCOSH in place of COSH. You can use the specific name CCOSH for COMPLEX*8 arguments. For COMPLEX*16 arguments, you ca use the specific name CDCOSH.

The TANH function returns the hyperbolic tangent of an argument. For REAL*8 or DOUBLE PRECISION arguments, you can use the specific name DTANH in place of TANH. You can use the specific name CTANH for COMPLEX*8 arguments. For COMPLEX*16 arguments, you can use the specific name CDTANH.

6.3.2.9 The Lexical-Relationship Functions

The lexical-relationship functions take two CHARACTER arguments and depending on the position of the arguments in the ASCII collating sequence (see Appendix E), return a LOGICAL value. The syntax is as follows:

name(*arg1* , *arg2*)

where

name	is the intrinsic-function name.
arg1 and *arg2*	are the CHARACTER arguments on which the function will be performed.

See Table 6-10 for details.

The LGE function determines if the first argument is equal to, or greater than, the second argument in the ASCII collating sequence. If the arguments have the same value or *arg1* follows *arg2* in the collating sequence, the result is .TRUE.. If any other condition exists, the result is .FALSE..

The LGT function determines if the first argument is greater than the second argument in the ASCII collating sequence. If *arg1* follows *arg2* the result is .TRUE.. If any other condition exists, then the result is .FALSE..

Table 6-10. Lexical-Relationship Functions

Generic Name	Specific Name	Category	Function	Type	
				Arguments	Results
LGE	LGE	Lexical Relationship	Lexically Greater or Equal	CHARACTER	LOGICAL
	LGT	Lexical Relationship	Lexically Greater	CHARACTER	LOGICAL
	LLE	Lexical Relationship	Lexically Less or Equal	CHARACTER	LOGICAL
	LLT	Lexical Relationship	Lexically Less	CHARACTER	LOGICAL

The LLE function determines if the first argument is equal to, or less than, the second argument in the ASCII collating sequence. If the arguments have the same value or *arg2* follows *arg1* in the collating sequence, the result is .TRUE.. If any other condition exists, the result is .FALSE..

The LLT function determines if the first argument is less than the second argument in the ASCII collating sequence. If *arg2* follows *arg1* the result is .TRUE.. If any other condition exists, the result is .FALSE..

6.3.2.10 8087 Control Intrinsics

Fortran-86 provides default computation modes and exception masks which are suitable for most floating-point applications. Occasionally, however, you may require a special option, such as unmasked overflow exception handling, or normalizing mode for denormal (gradual-underflow) results. Changing computation modes and exception masks, as well as in-line testing of (masked) exception flags, is possible using 8087-control intrinsic procedures.

Table 6-11 shows the 8087 control intrinsics together with the resulting in-line assembly instructions. Figure 6-1 describes the format of the 8087 control word, which specifies the various processor options relevant to Fortran-86. Also shown is the 8087 status-word format which is required for testing whether masked exceptions have occurred (STSW87).

For further information relating to the 8087 Numeric Data Processor, refer to the *ASM86 Language Reference Manual*, 121703, which explains the 8087 instructions and control in detail.

Table 6-11. 8087 Control Intrinsics

Form	Function	8087 Instruction Generated
Call STSW87(wd)	Store 87 Status Word	PUSHF CLI FNSTSW @ wd FNCLEX FWAIT POPF
Call LDCW87(wd)	Load 87 Control Word	PUSHF CLI FNLDCW @ wd POPF
Call STCW87(wd)	Store 87 Control Word	PUSHF CLI FNSTCW @ wd POPF
Call SAV87(st)	Save 87 State	PUSHF CLI FNSAVE @ st FWAIT POPF
Call RST87(wd)	Restore 87 State	FRSTOR @ st FWAIT

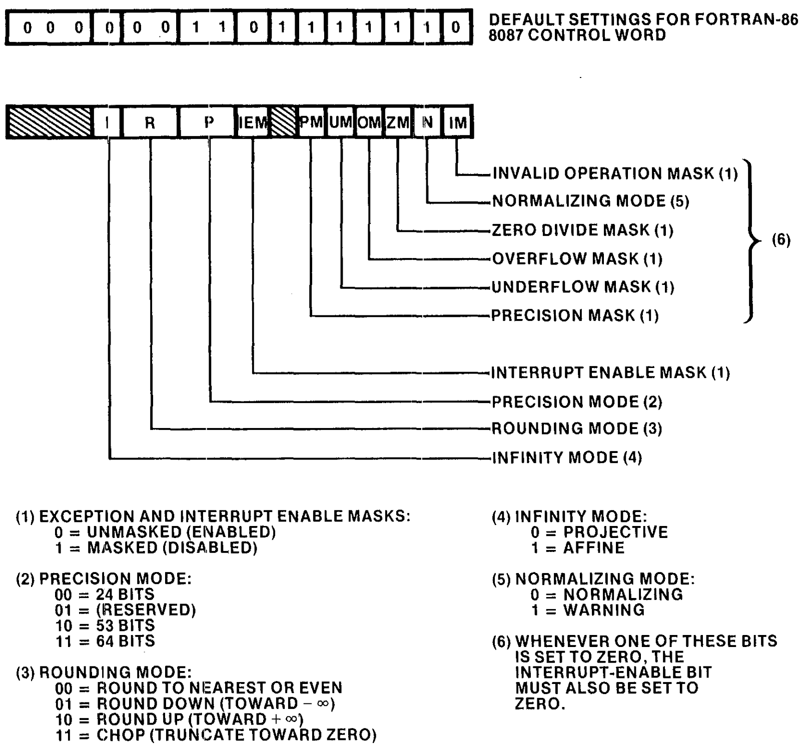

0 0 0 0 0 0 1 1 0 1 1 1 1 1 1 0
DEFAULT SETTINGS FOR FORTRAN-86 8087 CONTROL WORD
I
R
P
IEM
PM
UM
OM
ZM
N
IM
INVALID OPERATION MASK (1)
NORMALIZING MODE (5)
ZERO DIVIDE MASK (1)
OVERFLOW MASK (1)
UNDERFLOW MASK (1)
PRECISION MASK (1)
(6)
INTERRUPT ENABLE MASK (1)
PRECISION MODE (2)
ROUNDING MODE (3)
INFINITY MODE (4)
(1) EXCEPTION AND INTERRUPT ENABLE MASKS: 0 = UNMASKED (ENABLED) 1 = MASKED (DISABLED)
(2) PRECISION MODE: 00 = 24 BITS 01 = (RESERVED) 10 = 53 BITS 11 = 64 BITS
(3) ROUNDING MODE: 00 = ROUND TO NEAREST OR EVEN 01 = ROUND DOWN (TOWARD − ∞) 10 = ROUND UP (TOWARD + ∞) 11 = CHOP (TRUNCATE TOWARD ZERO)
(4) INFINITY MODE: 0 = PROJECTIVE 1 = AFFINE
(5) NORMALIZING MODE: 0 = NORMALIZING 1 = WARNING
(6) WHENEVER ONE OF THESE BITS IS SET TO ZERO, THE INTERRUPT-ENABLE BIT MUST ALSO BE SET TO ZERO.

Figure 6-1. 8087 Control Word Format for Fortran-86 121570-4

STORE/LOAD 8087 Control Word. STCW87 and LDCW87 are used to change computation modes and exception masks. They may also be used separately to examine current settings or to set all options simultaneously. The following example changes the default “”warning” mode to “”normalizing” mode for denormalized operands, and illustrates the use of these intrinsics:

```
INTEGER*2    ICW87
CALL STCW87(ICW87)
ICW87 = ICW87 .AND. #FF7DH
CALL LDCW87 (ICW87)
```

See Figure 6-1 for an overview and explanation of the 8087 control word. Other control-word changes can be done in a similar manner.

STORE 8087 Status Word. STSW87 returns the current exception status of the floating-point processor in the form of an INTEGER*2 bit string (see Figure 6-2), and clears the 8087 exception flags. The 8087 status word represents an accumulation of all masked floating-point exceptions that have occurred since execution of the

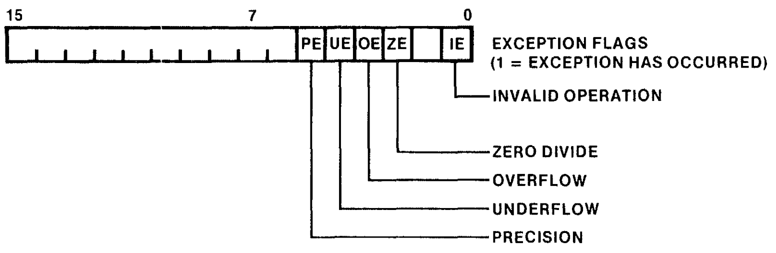


Figure 6-2. 8087 Status-Word Format for Fortran-86 (STSW87) 121570-5

program began or since the last time STSW87 was executed. The following example tests to see if a masked overflow exception has occurred, and illustrates the use of this intrinsic:

```
      INTEGER*2 ISW87
      CALL STW87(ISW87)
      IF (ISW87 .AND. 00008H) 20,10,20
20    (exception has occurred)
10
```

SAVE/RESTORE 8087 State. SAV87 and RST87 transfer the state of the 8087 Numeric Data Processor to and from memory. These functions are useful for preserving the context of the processor for reentrancy, or for determining the cause of a floating-point exception and recovering from it. Both intrinsics operate on a 47-word buffer in memory. The format of the 8087 state information present in that buffer is defined in the 8087 Numeric Supplement. The following is an example of these functions:

```
CHARACTER*94 STATE
CALL SAVE87(STATE)
      .
      .
      .
CALL RST87(STATE)
```

6.3.2.11 8086 Interrupt Control Intrinsics

SETINT establishes a current interrupt procedure and associates it with an interrupt number. Its syntax is as follows:

CALL SETINT(*num*,*name*)

where

num	is an integer expression that is the interrupt number.
name	is the name of the external procedure.

Interrupt procedures may be written in any language that supports 8086 interrupts. See Chapter 11 for more information on the INTERRUPT compiler control.

Example:

```
$INTERRUPT
  SUBROUTINE INTRPT
        .
        .
  (INTERRUPT PROCESSING CODE)
        .
        .
  RETURN
  END

  PROGRAM main
        .
  EXTERNAL INTRPT
        .
  CALL SETINT(8, INTRPT)
        .
        .
        .
```

When an interrupt occurs, the hardware automatically disables further interrupts, and enables them again at termination of the procedure. ENABLE/DSABLE provide the user with additional flexibility both within and outside of interrupt procedures.

DSABLE

```
CALL DSABLE
```

DSABLE disables the 8086 interrupt mechanism (prevents interrupts from occurring) until it is ENABLEd again.

ENABLE

```
CALL ENABLE
```

ENABLE enables the 8086 interrupt mechanism.

NOTE

Do not use the interrupt procedures in any program that will run in an iRMX 86 environment. Instead use the iRMX 86 system calls to set up interrupt processing routines.

6.3.3 Statement Functions

A *statement function* is a user-defined internal function that calculates a mathematical value. All statement-function definitions must follow all specification statements and must precede all executable statements. Its syntax is as follows:

name([*arg*, [*arg*, ...]]) = *exp*

where

name	is the symbolic name you give your statement function.
arg	is a dummy argument that becomes associated with an actual argument when the function is referenced.
exp	is an expression.

The statement-function name and its expression can be of different result types. Table 6-12 shows the implicit type conversions of statement functions.

Each operand in the statement-function expression must be one of the following:

- One of the dummy arguments, *arg*
- A constant
- A variable reference
- An array-element reference
- An intrinsic-function reference
- A statement-function reference
- An external-function reference
- A dummy-procedure reference

The symbolic name of a statement function is local and cannot be a symbolic name in any specification statement, except a type statement. You cannot use the name in an EXTERNAL statement or as an actual argument.

The dummy argument list indicates the order, number and type of arguments for the statement function. These dummy argument names have the scope of the statement function only, and each name can appear only once in the dummy argument list. The type of the dummy argument name is the same as it would be if you used it outside the statement function.

You can use the dummy argument name to identify other dummy arguments of the same type in other statement-function statements. You can also use it to identify a variable of the same type within the same program unit, but they have no other relationship.

You reference a statement function by specifying its symbolic name, with all required actual arguments. For example:

```
DATA A,B,C/10.0,10.0,3.8/
FSUM(X)=A * (X**2) + B*X+C
TOTAL = 33.0 + FSUM(3.0)
```

Table 6-12. Implicit Type Conversions in Statement Functions

Statement Function Type		Type Conversion

INTEGER	INTEGER
	INTEGER*1
	INTEGER*2
	INTEGER*4
		INT(exp)
		INT1(exp)
		INT2(exp)
		INT4(exp)

REAL	REAL
	REAL*4
	REAL*8
		REAL(exp)
		REAL(exp)
		DBL(exp)

DOUBLE PRECISION	DOUBLE PRECISION	DBLE(exp)
TEMPREAL	TEMPREAL	TREAL(exp)

Complex	COMPLEX*8
	COMPLEX*16
	COMPLEX
		CMPLX(e)
		DCMPLX(e)
		CMPLX(e)

In this example, Fortran substitutes the value 3.0 for every occurrence of X in the function definition. At the end of the operation, the value of TOTAL is 156.8.

You can reference a statement function only in the program unit where you defined it. A statement function in a FUNCTION subprogram cannot reference the name of the subprogram. A reference to an external function in the expression of a statement function must not cause a dummy argument of the statement function to become undefined or redefined.

6.3.4 The % VAL Function

The % VAL function is a Fortran-86 extension which enables parameter passing by value for program linkage with non-Fortran programs. It can appear only as an actual argument in external function or subroutine references, or as a dummy argument in SUBROUTINE or FUNCTION statements. The % VAL function accepts only one actual argument, which must be a constant, variable, array element, or expression of type INTEGER or LOGICAL.

For INTEGER arguments passed to a subroutine or function, the length of the argument passed is determined by the argument length, not by the default length. This allows you to explicitly control the argument length using the functions INT1, INT2, or INT4.

For variables, function references, and symbolic constants, the length is the explicitly or implicitly defined length of the name. For numeric constants, the length is the minimum length required to contain the value.

6.4 BLOCK DATA Subprograms

A BLOCK DATA subprogram initializes variables and array elements in named COMMON blocks using DATA statements. The first statement of any BLOCK DATA subprogram must be a BLOCK DATA statement (see Section 4.2.4). The last statement must be the END statement. You can use only IMPLICIT, DIMENSION, named COMMON, SAVE, EQUIVALENCE, DATA, and type statements in a BLOCK DATA subprogram.

The name of a BLOCK DATA subprogram is optional. If you do name the subprogram, that name is global and cannot be the same as the name of any external procedure, main program, COMMON block, or other BLOCK DATA subprogram. You can have at most one unnamed BLOCK DATA subprogram per executable program.

Only variables appearing in named COMMON statements can appear in a DIMENSION, EQUIVALENCE, DATA, SAVE, or type statement in a BLOCK DATA subprogram.

If you initialize a named COMMON block, you must list all the variables in that block even if you are not initializing all of them. You can initialize variables in more than one named COMMON block in a single BLOCK DATA subprogram, but you cannot specify the same named COMMON block in more than one BLOCK DATA subprogram.

CHAPTER 7 EXPRESSIONS

Expressions in Fortran consist of symbols, constants, and operators (including parentheses) which perform specified operations. There are four types of expressions in Fortran:

- Arithmetic
- Relational
- Logical
- Character

Fortran-86 also allows bitwise Boolean operations using logical operators on INTEGER values.

7.1 Arithmetic Expressions

An arithmetic expression performs a numeric computation. The range and precision of numeric values representable in Fortran restricts the range and precision of the results.

Table 7-1 lists the arithmetic operators and their meanings.

The types of the operands in an arithmetic expression determine the type of the result. An operand may be type INTEGER, REAL, DOUBLE PRECISION, COMPLEX, COMPLEX*16, or TEMPREAL. When a plus sign (+) or a minus sign (−) precedes a single operand, the type and length of the result are those of the operand. Table 7-2 shows the data type and length of the result when an arithmetic operator joins a pair of operands. The type rules are the same for all operators.

For mixed-mode arithmetic, Fortran converts both operands to the same type (the type of the result) before doing the operation, except for an INTEGER exponent which is sometimes used for repeated multiplications (see Table 7-3).

In an arithmetic expression with operands of type REAL, DOUBLE PRECISION, COMPLEX, COMPLEX*16, or TEMPREAL, Fortran converts the operands to type TEMPREAL and the result of the expression is type TEMPREAL. However, when the result is an actual argument to a function or subroutine, Fortran rounds it to the type specified in Table 7-2.

Table 7-1. Arithmetic Operators

Operator	Meaning
**	Exponentiation
/	Division
*	Multiplication
+	Unary or Binary Addition
−	Unary or Binary Subtraction

Table 7-2. Type and Length of Arithmetic Expressions (Addition, Subtraction, Multiplication, Division and Exponentiation)

OP 1 \ OP 2	INTEGER*1	INTEGER*2	INTEGER*4	REAL*4	REAL*8	DOUBLE PRECISION	TEMPREAL
INTEGER*1	INTEGER*1	INTEGER*2	INTEGER*4	REAL*4	REAL*8	DOUBLE PRECISION	TEMPREAL
INTEGER*2	INTEGER*2	INTEGER*2	INTEGER*4	REAL*4	REAL*8	DOUBLE PRECISION	TEMPREAL
INTEGER*4	INTEGER*4	INTEGER*4	INTEGER*4	REAL*4	REAL*8	DOUBLE PRECISION	TEMPREAL
REAL*4	REAL*4	REAL*4	REAL*4	REAL*4	REAL*8	DOUBLE PRECISION	TEMPREAL
REAL*8	REAL*8	REAL*8	REAL*8	REAL*8	REAL*8	REAL*8	TEMPREAL
DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	TEMPREAL
TEMPREAL	TEMPREAL	TEMPREAL	TEMPREAL	TEMPREAL	TEMPREAL	TEMPREAL	TEMPREAL
COMPLEX*8	COMPLEX*8	COMPLEX*8	COMPLEX*8	COMPLEX*8	COMPLEX*16	COMPLEX*16	COMPLEX*16
COMPLEX*16	COMPLEX*16	COMPLEX*16	COMPLEX*16	COMPLEX*16	COMPLEX*16	COMPLEX*16	COMPLEX*16

For example, Fortran computes the expression that follows:

```
INTEG1 + REAL4
```

as though you had written the following:

```
TREAL(INTEG1) + TREAL(REAL4)
```

with a result of type TEMPREAL, unless the result is an argument in a function or subroutine reference. In that case, Fortran computes the following:

```
REAL(TREAL(INTEG1) + TREAL(REAL4))
```

NOTE

When one INTEGER divides into another INTEGER, Fortran truncates the result towards zero. For example:

The value of 1/3 is 0
The value of 8/3 is 2
The value of −8/3 is −2

INTEGER division by zero always causes invocation of the current error handler and cannot be masked. INTEGER overflow results in unsigned arithmetic modulo 2**(8*n), where n is the size in bytes of the expression result.

Table 7-3. Evaluation Methods For YX**

X	Y	Evaluation Method
INTEGER INTEGER (0<X<64)	INTEGER, REAL, DOUBLE PRECISION, or TEMPREAL	y*y*y*...[x times]
INTEGER (X>64) REAL	REAL, DOUBLE PRECISION, or TEMPREAL (1)	
DOUBLE PRECISION TEMPREAL	INTEGER, REAL, DOUBLE PRECISION, or TEMPREAL (2)	2**(x * lg 2(y))
INTEGER (X<0)	REAL, DOUBLE PRECISION, or TEMPREAL	1/(y**|x|) If over- or underflow occurs, or Y is unnormal then (1/y)**|x|

NOTES:

1. If y is unnormal, then the first method (multiplication) is used.
2. If y is negative, then x must be a whole number.

The following floating-point exceptions can occur during the evaluation of arithmetic expressions:

- Division by zero
- Overflow
- Underflow
- Inexact result
- Invalid operation (including INTEGER overflow during conversion from floating-point)

See Section 15.3.6 for details on the causes and handling of these exceptions.

7.2 Character Expressions

A character expression results in a character string. This expression can be a single character operand (character constant, character variable, character array element, character substring, or character function reference), multiple operands joined by the character operator, or another character expression in parentheses. A character expression always returns a value of type CHARACTER.

The only character operator is //, representing concatenation.

The length of the result of a concatenation operation is the sum of the length of the operands. For example, if the following expression:

```
'AB' // 'CDE'
```

appears, the result is the string that follows:

```
'ABCDE'
```

7.3 Relational Expressions

A relational expression compares two arithmetic or two character expressions and returns a TRUE or FALSE value of type LOGICAL.

Table 7-4 lists the relational operators and their meanings.

7.3.1 Arithmetic Relational Expressions

An arithmetic relational expression compares two arithmetic expressions and returns a TRUE or FALSE value of type LOGICAL. Its syntax is as follows:

exp1 relop exp2

where

exp1 and *exp2*	are arithmetic expressions (see Section 7.1).
relop	is any relational operator.

If the operands are of different arithmetic types, they are converted to the same type as that of an arithmetic expression with the same operands (see Section 7.1).

If the operands of a floating-point relational operation are unordered (see Section 7.7, "Floating Point Topics"), then the LOGICAL result is .TRUE. for the .NE. operator but .FALSE. for any other relational operator.

7.3.2 Character Relational Expressions

A character relational expression compares two character expressions and returns a value of TRUE or FALSE of type LOGICAL. Its syntax is as follows:

exp1 relop exp2

where

exp1 and *exp2*	are character expressions.
relop	is any relational operator.

The relative positions of *exp1* and *exp2* in the ASCII collating sequence compared from left to right determines the value of the result unless the operator is either "equal to" (.EQ.) or "not equal to" (.NE.). If the operators are of unequal length, then Fortran extends the shorter operand to the right with blanks so that its length equals the length of the longer operand, and then compares the lengths of the two operands.

Table 7-4. 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

7.4 Logical Expressions

A logical expression performs a logical computation and returns a value of TRUE or FALSE of type LOGICAL. This expression can be a single logical operand (logical constant, logical variable reference, logical array element, logical function reference, or relational expression) or a combination of logical operands joined by logical operators and parentheses. Table 7-5 shows the logical operators and their meanings.

Table 7-5. Logical Operators

Operator	Meaning
.NOT.	Logical Negation
.AND.	Logical Conjunction
.OR.	Logical Inclusive Disjunction
.EQV.	Logical Equivalence
.NEQV.	Logical Nonequivalence

Fortran determines the value of a logical expression using the rules summarized in Tables 7-6 through 7-10.

A logical expression involving .NOT. has the opposite value as its operand as shown in Table 7-6.

Table 7-6. Value of a Logical Expression with .NOT.

OP1	.NOT. OP1
.TRUE.	.FALSE.
.FALSE.	.TRUE.

In a logical expression with .AND., the result is .TRUE. only if both operands are .TRUE., as shown in Table 7-7.

Table 7-7. Value of a Logical Expression with .AND.

OP1	OP2	OP1 .AND. OP2
.TRUE.	.TRUE.	.TRUE.
.TRUE.	.FALSE.	.FALSE.
.FALSE.	.TRUE.	.FALSE.
.FALSE.	.FALSE.	.FALSE.

In a logical expression with .OR., the result is .FALSE. only if both operands are .FALSE., as shown in Table 7-8.

Table 7-8. Value of a Logical Expression with .OR.

OP1	OP2	OP1 .OR. OP2
.TRUE.	.TRUE.	.TRUE.
.TRUE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.TRUE.
.FALSE.	.FALSE.	.FALSE.

In a logical expression with .EQV., the result is .TRUE. only if both operands are logically the same, as shown in Table 7-9.

Table 7-9. Value of a Logical Expression with .EQV.

OP1	OP2	OP1 .EQV. OP2
.TRUE.	.TRUE.	.TRUE.
.TRUE.	.FALSE.	.FALSE.
.FALSE.	.TRUE.	.FALSE.
.FALSE.	.FALSE.	.TRUE.

In a logical expression with .NEQV., the result is .TRUE. only if both operands are logically different, as shown in Table 7-10.

Table 7-10. Value of a Logical Expression with .NEQV.

OP1	OP2	OP1 .NEQV. OP2
.TRUE.	.TRUE.	.FALSE.
.TRUE.	.FALSE.	.TRUE.
.FALSE.	.TRUE.	.TRUE.
.FALSE.	.FALSE.	.FALSE.

Any LOGICAL-data element can occupy one, two, or four bytes. Table 7-11 shows the length of the result of a logical expression. The type rules are the same for all operators. For any logical expression with .NOT., the resulting value has the same length as its operand.

Table 7-11. Length of Results of Logical Expressions (.AND., .OR., .EQV., .NEQV.)

OP1 \ OP2	LOGICAL*1	LOGICAL*2	LOGICAL*4
LOGICAL*1	LOGICAL*1	LOGICAL*2	LOGICAL*4
LOGICAL*2	LOGICAL*2	LOGICAL*2	LOGICAL*4
LOGICAL*4	LOGICAL*4	LOGICAL*4	LOGICAL*4

7.5 Bitwise Boolean Operations

Fortran-86 allows the use of logical operators with INTEGER as well as LOGICAL operands. In this case, the resulting INTEGER value is the bitwise complement, conjunction, inclusive disjunction, equivalence, or exclusive disjunction of the integer operands. This feature is a machine-dependent extension and assumes that the processor represents INTEGER data in two's complement form. If the lengths of the two operands differ, Fortran sign-extends the shorter operand.

If the operator is .NOT., the length of the result is the same as the length of the operand. The length of an expression result with two operands is the same as that for integer arithmetic expressions (see Table 7-2).

Fortran-86 does not allow operations between LOGICAL and INTEGER (Boolean) operands.

7.6 Precedence of Operators

Fortran generally evaluates operators of higher precedence before operators of lower precedence. When two operators have equal precedence, Fortran evaluates the leftmost one first.

The use of parentheses overrides the normal rules of precedence. The part of the expression enclosed in parentheses is evaluated first. With nested parentheses, Fortran evaluates the innermost set first.

The following list shows the precedence of operators in decending order:

- Parenthesized expressions
- Concatenation: //
- Exponentiation: **
- Multiplication or Division: *, /
- Addition or Subtraction (unary and binary): +,−
- Relational Operators: .LT.,.LE.,.EQ.,.NE.,.GT.,.GE.
- Logical or Boolean .NOT.
- Logical or Boolean .AND.
- Logical or Boolean .OR.
- Logical or Boolean .EQV.,.NEQV.

For example, Fortran interprets the expression that follows:

```
D .OR. A + B .GE. C
```

as though you had written the following:

```
D .OR. ((A + B) .GE. C)
```

The only exception to the left-to-right rule is the case where two or more exponentiation terms occur together. For example:

```
A ** B ** C
```

In this case, the compiler interprets the expression from right to left as though you had written the following:

```
A ** (B**C)
```

7.7 Floating-Point Topics

7.7.1 Rounding

In Fortran-86, the default for all implicit rounding (except i=x) is round-to-nearest mode. In this mode, the operand is rounded to the nearest representable value, or to the nearest even number in case of a tie. The rounding mode determines the sign assigned to zero: if x is finite, then x − x = x + (− x) = + 0 in round-to-nearest mode. However, x + x = x − (− x) always has the same sign as x even when x is zero.

7.7.2 Normalized, Denormalized, and Unnormalized Numbers

A value is *normalized* if it is within the range and precision defined for its data type. A *denormalized* number is one whose exponent has underflowed the exponent range of its data type, but which still approximates the intended value by containing zeroes to the left of its fraction digits with a corresponding loss of precision ("gradual underflow"). An *unnormalized* number is one whose exponent is normal, but whose precision is the same as (inherited from) a denormal value. The term *unnormalized* applies only to values in TEMPREAL format, including temporary operands and results of expressions.

7.7.3 Warning Mode

In warning mode, unnormal numbers may appear as the results of operations performed on denormal numbers. Since a denormal number results from underflow in which more precision has been lost than rounding can account for, the unnormal number serves as a reminder that the precision loss has occurred. If an unnormal or denormal number is added to, or subtracted from, a normalized number of greater absolute value, the result is normalized since the precision loss due to the denormalization is now within the range of rounding error.

The following rules apply when at least one operand is not normalized, provided the Invalid Operation exception does not occur. They specify when normalization is to occur and the resulting exponent value if normalization does not occur. Rounding and the handling of overflow and underflow are performed after the assignments shown below. Such rounding and overflow/underflow handling may modify the results. In the following, *x* and *y* are real expressions, and *expon(x)* refers to the exponent of *x*.

Assignment:	(z:=x): expon(z)=expon(x).
Add/subtract:	(z:=x*PLUS/MINUS*y): Let m=max(expon(x), expon(y)). If at least one of the operands having exponent m is normalized, then z is normalized before rounding. Otherwise expon(z)=m.
Multiply:	(z:=x*y): expon(z)=expon(x)+expon(y).
Divide:	(z:=x/y): expon(z)=expon(x)−expon(y)−1 when y is normalized and nonzero.
Comparison:	Comparisons are made as if both operands had been normalized first.
Integer-conversion functions:	The result is as if the argument had been normalized first.
Normalizing Mode. In normalizing mode, action is taken depending on the precision of the operand:	A single precision operand (REAL*4) is normalized and the operation is retired. A double precision operand (REAL 8 or DOUBLE PRECISION) is normalized and the operation is retried. A TEMPREAL operand is set to zero and the operation is retried.

TEMPREAL denormal numbers are indistinguishable from zero. Normalizing mode provides gradual underflow for single and double precision numbers and sharp underflow for TEMPREAL numbers.

7.7.4 Infinity Arithmetic

The representation of infinity is a TEMPREAL value that behaves like an infinite value in floating-point computations. Projective mode is default for Fortran-86. In projective mode, $+\infty = -\infty$, and compares unordered with everything but itself. In affine mode, $-\infty < 0 < +\infty$ and each infinity is ordered with respect to everything else except NaN's.

The sign bit of the product or quotient of two floating-point numbers is the exclusive OR of the operands' sign bits, even when the operand is zero or infinite.

7.7.5 Unordered Relation

Four mutually exclusive relations are possible between two arithmetic operands: "less than", "equal to", "greater than", and "unordered". The last case arises when at least one operand is a NaN, or when infinity in the projective mode is compared to anything except infinity.

Note that, in the projective mode, positive infinity is equal to negative infinity, and that positive zero is always equal to negative zero in any mode.

7.7.6 Not a Number (NaN)

When a masked invalid operation error occurs for any operation delivering a floating-point value, the result is Not a Number, or a NaN. Program execution proceeds normally until the NaN appears as an operand for a further floating-point operation, when Fortran-86 takes action depending on the data type of the result. A non-trapping NaN is one whose leading fraction bit is 1; a trapping NaN has a leading fraction bit of 0.

Floating-Point Result: The NaN itself becomes the result and execution continues normally without signalling further errors. If both operands are NaNs, the NaN with the larger magnitude is returned. Thus each NaN is propogated through later floating-point calculations until it is ultimately either ignored, or referenced by operations delivering non-floating-point results.

Formatted Output: On formatted output using an F, E, D, or G edit descriptor, a field of periods (...) is written to indicate an undefined (NaN) result. The A, Z, or B descriptor results in the ASCII, hexadecimal, or binary interpretation, respectively, of the internal representation of the NaN. No error is signalled for output of a NaN.

LOGICAL Result: By definition, a NaN has no ordinal rank with respect to any other operand, even itself. Tests for equality (.EQ.) and inequality (.NE.) are the only Fortran relational operations for which results are defined (.FALSE. and .TRUE., respectively) for unordered operands so program execution continues without error. Any other operator yields an undefined result when applied to NaNs causing an invalid operation error. The masked result is always .FALSE.

The arithmetic IF belongs to the latter category, with falling through to the next statement being the only logically consistent masked response possible.

INTEGER Result: Since no internal NaN representation exists for the INTEGER data type, an invalid-operation error is normally signalled. The masked result is the highest magnitude negative integer for INTEGER*4 or INTEGER*2. An INTEGER*1 result is the value of an INTEGER*2 intermediate modulo 256.

7.7.7 Trapping NaN

A trapping NaN is an explicitly created NaN whose function is to signal an invalid-operation exception (trap) whenever it is used in a computation, comparison, or conversion. Contrary to ordinary NaN's, which are tolerated during most floating-point operations, trapping NaN's cause exceptions virtually every time they are referenced.

If the invalid-operation exception is masked, trapping NaN's behave like ordinary NaN's.

For more information on trapping NaN's, see the *iAPX 86, 88 User's Manual*. Note that the 8087 hardware treats all NaN's as trapping NaN's.

CHAPTER 8 EXECUTABLE STATEMENTS

There are two categories of Fortran statements: nonexecutable and executable. Nonexecutable statements define the characteristics or initial values of data, or define program units. These statements are described in previous chapters. Executable statements do calculations, control program execution, and read or write data from external media. The executable statements for doing calculations and controlling program execution are described in this chapter. The I/O statements are described in Chapter 9, "Input and Output."

8.1 Assignment Statements

Assignment statements give values to variables, arrays, or array elements. There are three kinds of assignment statements:

- Arithmetic
- Logical
- Character

8.1.1 Arithmetic Assignment Statements

An arithmetic assignment statement resembles a conventional arithmetic formula. Its syntax is as follows:

name = *exp*

where

name	is the *name* you give to a variable, array, or array element.
exp	is an arithmetic expression.
=	means "is assigned the value" rather than "is equal to". Therefore, the statement that follows: `I = I + 1` is legal in Fortran.

Execution of an arithmetic assignment statement causes Fortran to evaluate *exp* according to the rules for arithmetic expressions (see Table 7-2). It then converts the result to the type of *name* and assigns it to *name*. Table 8-1 shows this process for different Fortran-86 data types. In Table 8-1, the functions in the CONVERSION column are the generic type conversion functions described in Section 6.1.2.2, "Intrinsic Functions."

If the length of *name* is longer than the result of *exp*, Fortran converts the length of result to the length of *name* while preserving its value.

If the value of *exp* is too large for *name*, the result depends on the data types of *name* and *exp*. If *name* has a floating-point data type, a floating-point overflow exception occurs. If *name* is of type INTEGER*n and *exp* floating point, then an invalid-operation exception occurs for $n = 2$ or 4, and for $n = 1$ if the result overflows INTEGER*2 as well. In all other cases, unsigned integer assignment modulo 2**(8*n) takes place.

See Section 15.3.6 for a description of floating-point exceptions and their handling.

Table 8-1. Type Conversions in Arithmetic Assignment Statements

Type of Target Variable	Type Conversion
INTEGER	INT(exp)
INTEGER*1	INT1(exp)
INTEGER*2	INT2(exp)
INTEGER*4	INT4(exp)
REAL	REAL(exp)
REAL*4	REAL(exp)
REAL*8	DBLE(exp)
DOUBLE PRECISION	DBLE(exp)
TEMPREAL	TREAL(exp)

*Note that conversions on COMPLEX data in arithmetic assignment statements take place on the component data types.

8.1.2 Character Assignment Statements

The character assignment statement assigns a character value to a character variable or array element. Its syntax is as follows:

name = *exp*

where

name	is the name you give to a character variable or character array element. Name may be a substring, but must not contain or overlap any string referenced in expression.
exp	is a character expression (see Section 7.2).

The two sides of a character assignment statement can have different lengths. If *name* is longer than the result of *exp*, Fortran pads the result on the right with blanks. If *name* is shorter than the result of *exp*, Fortran truncates *exp* on the right until it fits into *name*.

8.1.3 Logical Assignment Statements

The logical assignment statement assigns the value .TRUE. or .FALSE. to a logical variable or array element. Its syntax is as follows:

name = *exp*

where

name	is the name you give to a logical variable or logical array element.
exp	is a logical expression (see Section 7.4).

8.2 IF Statements

An IF statement transfers control from one part of the program to another under certain specified conditions. It can also provide alternative actions for the program to perform if these conditions are not met. There are three basic IF constructs:

- Block IF
- Logical IF
- Arithmetic IF

8.2.1 Block IF

A block IF construct is introduced by a block IF statement, and terminated by an END IF statement. The intervening statements form the IF block, any number of ELSE IF blocks, and at most one ELSE block, in that order. The first statement of each of these blocks must be IF, ELSE IF, or ELSE statements, respectively; the block is terminated by the next ELSE IF, ELSE, or END IF statement.

These blocks can be nested. For example, an IF block may contain another IF block, which may contain another IF block, etc. These blocks can also be empty, meaning that there need not be any executable statements between the first statement of a block and its corresponding terminating statement.

You cannot transfer control into an IF, ELSE IF, or ELSE block from outside the IF block.

Figure 8-1 illustrates a possible nesting of IF, ELSE IF, and ELSE blocks.

8.2.1.1 Block IF Statement

The block IF statement introduces an IF block and must be the first statement of that block. Its syntax is as follows:

```
IF(exp)THEN
```

where

exp	is a logical expression (see Section 7.4). If the value of *exp* is true, Fortran executes the statements of the IF block. As soon as an ELSE IF or ELSE statement on the same nesting level as the block IF is encountered, control passes to the END IF statement of the block IF statement. If *exp* is false, Fortran passes control to the first ELSE IF, ELSE, or END IF statement on the same nesting level as the block IF statement.

Each block IF statement must have a corresponding END IF statement in the same program unit.

8.2.1.2 ELSE IF Statement

The ELSE IF statement introduces an ELSE IF block and must be the first statement in that block. Its syntax is as follows:

```
ELSE IF(exp)THEN
```

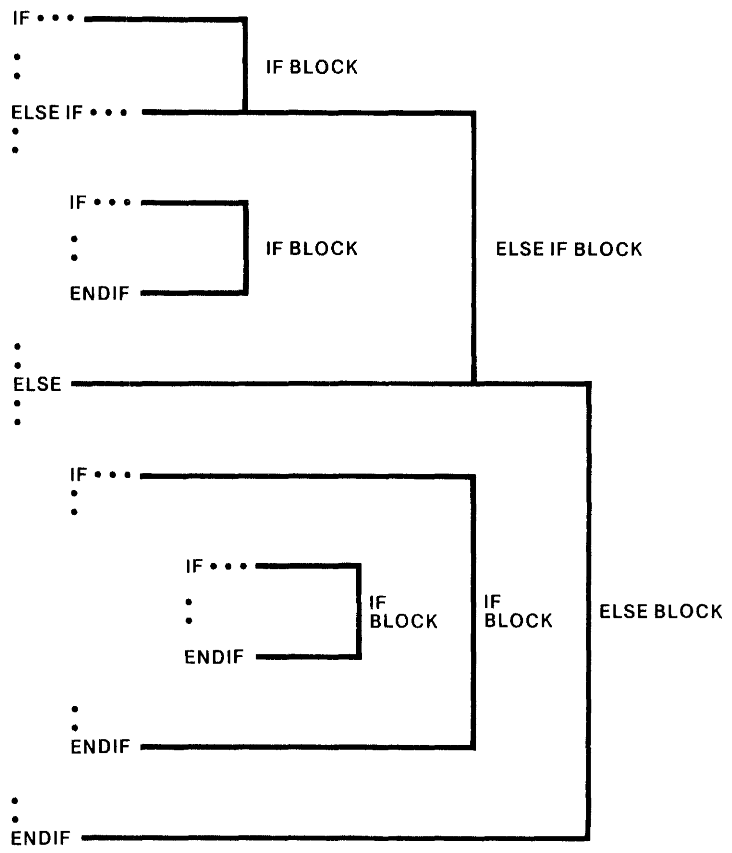


Figure 8-1. Nesting Levels of IF, ELSE IF, and ELSE Blocks 121570-6

where

exp	is a logical expression. If *exp* is true, execution continues with the first statement of the ELSE IF block. If *exp* is false, Fortran passes control to the next ELSE IF, ELSE, or END IF statement on the same nesting level as the ELSE IF statement.

An ELSE block must be immediately preceded by an IF or another ELSE IF block of the same nesting level and is terminated by another ELSE IF, ELSE, or END IF statement. No statement can reference the statement label of an ELSE IF statement.

8.2.1.3 ELSE Statement

An ELSE statement introduces an ELSE block. Its syntax is as follows:

```
ELSE
```

An ELSE block must be immediately preceded by an IF or ELSE IF block, and is terminated by the END IF statement.

No statement can reference the statement label of an ELSE statement.

8.2.1.4 END IF Statement

The END IF statement terminates the last IF, ELSE IF, or ELSE block of a block IF construct. Its syntax is as follows:

```
END IF
```

Each block IF statement must have a corresponding END IF statement.

8.2.2 Logical IF Statement

The logical IF statement executes a statement in the program depending on the value of a controlling expression. Its syntax is as follows:

IF(*exp*)*stmt*

where

exp	is a logical expression.
stmt	is any executable statement except a DO or another IF statement.

If *exp* is true, Fortran executes *stmt* next. If it is false, Fortran executes the statement following the logical IF and ignores *stmt*.

A function reference in the controlling logical expression can affect the operands in *stmt*.

8.2.3 Arithmetic IF Statement

The arithmetic IF statement transfers control of the program to one of four possible statements depending on the value of a controlling expression. Its syntax is as follows:

IF(*exp*)*s1*,*s2*,*s3*

where

exp	is any expression (see Section 7.1). If the value of *exp* is less than zero, control passes to the first statement listed. If *exp* equals zero, control passes to the second statement. If *exp* is greater than zero, control passes to the third statement. If the result of *exp* is unordered (see Chapter 7), control continues with the next executable statement following the arithmetic IF statement.
s1, *s2*, and *s3*	are statement labels of any executable statements in the same program unit as the arithmetic IF. The same statement label can appear more than once in the same arithmetic IF statement.

8.3 DO Statement

Frequently, you will want to repeat a series of operations several times. Rather than copy the statements that perform these operations many times, you can create a loop that causes the program to perform the same statements over and over a specified number of times. This is the concept of a DO loop. The DO statement introduces and defines a DO loop. Its syntax is as follows:

```
DO stl[,] var=e1,e2[,e3]
```

where

stl	is the statement label of an executable statement that is the last statement in the DO loop.
var	is an integer variable that acts as the index value of the DO loop.
e1, *e2*, and *e3*	are integer expressions. In this format, *e1* is the initial index value, *e2* is the loop termination value, and *e3* is the optional loop increment/decrement value. If you do not specify *e3*, the compiler assumes an increment of one. The values of *e1* and *e2* may be such that no iterations are performed. (See Section 11.4.3, DO66 DO77 Controls for details.)

The last statement of a DO loop must not be an unconditional GO TO, assigned GO TO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement. If the last statement of the DO loop is a logical IF statement, it can contain any executable statement except a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF statement.

DO loops can be nested. For example, a DO loop can contain another DO loop which can contain another DO loop, etc. If a DO statement appears within the range of another DO loop, the entire inner DO loop must be within the range of the outer DO loop. DO loops can share the same last statement.

If a DO statement appears within an IF, ELSE IF, or ELSE block, the range of the DO loop must be entirely within that block.

If a block IF statement is within the range of a DO loop, its corresponding END IF statement must also be within the range of the DO loop.

You cannot transfer program control into a DO loop.

8.4 CONTINUE Statement

The CONTINUE statement has no effect on program execution. Execution simply continues with the next executable statement. Its syntax is as follows:

```
CONTINUE
```

8.5 CALL Statement

The CALL statement invokes a subroutine. The main program or any subprogram can reference a subroutine using the CALL statement. Its syntax is as follows:

```
CALL name[([arg[,arg]...])]
```

where

name	is the name of the subroutine.
arg	is an actual argument. The actual arguments in the CALL statement must agree in order, number, type, and length with the corresponding dummy argument list of the referenced subroutine. (See Section 6.1 for a complete description of subroutines and arguments.)

8.6 RETURN Statement

The RETURN statement transfers control back to the calling program unit. Its syntax is as follows:

```
RETURN
```

The RETURN statement may appear only in FUNCTION or SUBROUTINE subprograms. These subprograms may have one or more RETURN statements, or none at all. An END statement terminating such a program unit has the same effect as a RETURN statement.

When Fortran executes a RETURN statement in a FUNCTION subprogram, a return value of the function must already have been defined.

When Fortran executes a RETURN statement, it terminates the association between the dummy arguments of the procedure and the current actual arguments (see Section 6.1, "Subroutines and Functions").

8.7 ASSIGN Statement

The ASSIGN statement is the only way you can assign a statement label to a symbolic name. A GO TO statement or a format identifier in an I/O statement can then reference this symbolic name. To use the symbolic name in another context, you must redefine it with an integer value in an arithmetic assignment statement. Its syntax is as follows:

```
ASSIGN stl TO name
```

where

stl	is a statement label. The statement label must be the label of an executable statement or a FORMAT statement in the same program unit as the ASSIGN statement.
name	is an integer variable name. You cannot declare *name* to be of length INTEGER*1.

8.8 GO TO Statements

The GO TO statements pass program control to another part of the program, either conditionally or unconditionally. There are three GO TO statements:

Unconditional GO TO
Computed GO TO
Assigned GO TO

8.8.1 Unconditional GO TO Statement

The unconditional GO TO statement transfers control to a specified statement. Its syntax is as follows:

GO TO *stl*

where

stl	is a statement label of an executable statement in the same program unit as the GO TO statement.

8.8.2 Computed GO TO Statement

The computed GO TO statement branches to one of several executable statements based on the value of a controlling expression. Its syntax is as follows:

GO TO(*stl* [, *stl*]...) *exp*

where

stl	is the statement label of an executable statement in the same program unit as the computed GO TO statement. The same statement label can appear more than once in the same computed GO TO statement.
exp	is an integer expression. If *exp* has a value in the range $1 \leq exp \leq n$ (where *n* is the number of statement labels in the list), control passes to the statement that corresponds to this value. If *exp* is outside of this range, execution continues with the statement following the GO TO and all the statement labels in the list are ignored.

8.8.3 Assigned GO TO Statement

The assigned GO TO statement transfers control to one of several executable statements based on an integer variable name. You use it with the ASSIGN statement. Its syntax is as follows:

GO TO *name* [(*stl*[, *stl*]...)]

where

name	is an integer variable name. Before the assigned GO TO statement can be executed, an ASSIGN statement in the same program unit must have defined the variable name with the value of a statement label.
stl	is the statement label of an executable statement in the same program unit as the assigned GO TO statement. The same statement label may appear more than once in the statement. If the parenthesized list of statement labels is present, the statement label assigned to *name* must be one of the labels in the list.

8.9 Program Halt Statements

Fortran provides the following three statements for halting or terminating program execution:

- PAUSE
- STOP
- END

For details on the END statement, see Section 4.3.

8.9.1 PAUSE Statement

The PAUSE statement suspends program execution and allows execution to continue or terminate depending on an external signal. Its syntax is as follows:

```
PAUSE[msg]
```

where

msg	is either a string of not more than five digits or a character constant.

When the PAUSE statement is executed, a message in the following form:

```
***PROGRAM PAUSE. [msg]
```

is written to the file connected to Unit 6 (see Section 14.5, "Preconnecting Files"), and program execution is suspended. By entering anything starting with an S (either upper or lower case) on Unit 5, the operator can cause execution of the program to terminate; any other input causes execution to continue with the statement following the PAUSE statement.

8.9.2 STOP Statement

The STOP statement terminates program execution from within a program. Its syntax is as follows:

```
STOP [msg]
```

where

msg	is either a string of not more than five digits or a character constant.

When the STOP statement is executed, a message in the form

```
***PROGRAM STOP. [msg]
```

is written to the file connected to Unit 6 (see Section 14.5), and program execution is terminated.

The STOP statement is intended as a means to terminate program execution abnormally, that is, to inform the operator of a special program-detected condition that makes further execution undesirable or impossible. For normal program termination, execution of the END statement of the main program is preferred for reasons of run-time efficiency.

CHAPTER 9
INPUT AND OUTPUT

Fortran input/output statements direct the transfer of data between the processor and some external unit or within the processor itself. There are two categories of statements: file handling and data transfer. The file-handling statements connect and disconnect, position, and mark the end of files. The data-transfer statements supply the external or internal unit and the list of input or output variables including any necessary formatting information. This chapter describes each of these statements.

9.1 Records, Files and Units

The following sections provide information on records, files, and units.

9.1.1 Records

A record is a logically related set of values or characters. There are two types of records: formatted and unformatted.

A formatted record is a sequence of ASCII printable characters. An unformatted record is a sequence of values containing any combination of data types. Only formatted and unformatted I/O statements, respectively, can read or write these records.

9.1.2 Files

A file is a sequence of records. There are two kinds of files: external and internal.

9.1.2.1 External Files

An external file is stored on an external unit, such as a line printer or flexible disk. You can access an external file in one of two ways: sequentially or directly.

A sequential-access file has the following characteristics:

- The file consists of a sequence of variable-length records.
- The records are all accessed in the same order as they were created.
- The records are either all formatted or all unformatted.
- You can read from or write to the files using only sequential-access I/O statements.

A direct-access file has the following characteristics:

- All the records have the same length.
- You can read from or write to the file in any order.
- The records are either all formatted or all unformatted.
- You can read from or write to the file using only direct-access I/O statements.
- Each record has a unique record number determined when the record was created. You may not delete a record or change its number. You can rewrite an existing record.

9.1.2.2 Internal Files

An internal file is a character variable, character array, or character array element. Using internal files, you can transfer and format data within processor memory.

An internal file has the following characteristics:

- Each record is a character variable or array element.
- The size of the file depends on the kinds of records in the file. If the file is a character variable or array element, it is a single record whose length is that of the variable or array element. If it is a character array, every record has the same length as an array element in that array and the file has as many records as the array has elements.

You cannot reference an internal file in a file-handling statement. You can use only sequential-access, formatted I/O statements that do not specify list-directed formatting.

9.1.3 Units

A unit is a logical way of referring to a file. A unit can be connected or disconnected. All I/O statements, except OPEN and CLOSE, must reference a unit connected to a file.

You can connect a file to a unit using the OPEN statement and disconnect the file using the CLOSE statement. Depending on the operating environment, some units may be preconnected and you can reference them in I/O statements without first using an OPEN statement. A preconnected file becomes connected the first time an I/O statement references it.

For example, in the Series-III operating system environment, the console output device and console input device are always preconnected for unit numbers 6 and 5 respectively, but you can override these defaults by preconnecting the units explicitly (see Section 14.5).

A unit cannot be connected to more than one file at a time and vice versa. The only way to refer to a disconnected file is by naming it in an OPEN statement. Consequently, an unnamed file cannot be reconnected once it has been disconnected.

9.2 File-Handling Statements

Fortran provides five file-handling statements: OPEN, CLOSE, BACKSPACE, REWIND, and ENDFILE. These statements are valid for external files only.

9.2.1 OPEN

The OPEN statement can connect an existing file to a unit, create a preconnected file, create a file and connect it to a unit, or change certain specifiers in an existing file/unit connection. Its syntax is as follows:

OPEN(*open-list*)

where

open-list is a list of specifiers separated by commas. The list of specifiers is as follows:

[UNIT=]*unit*	Unit specifier
IOSTAT=*stname*	I/O status specifier
ERR=*stl*	Error specifier
FILE=*name*	File-name specifier
STATUS=*stat*	File-status specifier
ACCESS=*acc*	Access-method specifier
FORM=*fmat*	Formatting specifier
RECL=*reclen*	Record-length specifier
BLANK=*blank*	Blank specifier
CARRIAGE=*car*	Carriage-control specifier

The unit specifier, *unit*, must be present. All of the other specifiers are optional except that if you connect a file for direct access, the record-length specifier must be present. Some specifiers have default values. The following sections describe each of the specifiers in detail.

9.2.1.1 Unit Specifier

The format of the unit specifier is as follows:

[UNIT=]*unit*

where

unit is an integer value between 0 and 255 that identifies an external file. If you omit the optional UNIT = , *unit* must be the first item in *open-list*.

Examples:

```
OPEN(UNIT=3)
OPEN(4)
```

9.2.1.2 I/O Status Specifier

The format of the I/O status specifier is as follows:

IOSTAT=*stname*

where

stname is an integer variable or integer array-element name. The variable must be INTEGER*2.

If no error occurs, executing an I/O statement with this specifier causes *stname* to be assigned a zero value. If an error does occur, *stname* is assigned an error message number (see Section 15.3, "Run-Time Errors").

Example:

```
OPEN(4,IOSTAT=ERRFLG)
```

9.2.1.3 Error Specifier

The format of the error specifier is as follows:

ERR=*stl*

where

stl	is the statement label of an executable statement in the same program unit as the I/O statement.

If an error occurs during execution of the I/O statement, the following steps occur:

1. The I/O operation terminates.
2. The position of the file specified by the I/O statement becomes indeterminate.
3. If the I/O statement has an IOSTAT specifier, Fortran sets *stname* to reflect the error condition.
4. Execution continues with the statement named by the ERR specifier. If you did not specify ERR, a run-time error occurs.

Example:

```
OPEN(4,IOSTAT=ERRFLG,ERR=200)
```

9.2.1.4 File-Name Specifier

The format of the file-name specifier is as follows:

FILE=*name*

where

name	is the name of the file expressed as a character constant enclosed in quotation marks or a variable. It must be a valid file name for the operating environment. If you omit FILE, the unit is connected to a scratch file (:WORK:) unless it was previously associated with a specific file (i.e., in a preconnection). A filename cannot be specified if STATUS = 'SCRATCH' is specified.

Example:

```
OPEN(UNIT=3,FILE='MYPROG.FIL')
```

9.2.1.5 File-Status Specifier

The format of the file-status specifier is as follows:

STATUS=*stat*

where

stat	is a character expression evaluating to 'OLD', 'NEW', 'SCRATCH', or 'UNKNOWN'. If you omit the STATUS specifier, the default value is UNKNOWN. If you specify OLD or NEW, the FILE specifier must also be present or the file must be preconnected. When you specify SCRATCH, a temporary file is connected to the specified unit for the duration of program execution or until you issue a CLOSE statement for the same unit and then delete it. You cannot specify SCRATCH with a named file. If you specify UNKNOWN, the file status is environment-dependent. In the Series-III environment, UNKNOWN is allowed only for a named file. In this case, it is equivalent to OLD if the file exists and NEW if it does not.

Example:

```
OPEN(3,FILE='MYPROG.FIL',STATUS='NEW')
```

9.2.1.6 Access-Method Specifier

The format of the access-method specifier is as follows:

ACCESS=*acc*

where

acc	is a character expression evaluating to either 'SEQUENTIAL' or 'DIRECT' (see Section 9.1.2.1). If you omit the ACCESS specifier, the default is SEQUENTIAL.

If the file already exists, the specified access method must match the characteristics of that file. For example, iRMX 86 physical files are by definition sequential files and must be opened for sequential access only. New files are created with the specified access method. If the access method is DIRECT, the record-length specifier must be present in the specifier list.

Example:

```
OPEN(3,FILE='MYPROG',STATUS='NEW',
&ACCESS='SEQUENTIAL')
```

9.2.1.7 Formatting Specifier

The format of the formatting specifier is as follows:

FORM=*fmat*

where

fmat	is a character expression evaluating to 'FORMATTED' or 'UNFORMATTED'. If you omit the FORM specifier, the default is UNFORMATTED if you connect the file for direct access and FORMATTED if you connect the file for sequential access.

Example:

```
OPEN(3,FILE='MYPROG.FIL',STATUS='NEW',
&ACCESS='SEQUENTIAL',FORM='FORMATTED')
```

9.2.1.8 Record-Length Specifier

The format of the record-length specifier is as follows:

RECL=*reclen*

where

reclen is a positive integer expression that evaluates to the length of each record of the file being connected for direct access.

If you connect the file for formatted I/O, *reclen* is the number of characters. If you connect the file for unformatted I/O, *reclen* is the number of bytes.

You must include the RECL specifier in the OPEN statement when you connect the file for direct access.

When you connect the file for sequential access, the run-time system uses the record length to pad formatted input records with blanks to simplify data entry using a terminal. The RECL specifier is ignored for formatted sequential output and invalid for unformatted sequential I/O.

Example:

```
OPEN(3,FILE='MYPROG.FIL',STATUS='NEW',
&ACCESS='DIRECT',FORM='FORMATTED',RECL=80)
```

9.2.1.9 Blank Specifier

The format of the blank specifier is as follows:

BLANK=*blank*

where

blank is one of the character constants 'NULL' or 'ZERO'. If you omit the BLANK specifier, the default value is NULL.

If you specify NULL, Fortran ignores all blanks in numeric formatted input fields, except that a field of all blanks has the value zero. If you specify ZERO, all blanks, except leading blanks, have the value zero.

You can use this specifier only for formatted I/O.

Example:

```
OPEN(3,FILE='MYPROG.FIL',STATUS='NEW',
&FORM='FORMATTED',BLANK='ZERO')
```

9.2.1.10 Carriage-Control Specifier

The format of the carriage-control specifier is as follows:

```
CARRIAGE=car
```

where

car	is a character expression evaluating to 'Fortran', 'NULL', or 'CONSOLE'. If you omit the CARRIAGE specifier, the default value is NULL.

If you specify Fortran, the compiler interprets the first character of formatted output as a vertical-spacing indicator for a printer. If you specify NULL, the first character is simply the first character of the new record. NULL and CONSOLE are equivalent.

When the Fortran convention is specified, the first character of each output record is interpreted as follows:

Character	Vertical Spacing
Blank	One line
0	Two lines
1	Skip to next page
+	No advance

For list-directed I/O, Fortran inserts the blank character.

If you write a file with CARRIAGE='Fortran', that file cannot be read in by a Fortran program.

```
OPEN(6,FILE='MYPROG.FIL',STATUS='OLD',
&FORM='FORMATTED',CARRIAGE='NULL')
```

9.2.1.11 Opening a Connected Unit

A unit is considered connected if it was referenced in a previous I/O statement without an intervening CLOSE statement. You can specify an OPEN statement for a unit already connected to a file.

If the file name specified by the OPEN statement is missing or is the same as that of the connected file, the BLANK and CARRIAGE specifiers (and the RECL specifier for sequential files) can differ from existing attributes, and result in changes to those attributes.

If the file name specified by the OPEN statement is not the same as that of the connected file, Fortran disconnects the previous file as if a CLOSE statement, without STATUS specifier, were issued and opens the new one with the new attributes.

If a file is already connected to a unit, you cannot specify an OPEN statement connecting that file to a different unit.

9.2.2 CLOSE Statement

The CLOSE statement disconnects a file from a unit. Its format is as follows:

CLOSE(*close-list*)

where

close-list	is the following list of specifiers separated by commas:	
	[UNIT=]*unit*	Unit specifier
	IOSTAT=*stname*	I/O status specifier
	ERR=*stl*	Error specifier
	STATUS=*stat*	File disposition specifier

The unit specifier must be present. All other specifiers are optional, and you can only specify them once.

The IOSTAT and ERR specifiers have the same interpretations as for the OPEN statement. (See Sections 9.2.1.2 and 9.2.1.3.)

9.2.2.1 Unit Specifier

The unit specifier has the same interpretation as in the OPEN statement. However, execution of the CLOSE statement containing this specifier need not occur in the same program unit as its corresponding OPEN statement. If the specified file does not exist, CLOSE has no effect.

Once a CLOSE statement disconnects a unit, it can be reconnected to the same file or a different file within the same program. Similarly, once a CLOSE statement disconnects a file, it can be reconnected to the same or a different unit, so long as the file still exists.

Example:

```
CLOSE(3,IOSTAT=ERRFLG,ERR=100)
```

9.2.2.2 File-Disposition Specifier

The format of the file-disposition specifier is as follows:

STATUS=*stat*

where

stat is a character expression evaluating to 'KEEP' or 'DELETE'. If you omit this specifier, the default value is DELETE for a file that previously had a status of SCRATCH, and KEEP otherwise. You cannot specify KEEP for a file opened with SCRATCH status.

If you specify KEEP for an existing file, the file continues to exist after Fortran executes the CLOSE statement. KEEP has no other effect.

If you specify DELETE, the file ceases to exist after Fortran executes the CLOSE statement.

Following normal program termination, Fortran closes all connected units and deletes all those designated as scratch files.

Example:

```
CLOSE(4,ERR=100,STATUS='KEEP')
```

9.2.3 BACKSPACE

The BACKSPACE statement causes the file pointer to move to the start of the preceding record. The file must be connected for sequential access. The possible formats are as follows:

BACKSPACE*unit*
BACKSPACE(*arg-list*)

where

unit	is an integer expression between 0 and 255 that identifies an external unit. The external-unit specifier must be present but the other specifiers are optional.
arg-list	is a list of arguments separated by commas. The following is a list of the arguments:

[UNIT=]*unit*	External-unit specifier
IOSTAT=*stname*	I/O status specifier
ERR=*stl*	Error specifier

If the file has no preceding record, the BACKSPACE statement has no effect. If the last I/O statement was a READ past the end-of file, the file is repositioned to the end of the file. You cannot backspace over a record written using list-directed formatting.

Backspacing a file that is connected but does not exist is prohibited. Do not use the BACKSPACE statement to manipulate iRMX 86 physical files such as :CI:, :CO:, line printers, or other such files. Fortran-86 returns run-time errors in these cases.

Examples:

```
BACKSPACE 3
BACKSPACE(3,ERR=100)
```

9.2.4 REWIND

The REWIND statement causes the file pointer to move to the initial point of the file. The file must be connected for sequential access. The possible formats are as follows:

REWIND *unit*
REWIND(*arg-list*)

where

unit	is an integer expression between 0 and 255 that identifies an external unit.
arg-list	is a list of arguments separated by commas. The *arg-list* for REWIND and the *arg-list* for BACKSPACE are the same.

If the file is positioned at its initial point, the REWIND statement has no effect.

Example:

```
REWIND 3
REWIND(3,IOSTAT=ERRFLG)
```

9.2.5 ENDFILE

The ENDFILE statement causes the preceding record to become the last record of the file. No further data-transfer I/O statements can be executed without first issuing a BACKSPACE or a REWIND statement. The file must be connected for sequential access.

The possible formats are as follows:

ENDFILE *unit*
ENDFILE(*arg-list*)

where

unit	is an integer between 0 and 255 that identifies an external unit.
arg-list	is a list of arguments. These arguments are the same as those for BACKSPACE and REWIND.

Do not use the ENDFILE statement to manipulate iRMX 86 physical files such as :CO:, :CI:, line printers, or other such files. Fortran-86 will return a run-time error in such cases.

Examples:

```
ENDFILE 4
ENDFILE(4,ERR=100)
```

9.3 Data-Transfer I/O Statements

Fortran provides three data-transfer I/O statements: READ, WRITE, and PRINT.

9.3.1 READ Statement

The READ statement reads data from a specified unit. Its formats are as follows:

READ(*ctl-list*) [*in-list*]
READ *f*[,*in-list*]

where

ctl-list is a list of control information specifiers. The control-information specifiers are as follows:

[UNIT=] *unit*	Unit specifier
[FMT=] *f*	Format specifier
REC=*recno*	Record number specifier
IOSTAT=*stname*	I/O status specifier
ERR=*stl*	Error specifier
END=*stl*	End-of-file specifier

in-list	is a list of the variables which are to receive the input data.
f	is a format identifier, which is the same as the FMT specifier in *ctl-list*.

9.3.1.1 Control-Information List

The control-information list must contain a unit specifier. If you use the second form of the READ statement, the unit is the default input unit.

The list can contain only one of each of the other specifiers.

The following sections describe the control list specifiers in detail.

Unit Specifier

The unit specifier has the form that follows:

```
[UNIT=]unit
```

where

unit	is an integer value between 0 and 255 that identifies an external unit, an asterisk (*) to specify the default input unit, or an internal file. For internal files, *ctl-list* must contain a format identifier but must not contain a record number specifier. If you omit UNIT= , *unit* must be the first item in *ctl-list*.

Example:

```
READ(2)BILL,STAT
```

Format Specifier

If *ctl-list* contains a format specifier, the READ statement is a formatted I/O statement. Otherwise, it is an unformatted I/O statement.

The format is as follows:

```
[FMT=]f
```

where

f	is one of the following: • The label of a FORMAT statement in the same program unit as the READ statement • An integer variable assigned the label of a FORMAT statement in an ASSIGN statement • A character array name, character variable name, or character expression containing a format specification • An INTEGER, floating-point, or LOGICAL array containing a format specification as Hollerith data • An asterisk (*) specifying list-directed formatting (Section 9.4.2) If you omit FMT=, the format specifier must be the second item in *ctl-list* and you must omit UNIT= as well. If you specify an asterisk (*) as *f*, *ctl-list* cannot contain a record number specifier. If the unit is an internal file, the format specifier must also be present, but cannot be an asterisk (*).

Examples:

```
        READ(2,25)BILL,STAT
25      FORMAT....

        READ 30,BILL,STAT
30      FORMAT....

        ASSIGN 45 TO HORN
        READ(2,HORN)BILL
45      FORMAT....

        READ(2,*)BILL
```

Record-Number Specifier

If you connected the file for direct access, you must include the record-number specifier in *ctl-list*. Its format is as follows:

REC=*recno*

where

recno	is a positive integer expression whose value is the number of the record to be read.

Examples:

```
READ(3,REC=15)
READ(2,REC=J)
```

Input/Output Status Specifier

The I/O status specifier is essentially the same as for the OPEN statement (Section 9.2.1.2). In addition, Fortran assigns the variable *stname* a negative value at end-of-file.

Error Specifier

The error specifier has a similar interpretation as for the OPEN statement (Section 9.2.1.3), with one difference: if the error is the result of an end-of-file condition, the position of the file is defined as past the end-of-file marker; further I/O operations except CLOSE, REWIND, or BACKSPACE are undefined.

End-Of-File Specifier

The format of the end-of-file specifier is as follows:

END=*stl*

where

stl	is the label of an executable statement in the same program unit as the READ statement.

When Fortran detects an end-of-file during a READ operation, processing procedes as for the error specifier except that execution continues with the statement specified by END.

If you specify END, the file must be connected for sequential access.

Example:

```
READ(3,30,IOSTAT=STFLG,ERR=100,END=300)BILL,STAT
```

9.3.1.2 Input List

The input list, *in-list*, identifies the items to be read. An item in *in-list* must be a variable name, array name, or array element name. If you list an array name, Fortran reads the entire array in normal array element ordering sequence. You cannot list the name of an assumed-size dummy array in the input list.

9.3.1.3 Implied-DO List

An implied-DO list embedded in the READ statement allows you to use a range of subscripts for input list array elements. For example, Fortran can read some of the items in an array without your specifying each individual array element. The format of the implied-DO list is as follows:

(*inlist*, *var*=*e1*, *e2*, *e3*)

where

var,*e1*,*e2* and *e3*	have the same interpretation as for the DO statement (Section 8.3).
inlist	is the list of input items described above. The list, *in-list*, may contain additional implied-DO lists.

For READ statements, the DO variable *var* cannot appear as an item in *in-list*.

Example:

```
C READ THE ODD ELEMENTS IN THE ARRAY 'TABLE'
      DIMENSION TABLE(60)
      READ (2,20)(TABLE(N),N=1,59,2)
20 FORMAT....
```

9.3.2 WRITE

The WRITE statement outputs data to a specified unit. The format is as follows:

WRITE(*ctl-list*) [*out-list*]

where

ctl-list is a list of control-information specifiers. The control-list specifiers are analogous to those for READ (Section 9.3.1.1). The control-information list is as follows:

[UNIT]*unit*	Unit specifier
[FMT=]*f*	Format specifier
REC=*recno*	Record-number specifier
IOSTAT=*stname*	I/O status specifier
ERR=*stl*	Error specifier

out-list is a list of the data to be written.

The syntax of the output list, *out-list*, is similar to that of the *in-list* in the READ statement, including the implied-DO option (Sections 9.3.1.2 and 9.3.1.3). In addition, an output list item may be an expression of any data type.

Examples:

```
       WRITE(6,120)BILL,STAT
120    FORMAT....

       WRITE(6,120,IOSTAT=ERRFLG,ERR=200)
      &BILL+1,STAT+1
120   FORMAT....

       DIMENSION BILL(25),STAT(25)
C WRITE A DOUBLE COLUMN PRINTOUT OF THE
C FIRST ITEMS OF EACH ARRAY
       WRITE(6,120)(BILL(H),STAT(H),H=1,10)
120    FORMAT(1X,A,5X,F4.3)
```

9.3.3 PRINT

The PRINT statement outputs formatted data to the default output unit. Its format is as follows:

PRINT *f*[, *out-list*]

where

f	is a format identifier.
out-list	is a list of the data to be written.

The format specifier *f* and *out-list* have the same meaning as in the WRITE statement.

Examples:

```
       PRINT 50,BILL,HORN
50     FORMAT....

       ASSIGN 50 TO STAT
       PRINT STAT,BILL,HORN
50     FORMAT....
```

9.4 Formatted Data Transfer

The default for the FORM specifier in the OPEN statement is *FORMATTED* for sequential-access files. During formatted data transfer, Fortran transfers data with editing between the file and the I/O list. The editing is directed by some kind of formatting specification. You can specify formats as follows:

- In FORMAT statements
- As values of character arrays, character variables, or other character expressions
- As Hollerith values assigned to integer, floating-point, or logical arrays
- As list-directed I/O (see Section 9.4.2)

If the format specifier in a formatted I/O statement is an array or expression, its value must be a valid format specification in its leftmost character or Hollerith positions. Any data following the right parenthesis that ends the format specification has no affect on the format specification itself.

If a formatted record is written using the CARRIAGE = 'Fortran' option, the first character of the record is not printed. This character indicates vertical spacing for external printers (see Section 9.2.1.10). The remaining characters are printed beginning at the left margin.

9.4.1 FORMAT Statement

The form of the FORMAT statement is as follows:

stl FORMAT([*flist*])

where

stl	is a 1 to 5 digit statement label.
flist	is a format specification list whose items are separated by commas. Each item in *flist* must be an edit descriptor or another (imbedded) parenthesized *flist*.

You can specify a FORMAT statement with no *flist* only if the I/O list is also empty.

There are two kinds of edit descriptors, repeatable and nonrepeatable. You repeat an edit descriptor by prefixing it with a nonzero, unsigned integer constant called a repeat specification. A repeat specification may also be present for an imbedded *flist*.

Both the format specification and its corresponding I/O list are scanned from left to right. Each item in the I/O-list corresponds to the next repeatable edit descriptor. For example, if a repeatable edit descriptor is repeated five times, it corresponds to five consecutive I/O list items. There is no corresponding I/O-list item for nonrepeatable edit descriptors which take effect whenever they are encountered.

If an embedded *flist* is preceded by a repeat specification, *flist* is scanned that many times before continuing to the next format item.

If a format-specification list ends before the I/O list ends, it reverts to the beginning of the last imbedded *flist* in the FORMAT statement including its repeat specification. If none is present, then it reverts to the beginning of the FORMAT statement. Repeat specifications have the same effect as during the first pass through the format specification list. A new record is begun each time format reversion occurs.

9.4.1.1 Repeatable Edit Descriptors

Each repeatable edit descriptor generally consists of a letter indicating the type of data involved and a number indicating the size of the data field; additional information may specify how it will be divided. The repeatable edit descriptors are as follows:

I*w*	Integer descriptor
F*w.d*	Floating-point descriptor
E*w.d*[E*e*]	Floating-point descriptor
D*w.d*	Floating-point descriptor
G*w.d*[E*e*]	Floating-point descriptor
L*w*	Logical descriptor
A[*w*]	Alphanumeric descriptor
B*w*	Binary descriptor
Z*w*	Hexadecimal descriptor

where

I,F,E,D	indicate the external type of data being edited.
B and Z	indicate the external number base of data being edited.
w	is a nonzero, unsigned integer constant representing the width of the entire external field.
d	is an unsigned integer constant representing the number of digits that follow the decimal point.
e	is a nonzero, unsigned integer constant representing the number of digits of the exponent.
G,L, and A	The I, F, D, E, and G edit descriptors are used for numeric data. E and G editing allows output of floating-point numbers in scientific notation.

The following remarks apply to the I, F, D, E, and G edit descriptors.

- On input, leading blanks are not significant. Further blanks are treated according to the setting of the nonrepeatable descriptors BN and BZ and the value of the BLANK specifier in the OPEN statement.
- A decimal point in input data overrides the decimal-point location specified by a descriptor. The input field may have more digits than are necessary for the value of the data item to be approximated.
- On output, Fortran right-justifies values. If necessary, the compiler fills the field with blanks on the left.
- On output, if the number of characters exceeds the field width *w*, or an exponent has more than *e* digits, the entire field is filled with asterisks (**).

The B and Z edit descriptors specify data I/O in binary and hexadecimal notation, respectively.

INTEGER Editing

An I/O-list item matched with an I*w* edit descriptor must be of type INTEGER. The integer constant read or written always consists of at least one digit.

Examples:

```
      PRINT 20,INTNUM
20    FORMAT(I5)

      READ(3,20)INTNM1,INTNM2,INTNM3
20    FORMAT(2I5,I4)
```

F Descriptor Editing

An I/O-list item matched with an F*w.d* descriptor must have a floating-point data type. If the input to this descriptor contains no decimal point, Fortran interprets the rightmost *d* digits of the string as the fractional part of the input value.

On input, an exponent consisting of a signed integer constant or the letter E followed by an optionally signed integer constant can follow the string of digits.

Fortran rounds output edited by the F descriptor to *d* fractional digits and can modify it by an established scale factor. (See the description of the nonrepeatable edit descriptor P.)

Examples

```
      READ(2,20)BILLN
20    FORMAT(F5.3)

      DIMENSION TABLE(10)
      PRINT 30,TABLE
30    FORMAT(5(F5.3,2X,F5.3))
C THE TABLE WILL PRINT IN TWO COLUMNS
```

E and D Descriptor Editing

An I/O-list item matched with an E*w.d*, D*w.d*, or E*w.d*E*e* descriptor must have a floating-point data type. The exponent *e* has no effect on input data.

On output, the format of the output field for a scale factor of zero is as follows:

[*sign*] [0].x1x2...*xd exp*

where

sign	is either a plus (+) or a minus (−) sign.
x1...xd	are the *d* most significant digits of the value after rounding.
exp	is a decimal exponent having one of the forms found in Table 9-1.

The scale factor, *k* (see the description of the nonrepeatable edit descriptor P), controls decimal normalization. If $-d < k < 0$, the number written will have exactly $|k|$ leading zeros and $d-|k|$ significant digits following the decimal point. If $0 < k < d + 2$, the number will have exactly *k* significant digits to the left of the decimal point and $d - k + 1$ significant digits to the right of the decimal point. Other values of *k* are illegal.

Examples

```
      READ(2,20)RLNUMB
20    FORMAT(E4.2)

      WRITE(6,110)ROUT
110   FORMAT(E15.5E6)
```

Table 9-1. Output Forms of Exponents For D and E Editing

Edit Descriptor	Magnitude of Exponent (exp)	Form of exponent (y=digit)
E*w.d*	$\lvert exp \rvert \leq 99$ $99 < \lvert exp \rvert \leq 999$	$E\pm y_1\, y_2$ $\pm y_1\, y_2\, y_3$
E*w.d* E*e*	$\lvert exp \rvert \leq (10^{**}e)-1$	$E\pm y_1\, y_2 \ldots y_e$
D*w.d*	$\lvert exp \rvert \leq 99$ $99 < \lvert exp \rvert \leq 999$	$D\pm\, y_1\, y_2$ $\pm\, y_1\, y_2\, y_3$

G Descriptor Editing

An I/O list item matched with a G*w.d* or G*w.d* E*e* must have a floating-point data type.

On input, G descriptor editing is the same as F descriptor editing.

On output, editing depends on the magnitude of the value to be written. Let *n* be the magnitude of the value. If $n < 0.1$ or $n \geq 10^{**}d$, G editing is the same as E editing with the current scale factor. If $0.1 \leq n < 10^{**}d$, the scale factor has no effect. Table 9-2 describes the editing in this case.

LOGICAL Editing

An I/O-list item matched with an L*w* descriptor must have a logical data type.

The input field includes optional blanks preceding an optional period followed by a T (for TRUE) or F (for FALSE). These letters may be followed by additional characters. For example, the logical constants .TRUE. and .FALSE. are acceptable inputs.

The output field consists of the letters T and F based on the TRUE or FALSE value of the internal data preceded by blanks, if necessary, to fill the output field.

Examples

```
        LOGICAL TRUTH
        DIMENSION TRUTH(4)
        READ(3,50)TRUTH(1),TRUTH(4)
50      FORMAT(2L6)

        WRITE(6,80)TRUTH(1)
80      FORMAT(L1)
```

Alphanumeric Editing

An I/O-list item matched with an A or A*w* descriptor must have type CHARACTER or be defined with Hollerith data. If you specify the field width, *w*, the field consists of *w* characters. Otherwise, the number of characters in the field is the *length* of the I/O-list item.

Table 9-2. G Editing for $0.1 \leq N < 10^{**}d$

Magnitude of Data	Equivalent Conversion
$0.1 \leq N < 1$	F(w−n).d, n(b)
$1 \leq N < 10$	F(w−n).(d−1), n(b)
•	•
•	•
•	•
$10^{**}(d-2) \leq N < 10^{**}(d-1)$	F(w−n).1.n(b)
$10^{**}(d-1) \leq N < 10^{**}d$	F(w−n).0, n(b)
	where n = 4 for G*w.d* *e* + 2 for G*w.d* E*e* b = blank

With A*w* editing, if *w* > *length*, the following are equivalent:

A*w* and (*w* - *length*)X,A*length*

If w ≤ *length*, then the data is transferred according to the rules for character assignment.

The following illustrates A*w* editing. In these examples, *b* indicates a blank.

A5 to CHARACTER*3:	ABCDE becomes CDE
	w>length
CHARACTER*3 to A5:	ABC becomes *bb*ABC
A3 to CHARACTER*5:	ABC becomes ABC*bb*
	w<length
CHARACTER*5 to A3:	ABCDE becomes ABC

Number-Base Editing

The B and Z edit descriptors are Fortran-86 extensions which allow formatted I/O of values expressed in terms of their internal data representation. B*w* specifies that the value represents a bit string consisting of *w* characters, each one 0, 1, or blank. Z*w* indicates a string of *w* characters consisting of the hexadecimal digits 0 through F, and blank. On input, blanks are interpreted according to the current BN/BZ edit descriptor, or the BLANK specifier of the OPEN statement. On output, blanks are inserted on the left if the field width *w* is greater than the total number of binary or hexadecimal digits in the output-list item.

Further interpretation of B- and Z-formatted data depends on the corresponding I/O-list item, as shown in Table 9-3.

Table 9-3. Interpretation of B and Z Values*

I/O-List Data Type	B or Z Value	Truncation or Zero Padding**
INTEGER LOGICAL	unsigned, integer value with the least significant digit on the right	left
REAL*4	leftmost bit: sign next 8 bits: exponent next 23 bits: significand	right
REAL*8 DOUBLE PRECISION	leftmost bit: sign next 11 bits: exponent next 52 bits: significand	right
TEMPREAL	leftmost bit: sign next 15 bits: exponent next 1 bit: normal bit next 63 bits: significand	right
CHARACTER	internal binary representation of ASCII from left to right	right

* COMPLEX data is interpreted as two REAL values.
**Zero Padding occurs only on input.

9.4.1.2 Nonrepeatable Edit Descriptors

The nonrepeatable edit descriptors are

'c1c2...cn'	Literal-string descriptor
*n*H*c1c2...cn*	Hollerith-string descriptor
*n*X	Record-position control descriptor
/	Record-termination descriptor
*k*P	Scale-factor descriptor
BN	Blank descriptor
BZ	Blank descriptor
$	Alternate record-termination descriptor

where

apostrophe ('), H,	indicate the kind of editing.
X, slash (/), P, BN, BZ, and the dollar sign ($)	indicate the kind of editing.
c	is any ASCII character.
n	is a nonzero, unsigned integer constant.
k	is an optionally signed integer constant representing a scale factor.

Apostrophe Editing

You use the apostrophe edit descriptor only for output. It causes Fortran to write the characters enclosed in apostrophes literally. To indicate an apostrophe within the character field, show it as two consecutive apostrophes.

The width of the field is the length of the character string.

Example

```
      WRITE(7,100)ITSTNO
100   FORMAT('THIS IS TEST NUMBER',2X,I2)
```

H Descriptor Editing

The Hollerith edit descriptor is an alternate way to perform literal-string editing. Like apostrophe editing, you can use it only for output. The *n*H descriptor causes the compiler to write the *n* characters following the H.

Example

```
      WRITE(7,100)ITSTNO
100   FORMAT(1H1,19HTHIS IS TEST NUMBER,2X,I2)
C FIRST H DESCRIPTOR CAUSES A SKIP TO A NEW PAGE
```

X Descriptor Editing

The *n*X descriptor indicates that the next edit descriptor applies to the character *n* positions from the current record position. On output, Fortran inserts *n* blanks into the output record. No blanks are output if there are no more items in the I/O list.

Example

```
      WRITE(7,100)ITSTNO
100   FORMAT(1X,'THIS IS TEST NUMBER',2X,I2)
C FIRST X DESCRIPTOR CAUSES SINGLE SPACING
C BY INSERTING A BLANK AS THE FIRST
C CHARACTER OF THE RECORD
```

Slash Editing

The slash (/) edit descriptor acts as an end-of-record indicator.

On input, Fortran skips the remainder of the current record. If the file is positioned at the beginning of a record, Fortran skips the entire record.

On output, Fortran terminates the current record and begins a new record. You can use the slash edit descriptor to write an empty record, a convenient way to provide blank lines on printed output.

The comma that normally separates FORMAT list items is not required before or after a slash.

Example

```
      WRITE(7,100)
100   FORMAT(1H1,'    BILL AVERAGE'/)
C THIS SLASH CAUSES A BLANK LINE FOLLOWING
C THE HEADINGS TO BE WRITTEN
      WRITE(7,150)BILL,AVG
150   FORMAT(1X,A12,4X,F4.3)
```

Scale Factor (P) Editing

The *k*P descriptor establishes a scale factor, *k*, which applies to certain subsequent floating-point descriptors until a new scale factor is specified. You can use it with the F, D, E, and G descriptors when editing floating-point numbers. If an F, D, E, or G immediately follows the P, no intervening comma is necessary.

Fortran assumes a scale factor of zero at the beginning of an I/O statement. Once the *k*P descriptor changes it, the new scale factor remains in effect until you assign another scale factor or until the end of the I/O statement.

On input, the scale factor has no effect if there is an exponent in the F, D, E, or G input field. Otherwise, the effect is that the externally represented number equals the internally represented number multiplied by 10***k*. The same is true on output with F editing.

On output with E or D editing, Fortran moves the decimal point *k* positions to the right (left if negative) and reduces the exponent by *k*.

On output with G editing, Fortran suspends the effect of the scale factor as long as the value is within the range of F editing. If not, the effect is the same as described for E editing.

The output range of a significand printed in scientific notation is 0.1 to, but not including, 1.0, with a scale factor of zero. Setting the scale factor to 1P changes this range to 1.0 to 10.0. Changing the scale factor is useful for very small or very large

E-edited numbers, but generally not for F-edited numbers. You should reset the scale factor as necessary for subsequent floating-point items.

Table 9-4 illustrates the use of the scale factor with E editing on output.

BN and BZ Editing

You can use these two edit descriptors to specify the interpretation of blanks, other than leading blanks, on input. If you specify BN, Fortran ignores all blanks, except that it treats a field of all blanks as zero. If you specify BZ, Fortran treats all blanks as zeros.

Unless you specify the BN or BZ descriptor, the BLANK specifier in the OPEN statement determines the interpretation of blanks. Once BN or BZ has been specified, the new specification remains in effect until changed again explicitly, or until the end of the I/O statement.

Example

```
      READ(2,50)INTNUM,FPNUM
50    FORMAT(BN,I5,5X,BZ,F7.4)
```

If the input values for this example are 1*b*0 and 1*b*0.0, where *b* is a blank, then the variables will contain 10 and 100.0, respectively.

Dollar-Sign Editing

You use the dollar-sign ($) edit descriptor for interactive I/O through a console terminal. It leaves the terminal cursor at the position immediately following the output data just processed, rather than at the beginning of a new line. If the FORMAT scanner encounters a dollar sign after processing the last output I/O-list item, format control terminates without positioning the file to the beginning of the next record. Any other imbedded dollar signs are ignored. The dollar-sign descriptor is ignored if the file is not defined for sequential output with CARRIAGE = 'CONSOLE' in the OPEN statement.

Example

```
      PRINT25,BILL
25    FORMAT(A20,$)
```

9.4.2 List-Directed Formatting

List-directed formatting allows free-form formatted input and output. To specify list-directed formatting, place an asterisk (*) in the format-specifier position of the data-transfer statement's control list. No FORMAT statement is necessary.

Table 9-4. Floating-Point Editing for Output with the Scale-Factor Edit Descriptor P

Real Number	F6.2	E11.5	1PE10.4
4.32	4.32	0.43200 E + 01	4.3200 E + 00
7255000.0	******	0.72550 E + 07	7.2550 E + 06
0.0065	0.01	0.65000 E − 02	6.5000 E − 03

A list-directed file is an external file whose records contain values and value separators. Each value can be any of the following:

- A constant
- A null value
- A constant or null value prefixed by a repeat specifier in the form that follows:

  *r*c*

  or

  *r**

  where

r	is an unsigned, nonzero integer constant.
c	is a value.

The form *r*c* is equivalent to *r* occurences of the value *c*. The form *r** is equivalent to *r* successive null values. Neither form can contain embedded blanks, except within the value *c*.

A value separator can be any of the following:

- A comma, optionally preceded or followed by blanks
- A slash, optionally preceded by blanks
- One or more blanks between two values or following the last value

9.4.2.1 List-Directed Input

Execution of a list-directed READ statement begins a new record and formats each input value according to the type of the corresponding input-list item and the width, *w*, of the value as follows:

Type of Input Item	Equivalent Format Descriptors
CHARACTER**n*	A*w* $w \leq n$ A*n*, (*w-n*)X otherwise
LOGICAL**n*	L*w*
INTEGER**n*	I*w*
REAL**n*	F*w*.0
DOUBLE PRECISION	F*w*.0
TEMPREAL	F*w*.0
COMPLEX	'('F*w*.0','F*w*.0')'
COMPLEX*16	'('F*w*.0','F*w*.0')'

All values acceptable to these FORMAT specifications are acceptable for list-directed formatting with a few exceptions:

- Since blanks are treated as separators, imbedded blanks are allowed only within character strings.
- An end-of-record specifier has the same effect as a blank except within a character string, which is continued on the next record.

- An input LOGICAL value must contain neither commas nor slashes among the optional characters following the T or F.
- An input character value consists of a string of characters enclosed by an apostrophe at each end.

You represent an apostrophe within the character constant by two consecutive apostrophes without intervening blanks or end-of-record. You can continue a character constant from the end of one record to the beginning of the next record. Although in list-directed formatting an end-of-record normally has the effect of a blank, that does not apply in this case. The characters blank, comma, and slash can appear within character constants. Fortran transfers a character string leftjustified, and blank fills or truncates them on the right if its width is not the same as the width of the input-list item.

You can specify null values in one of two ways.

- By having no values between successive separators or preceding the first value separator
- By specifying the *r** form

An end-of-record following a value, a comma, or another end-of-record, with or without separating blanks, does not imply a null value.

A null value has no effect on the corresponding input-list item. The item retains its previous value or remains undefined, depending on its status before the null value is encountered.

If you use a slash as a value separator during execution of a list-directed input statement, execution of that input statement is terminated at that point. If there are additional items in the input list, they are treated as null values.

9.4.2.2 List-Directed Output

Execution of a list-directed WRITE (or PRINT) statement begins a new record and formats the value of each output-list item by type as follows:

Type of Output Item	Equivalent Format Descriptors
CHARACTER**n*	A
LOGICAL**n*	L2
INTEGER**n*	*I13*
REAL**n*	1P, E25.15E4
DOUBLE PRECISION	1P, E25.15E4
TEMPREAL	1P, E25.15E4
COMPLEX	1P, (E25.15E4, E25.15E4)
COMPLEX*16	1P, (E25.15E4, E25.15E4)

See Section 9.4.1 for a description of these format descriptors.

Fortran-86 separates the output values into records of not more than 80 characters, with one exception: a character string of more than 80 characters results in a separate record whose size is the same as the length of the character string.

The following is an example of list-directed output:

```
CHARACTER   CH100*100,CH5*5
LOGICAL     LOG1


PRINT *,CH5,CH100,L1,I,X
```

is equivalent to the following:

```
      PRINT 100 CH5,CH100,LOG1,I,X
100   FORMAT(A,/,A,/,L2,I13,1P,E25.15E4)
```

9.5 Unformatted Data Transfer

Only external units are allowed in data-transfer statements involving unformatted data. The default for the form specifier in the OPEN statement is UNFORMATTED for direct-access files. Fortran transfers data without editing between the current record of the connected file and items on the I/O list. Exactly one record is read or written.

The number of items in an input list must not exceed the number of values in the record. The type of each value in the record must agree with the type of the corresponding input list item. The item and its value also must agree in length.

On output, if the file is connected for direct access and the values in the output list do not fill the record, the remainder of the record is undefined.

CHAPTER 10 EXAMPLES

This chapter gives example programs that illustrate Fortran-86 features. Each program resides on a Fortran-86 software package product disk.

10.1 I/O Examples

10.1.1 Program 1A (PROG1A.FTN)

The following example illustrates the use of direct access, unformatted I/O. The program first writes the digits 1 through 10 into the file LIST on drive :F1:. After reading two distinct sections of that file, the program prints the digits 5 through 10 and 3 through 7 to the console.

To execute this program, you must link it with the run-time libraries listed in the system specific appendix. Figure 10-1 lists PROG1A.FTN.

```
      PROGRAM PROG1A
      OPEN (1,FILE=':F1:LIST',ACCESS='DIRECT',RECL=2)

      DO 120 I=1,10
      WRITE (1,REC=I) I
120   CONTINUE

      DO 140 K=5,10
      READ (1,REC=K) I
C
C  SEQUENTIAL, FORMATTED I/O TO THE CONSOLE.
C
      WRITE (6,130) I
130   FORMAT (I2)
140   CONTINUE

      DO 160 J=3,7
      READ (1,REC=J) I
C
C  SEQUENTIAL, FORMATTED I/O TO THE CONSOLE.
C
      WRITE (6,150) I
150   FORMAT (I2)
160   CONTINUE

      END
```

Figure 10-1. PROG1A.FTN—Direct Access, Unformatted I/O

10.1.2 Program 1B (PROG1B.FTN)

The following example illustrates the use of sequential access, formatted I/O with the console. The program asks for two inputs: your name and your social security number, prompting you for the correct format.

To execute this program, you must link it with the run-time libraries listed in the system specific appendix. Figure 10-2 lists PROG1B.FTN.

```
      PROGRAM PROG1B

      CHARACTER*20 NAME
      INTEGER*4 SSNUM
C
C
5     WRITE(6,10)
10    FORMAT('What is your name?'/,5x,'enter using A20 format ',$)
      READ(5,20,ERR=70)NAME
20    FORMAT(A20)

      WRITE(6,50)
50    FORMAT('What is your social security number?'/,5x,'enter as nnnnnnnnn ',
     &$)
      READ(5,60,ERR=70)SSNUM
60    FORMAT(I9)

      GOTO 90

70    WRITE(6,80)
80    FORMAT('Incorrect input...please enter again'//)

      GOTO 5
90    CONTINUE

      WRITE(6,100)NAME,SSNUM
100   FORMAT('Name is: ',A20,/,'Social Security Number is: ',I9)

      END
```

Figure 10-2. PROG1B.FTN—Sequential Access, Formatted I/O

10.1.3 Program 1C (PROG1C.FTN)

The following example illustrates the use of list-directed I/O with the console. The program initially asks for two inputs: the first of one character, the second of six. Each input must be a quoted string. The program then prompts for you to re-enter your original input using an appropriate delimiter (a comma, a space, or a return). An input of X will terminate the program.

To execute this program, you must link it with the run-time libraries listed in the system specific appendix. Figure 10-3 lists PROG1C.FTN.

```
        PROGRAM PROG1C
        CHARACTER*1 ANS1
        CHARACTER*6 ANS2

10      WRITE(6,*) 'INPUT 1 CHARACTER - AN INPUT OF ''X'' WILL TERMINATE',
     &  ' THE PROGRAM'
        READ(5,*) ANS1

        IF (ANS1.EQ.'X') GO TO 20

        WRITE(6,*) 'THE CHARACTER YOU CHOSE IS: ', ANS1

        WRITE(6,*) 'INPUT 6 CHARACTERS'
        READ(5,*) ANS2
        WRITE(6,*) 'THE NEW CHARACTERS ARE: ', ANS2

        WRITE(6,*) 'NOW INPUT BOTH CHARACTERS.  REMEMBER TO USE A DELIMITER',
     &  ' BETWEEN EACH CHARACTER ( IE., COMMA, SPACE, or RETURN)'
        READ(5,*) ANS1,ANS2
        WRITE(6,*) 'YOUR TWO INPUTS ARE ', ANS1,', ',ANS2

        GO TO 10

20      STOP
        END
```

Figure 10-3. PROG1C.FTN—List Directed I/O

10.2 TEMPREAL Example

10.2.1 Program 2 (PROG2.FTN)

The following example illustrates the use of the TEMPREAL data type. This data type is recommended for use as an intermediate result of double precision arithmetic. The program asks for two real inputs prompting you for the correct format. These inputs are used to fill an array with double precision values. Two summations are calculated from this input: one double precision and one TEMPREAL. The intermediate results are compared and their difference is printed to the console.

To execute this program, you must link it with the run-time libraries listed in the system specific appendix. Figure 10-4 lists PROG2.FTN.

```
      PROGRAM PROG2
      DOUBLE PRECISION RARRAY,RTOTAL,RESULT,DPRES
      TEMPREAL TMPRES
      COMMON RTOTAL, RARRAY(500)

      CALL GETDAT
      DPRES = 0.0
      TMPRES = 0.0

      DO 10, I = 1,500
          DPRES  =  DPRES + RARRAY(I)/RTOTAL
          TMPRES = TMPRES + RARRAY(I)/RTOTAL
10    CONTINUE

      RESULT = TMPRES

      PRINT 100, RESULT, DPRES
100   FORMAT ('RESULT = ', E26.20E2, ', D-P RESULT = ', E26.20E2)

      RESULT = DPRES - RESULT

      PRINT 200, RESULT
200   FORMAT ('DIFFERENCE = ', E13.5E4)

      END

      SUBROUTINE GETDAT

      DOUBLE PRECISION RARRAY,RTOTAL,RVALUE,FACTOR
      TEMPREAL TMPTOT
      COMMON RTOTAL, RARRAY(500)

      TMPTOT = 0.0

      PRINT 100
100   FORMAT('ENTER STARTING VALUE BETWEEN 0.00 AND 4.00 IN F4.2 FORMAT')
      READ  200, RVALUE
200   FORMAT(F4.2)
      PRINT 300
300   FORMAT('ENTER MULTIPLICATIVE FACTOR BETWEEN 0.00 AND 4.00 IN F4.2 FORM'T'
      READ 200, FACTOR
```

Figure 10-4. PROG2.FTN—TEMPREAL

```
      DO 10, I = 1, 500
          RARRAY(I) = RVALUE
          TMPTOT = TMPTOT + RVALUE
          RVALUE = RVALUE * FACTOR
10    CONTINUE

      RTOTAL = TMPTOT

      END
```

Figure 10-4. PROG2.FTN--TEMPREAL (Cont'd.)

10.3 $INTERRUPT Example

10.3.1 Program 3 (PROG3.FTN)

The following example illustrates the use of the $INTERRUPT control and the SETINT intrinsic. This program initializes an 8253 interval timer on an iSBC-86/12A board to interrupt the host processor every ten milliseconds.

You must link this program with the run-time libraries listed in the system specific appendix. Figure 10-5 lists PROG3.FTN.

```
        PROGRAM PROG3

        INTEGER*1 CONTPT,CONTWD,CNTLOW,CNTHI,CNTREG
        EXTERNAL TIMER

        CALL SETINT (6,TIMER)

        CONTPT = #0D6H
        CONTWD = #030H
        CALL OUTPUT (CONTPT,CONTWD)

        CNTREG = #0D0H
        CNTLOW = #0CH
        CNTHI  = #030H
C
C       LOAD THE LOW ORDER COUNTER BYTE.
C
        CALL OUTPUT (CNTREG,CNTLOW)
C
C       LOAD THE HIGH ORDER COUNTER BYTE.
C
        CALL OUTPUT (CNTREG,CNTHI)
C
C       ALWAYS TRUE TEST TO CONTINUE INTERRUPTS FOREVER.
C
 5      IF (1.NE.1) GO TO 10
        GO TO 5

 10     END


$INTERRUPT

        SUBROUTINE TIMER
        INTEGER*1 CNTREG,CNTLOW,CNTHI

        CNTREG = #0D0H
        CNTLOW = #0CH
        CNTHI  = #030H

        CALL OUTPUT (CNTREG,CNTLOW)
        CALL OUTPUT (CNTREG,CNTHI)

        RETURN
        END
```

Figure 10-5. PROG3.FTN—$INTERRUPT Control

10.4 $REENTRANT Example

10.4.1 Program 4 (PROG4.FTN)

The following example illustrates the use of the $REENTRANT control to write a recursive procedure. This program solves the Towers of Hanoi problem. A description of the problem is as follows:

> There are three pegs labelled A, B, and C. Peg A holds a stack of discs (number provided by operator). Pegs B and C have none. Each disc is of a different size. The discs are ordered on Peg A by size, starting with the largest on the bottom. The discs can be moved one at a time to any other peg as long as no disc is placed on top of another disc that is smaller in size. The object is to transfer the discs from Peg A to Peg C.

To execute this program, you must link it with the run-time libraries listed in the system specific appendix. Figure 10-6 lists PROG4.FTN.

```
          PROGRAM PROG4

          WRITE(6,100)
100       FORMAT('How many disks are to be moved from peg A to peg B: ',$)
          READ(5,200)NUM
200       FORMAT(I5)

          CALL HANOI('A','B','C',NUM)

          END



$REENTRANT

          SUBROUTINE HANOI(FROM,TO,BUFF,NUM)
          CHARACTER*1 FROM,TO,BUFF

          IF(NUM .EQ. 0) RETURN

          CALL HANOI(FROM,BUFF,TO,NUM-1)

          WRITE(6,100)FROM,TO
100       FORMAT('Move a disk from peg ',A,' to peg ',A)

          CALL HANOI(BUFF,TO,FROM,NUM-1)

          END
```

Figure 10-6. PROG4.FTN—$REENTRANT Control

10.5 Function Subprogram Example

10.5.1 Program 5 (PROG5.FTN)

The following example illustrates the use of a function subprogram by calculating the area of a rectangle. The program asks you for two inputs: the height and the width. Using these measurements, the program calculates the area and outputs the result to the console.

To execute this program, link it with the run-time libraries listed in the system specific appendix. Figure 10-7 lists PROG5.FTN.

```
      PROGRAM GEO
      REAL*4 HEIGHT,WIDTH,ANSW,AREA
      CHARACTER*1 MORE
      EXTERNAL AREA
C
C  INPUT THE DATA
C

5     WRITE(6,10)
      READ(5,20)HEIGHT
      WRITE(6,30)
      READ(5,20)WIDTH
C
C  INVOKE THE AREA FUNCTION
C
      ANSW=AREA(HEIGHT,WIDTH)
C
C  OUTPUT THE AREA AND CONTINUE
C
      WRITE(6,40)ANSW
      WRITE(6,50)
      READ(5,60) MORE

      IF(MORE.EQ.'Y'.OR.MORE.EQ.'y') GOTO 5
C
C  FORMAT STATEMENTS
C

10    FORMAT(//'Enter the height of the rectangle ',$)
20    FORMAT(F10.5)
30    FORMAT('Enter the width of the rectangle ',$)
40    FORMAT('The area of the rectangle is ',F10.5)
50    FORMAT('Continue with another input? (Y or N) ',$)
60    FORMAT(A1)

      END


      REAL FUNCTION AREA(X,Y)
      REAL*4 X,Y

      AREA=X*Y

      RETURN
      END
```

Figure 10-7. PROG5.FTN—Function Subprogram

CHAPTER 11
COMPILER CONTROLS

Compiler controls manipulate Fortran-86 compiler features, such as whether a listing will be produced or whether an object file will be generated during compilation. All controls have default values preset to their most common usage, so few controls need to be specified for a typical compilation.

By default, the Fortran-86 compiler produces two files: *source*.OBJ for the object module with type records, and *source*.LST for the source listing including error messages, where *source* is the filename (without extension) of the Fortran-86 program text file.

11.1 Invoking the Compiler

The system specific appendix provides instructions and examples of compiler invocation.

11.2 Kinds of Compiler Controls

Compiler controls fall into two main categories:

- *Primary controls* precede the first line of a program or module, or are part of the command line that calls the Fortran-86 compiler. Some primary controls can be specified only once. Certain controls are considered initial primary controls. They are PRINT/NOPRINT and OBJECT/NOOBJECT. They can be specified only at the beginning of compilation (command line or before the first module), but cannot be changed between modules. All other primary controls can appear between modules.
- *General controls* are interspersed anywhere throughout your program source code. Additionally, you can specify most general controls in the Series-III RUN command line that calls the Fortran-86 compiler. Any controls embedded in the source program must appear on a line beginning with a dollar sign ($). Use blanks to separate more than one control placed on a single line.

Table 11-1 lists the primary and general controls.

You can specify negation of most controls with the prefix NO. Table 11-2 shows the compiler controls and their standard abbreviations. In this table, a plus sign (+) after a control name signifies that you cannot negate the control.

11.3 Using Compiler Controls

Controls to the compiler govern the format, processing, and content of both the input source file(s) and the output file(s). Certain controls override other controls even if they are explicitly specified. This section describes the use of controls and suggests which controls should be used during specific stages of program development.

Table 11-1. Types of Controls

Category	Primary Controls	General Controls
Listing Content	PRINT SYMBOLS XREF	LIST CODE
Listing Format	TITLE PAGEWIDTH PAGELENGTH	SUBTITLE EJECT
Input Format		INCLUDE FREEFORM
Object File	OBJECT DO66/DO77 STORAGE +INTERFACE ERRORLIMIT DEBUG TYPE INTERRUPT REENTRANT	OVERLAP
Control Status	IGNORE	

Table 11-2. Controls and Their Abbreviations

Control	Abbreviation
CODE	CO
DEBUG	DB
+DO66/DO77	none
+EJECT	EJ
ERRORLIMIT	EL
FREEFORM	FF
+IGNORE	IN
+INCLUDE	IC
+INTERRUPT	IT
LIST	LI
OBJECT	OJ
+PAGELENGTH	PL
+PAGEWIDTH	PW
PRINT	PR
+REENTRANT	RE
+STORAGE	SR
+SUBTITLE	ST
SYMBOLS	SB
+TITLE	TT
XREF	XR
+INTERFACE	ITF

11.3.1 Listing Device or File Selection

The PRINT control governs the selection of the file and device to receive printed output. To generate a listing that includes error messages and the source listing, use the PRINT control to specify the listing file, or allow the default PRINT control to send the listing to *source*.LST.

The NOPRINT control overrides all of the listing format controls described in 11.3.2 because it governs all printed output.

11.3.2 Controlling Listed Format and Content

If PRINT is active, the following controls govern the format and content of printed output:

CODE/NOCODE
EJECT
LIST/NOLIST
SUBTITLE('*subtitle*')
SYMBOLS/NOSYMBOLS
TITLE('*title*')
XREF/NOXREF

The default values specify listing of the source program without the assembly code listing (NOCODE), and without the symbol-table listing (NOSYMBOLS).

These default values assume the general case. If you need the assembly code listing of portions of the source file, use the CODE control. If you need to supress certain portions of the source listing, use NOLIST. Note that the NOLIST control does not override the CODE control.

The SYMBOLS control directs the compiler to produce a symbol-table listing as described in Section 11.4.18. NOSYMBOLS (the default) suppresses this action and NOPRINT overrides SYMBOLS.

Although paging is automatic, you can force a page eject on any line using the EJECT control. An EJECT in a control line is ignored if the control line occurs in an area governed by the NOLIST control. TITLE and SUBTITLE controls specify titles and subtitles in the listing. If NOLIST is in effect, the subtitle is saved until listing resumes with the LIST control. The compiler ignores all of these controls if NOPRINT is active.

11.3.3 Source Selection and Processing

The INCLUDE control governs the selection and processing of source files. There is only one primary source file but you can include other source files in the compilation by specifying them in INCLUDE controls.

The INCLUDE control must be the rightmost (last) control on a source control line. If controls are to the right of the INCLUDE control on a control line, the compiler issues a non-fatal error message and ignores the control.

11.3.4 Object Selection and Content

The following controls govern selection of the file to hold the object module, and the content of the object module, and the code generated.

DEBUG/NODEBUG	DO66/DO77	REENTRANT
INTERRUPT(*proc*[=*n*[,...])	STORAGE	
OBJECT(*file*)/NOOBJECT	INTERFACE	

The OBJECT control selects a file to receive the object module. The default file name has the same root name as the source file, with the extension OBJ. For example, if PROG1.SRC is the source file, PROG1.OBJ becomes the object file. NOOBJECT prevents the generation of an object module.

The INTERRUPT control enables you to compile specific procedures as interrupt procedures. Interrupt handling is discussed in Appendix I.

The DEBUG control generates debug records in the object module that are used by symbolic debuggers such as the ICE-86 emulator. The default value NODEBUG suppresses the generation of debug records. NOOBJECT overrides DEBUG.

DO66 and DO77 are primary controls that specify that all DO-loops in a program unit will conform to the ANSI 1966 or 1977 standard, respectively.

INTERFACE is a primary control that enhances the compatability of Fortran-86 with other programming languages. The INTERFACE control allows Fortran-86 programs to call procedures written in other languages. INTERFACE allows procedures written in other languages to call procedures written in Fortran-86. The calling conventions for procedures written in Pascal, Fortran, and PL/M are identical; therefore, the only language that needs a special designation for its calling convention is C.

The REENTRANT control specifies that reentrant code can be produced for a specified FUNCTION or SUBROUTINE.

The STORAGE control is a primary control that specifies default lengths, in bytes, as they apply to INTEGER and/or LOGICAL data items.

11.3.5 Use of Controls in Stages of Development

When you are compiling a program for the first time, use the default control settings with the following exception:

- Use XREF to generate a symbol and cross reference listing to aid your initial debugging effort.

As you develop and debug your program modules, you may use DEBUG to generate debug records for symbolic debugging. Selected source statements can be maintained in a separate file and included with the source file by using the INCLUDE control.

For quick compiling, you can maximize compilation speed by using default settings for all controls, with the following exception:

- Use NOPRINT to suppress printed output.

When preparing programs to test with the ICE-86 or ICE-88 emulators, use the CODE control to list the pseudo-assembly instructions and addresses.

Use the NOLIST control to save listing space by not listing portions of the source code that are already debugged. To make your listing more readable, use EJECT, TITLE, and SUBTITLE. You can direct the final listing to a specific output file using the PRINT control.

11.4 Control Definitions

The following sections present a description of each of the Fortran-86 compiler controls.

11.4.1 CODE/NOCODE

The CODE/NOCODE controls permit or prevent the listing of object code in pseudo-assembly language.

Syntax

```
CODE
NOCODE
```

Abbreviation

CO/NOCO

Default

NOCODE

Type

General

Description

The CODE control directs the compiler to produce a listing of the generated object code in pseudo-assembly language (a form that resembles the 8086 assembly language). This listing occurs only for portions of the source program where the CODE control is active; listing stops when a NOCODE is encountered. The pseudoassembly listing is appended to the source listing in the listing file created by the PRINT control (see Section 11.4.16, PRINT/NOPRINT).

The NOCODE control prevents the generation of this listing. If you specify neither control, the default is NOCODE.

The CODE control cannot create printed output if the NOPRINT control is in effect.

For an example of a listing in pseudo-assembly language, see Chapter 13.

11.4.2 DEBUG/NODEBUG

The DEBUG/NODEBUG controls generate debug records in the object module.

Syntax

```
DEBUG
NODEBUG
```

Abbreviation

DB/NODB

Default

NODEBUG

Type

Primary

Description

If an object file has been requested, the DEBUG control specifies that the object module will contain debug records. These records contain the name, data type, and relative address of each symbol in the program, and the statement number and relative address of each source program statement. This information can later be used for symbolic debugging of the source program using the ICE-86 emulator, DEBUG 86, or PSCOPE.

The default setting, NODEBUG, prevents generation of these records.

The compiler ignores the DEBUG control if the NOOBJECT control is in effect, since the compiler will not generate an object module.

NOTE

Array subscript references for the debugger must be written in reverse order. For example, in order to display the array element A(3,5) in the Fortran-86 program, you must use A(5,3) when communicating with the debugger. This is due to the reverse ordering of arrays in Fortran compared to other high-level languages. Intel debuggers are designed to support all high-level languages.

11.4.3 DO66/DO77

The DO66/DO77 controls specify that all DO-loops in a program must conform to the ANSI 1966 or 1977 standard, respectively.

Syntax

```
DO66
DO77
```

Abbreviation

none

Default

DO77

Type

Primary

Description

DO66 specifies that all DO-loops perform at least one iteration during execution, conforming to the ANSI 1966 standard.

DO77 permits zero iterations of DO-loops, which conforms to the ANSI 1977 standard.

11.4.4 EJECT

The EJECT control forces the start of a new page of printed output.

Syntax

EJECT[(*number*)]

Abbreviation

EJ

Default

paging as implied by the PAGELENGTH control

Type

General

Description

The EJECT control terminates the printing of the current page and starts a new page. The control line containing the EJECT control is the first line printed (following the page heading) on the new page.

If you do not use the EJECT control, a page eject will occur automatically as specified by the PAGELENGTH control.

The compiler ignores the EJECT control if the NOLIST or NOPRINT controls are in effect, since the compiler will not produce any printed output.

The EJECT control does not apply to the CODE listing.

11.4.5 ERRORLIMIT/NOERRORLIMIT

The ERRORLIMIT/NOERRORLIMIT controls terminate compilation prematurely after detecting a specified number of errors.

Syntax

```
ERRORLIMIT(number)
NOERRORLIMIT
```

Abbreviation

EL/NOEL

Default

NOERRORLIMIT

Type

Primary

Description

The ERRORLIMIT control enables the user to specify the number of compiler detected errors which will cause the compiler to cease compilation before a normal termination. The result of early termination can be incomplete PRINT listings, and all other compiler output will be deleted as if NOOBJECT were in effect.

The NOERRORLIMIT control allows compilation to continue until the end of the program regardless of the number of errors the compiler encounters.

11.4.6 FREEFORM/NOFREEFORM

The FREEFORM/NOFREEFORM controls permit or prevent entry of Fortran statements in a non-standard input format. (See Section 3.3.1, Line Format for a description of the Fortran-86 standard line format.)

Syntax

```
FREEFORM
NOFREEFORM
```

Abbreviation

FF/NOFF

Default

NOFREEFORM

Type

General

Description

Program statements after the FREEFORM control may begin in position 2 instead of position 7. Statement labels, continuation indicators (only the ampersand (&)), and comment indicators (both the asterisk (*) and the letter C) must begin in position 1. If a statement begins with any character except C, it may also start in column 1.

NOFREEFORM causes the compiler to issue error messages for all lines not conforming to the standard Fortran input format. Specifically, comment indicators (asterisk (*) and the letter C) belong in position 1, statement labels in positions 1-5, continuation line indicators in position 6, and statements in positions 7-132.

11.4.7 IGNORE

The IGNORE control allows specified general controls to be ignored by the compiler.

Syntax

IGNORE(*control*[, ...])

Abbreviation

IN

Default

None

Type

Primary

Description

The IGNORE control enables the user to specify certain general controls that will be ignored during the current compilation. If not specified otherwise prior to the appearance of the IGNORE control, the default settings for the specified controls will apply.

11.4.8 INCLUDE

The INCLUDE control adds other source files as input to the compiler.

Syntax

```
INCLUDE(file)
```

Abbreviation

IC

Default

no included files

Type

General

Description

When the compiler encounters the INCLUDE control in the source file, it reads from the other source file, *file*, until it reaches the end of that file. Then the compiler resumes reading the source lines that follow the INCLUDE control line in the original source file.

The INCLUDE control must be the rightmost control in the control line or the only control in that line.

The included file itself may contain INCLUDE controls, but the nesting of included files cannot exceed five (six included files).

The compiler always forces an end-of-line before reading from an included file.

END statements within INCLUDE files are ignored.

Your file must be a valid filename or an error will occur.

11.4.9 INTERFACE

The INTERFACE control allows Fortran-86 programs to call procedures written in other languages. In addition, INTERFACE allows procedures written in other languages to call procedures written in Fortran-86. The calling conventions for procedures written in Pascal, Fortran, and PL/M are identical; therefore, the only language that needs a special designation for its calling convention is C.

Syntax

INTERFACE (*lang* = *name* [*name*, ...])

where

lang	is the *name* of the language that requires a different calling convention for procedures; in this case, the language is C. Specifying INTERFACE for a language other than C has no effect; the standard calling convention for Fortran will be used.
name	represents the module(s) that will be called or referenced from the Fortran program. If *name* is the name of a Fortran subroutine or function, the compiler generates code that can be called according to the calling convention of the specified language. Any calls made to a subroutine or function with the specified name will use the calling conventions of the specified language unless the name is a dummy parameter to a subroutine or function.

Abbreviation

ITF

Default

none

Type

Primary

Description

Note that if other Fortran modules that are compiled separately reference the procedure, the name must be in an INTERFACE control that precedes the referring module. This control affects only the calling conventions for the names specified. INTERFACE does not affect the naming convention of procedures. (Many C compilers append an underline (_) to the name declared in the C program.)

11.4.10 INTERRUPT

The INTERRUPT control designates procedures as interrupt procedures.

Syntax

```
INTERRUPT [n]
```

Abbreviation

IT

Default

None

Type

General

Description

The INTERRUPT control allows you to specify procedures to be compiled as 8086 interrupt procedures.

Whatever procedure immediately follows the INTERRUPT control will be compiled with special prologue and epilogue code sequences so that it may be used to process interrupts during execution. In order for this to happen, however, you must associate each of your INTERRUPT procedures with the number of the interrupt it is designed to handle. This is done dynamically at run-time using the SETINT builtin procedure (see Section 6.1.2.4).

If *n* is specified, it represents the number of the interrupt associated with the specified procedure. The interrupt number must be a value between 0 and 255. You may also associate an interrupt procedure with an interrupt at run-time by using the SETINT built-in procedure (see Section 6.1.2.4).

11.4.11 LIST/NOLIST

The LIST/NOLIST controls permit or prevent the listing of source lines.

Syntax

```
LIST
NOLIST
```

Abbreviation

LI/NOLI

Default

LIST

Type

General

Description

The LIST control directs the compiler to begin or resume listing of the program with the next source line.

The NOLIST control directs the compiler to stop listing the program until the next occurrence, if any, of a LIST control.

When you specify neither control, or when LIST is in effect, the compiler lists all lines from the source file (or from a file read in with the INCLUDE control), including control lines. When NOLIST is in effect, the compiler lists only source lines associated with error messages.

The LIST control is ignored if the NOPRINT control is in effect.

The NOLIST control does affect the CODE control, which directs the compiler to produce a separate listing of the generated object code.

11.4.12 OBJECT/NOOBJECT

The OBJECT/NOOBJECT controls specify that an object module is to be created and the file name for that object module or prevent the creation of an object module.

Syntax

```
OBJECT[(filename)]
NOOBJECT
```

Abbreviation

OJ/NOOJ

Default

OBJECT (*source.OBJ*)

Type

Primary

Description

The OBJECT control directs the compiler to produce an object module. You can optionally specify a file for this object module by providing a legal filename (with optional device specifier) for file.

If you do not specify a file, or if you do not use the OBJECT control, the compiler will still produce the object module and direct it to the same disk or device as the source file, using filename *source.OBJ* (where source is the root name of the program text file).

The NOOBJECT control prevents the creation of an object module.

For details on the contents of the object modules, see Chapter 13, "Compiler Output."

11.4.13 OVERLAP/NOOVERLAP

The OVERLAP control enables porting of large programs to Fortran-86 without changes to the program logic.

Syntax

```
OVERLAP
NOOVERLAP
```

Abbreviation

OL/NOOL

Default

NOOVERLAP

Type

Module

Description

The OVERLAP control allows compilation of subprograms where a dummy variable or array element may be contained in more than one segment. OVERLAP allows the program to invoke special out-of-line run-time procedures for every reference to a dummy argument longer than one byte (except % VAL arguments).

Use this control only when the compiler requests it (compiler message F207), during a compilation of a program that refers to the subprogram.

The control is necessary when one or more of the actual arguments passed to the procedure has been allocated noncontiguous memory and requires special handling. The OVERLAP control is most likely to be needed with very large COMMON blocks, but also result from mixed-type EQUIVALENCE statements or odd-length CHARACTER arrays exceeding 64K bytes in size. See the description of the compiler message (F206) in Chapter 15 for alternative actions.

With the NOOVERLAP control, all dummy arguments are accessed directly from in-line instructions.

11.4.14 PAGELENGTH

The PAGELENGTH control specifies the maximum number of lines to appear on each page of the PRINT file.

Syntax

```
PAGELENGTH(n)
```

Abbreviation

PL

Default

PAGELENGTH(60)

Type

Primary

Description

The PAGELENGTH control enables the user to specify the maximum number of lines to appear on each page of the program listing. The minimum length is 5, which includes the four lines of each page heading. The maximum acceptable value for PAGELENGTH is 255 lines per page.

11.4.15 PAGEWIDTH

The PAGEWIDTH control specifies the maximum number of characters to appear on one line of the PRINT file.

Syntax

```
PAGEWIDTH(n)
```

Abbreviation

PW

Default

PAGEWIDTH(120)

Type

Primary

Description

The PAGEWIDTH control enables the user to specify the maximum number of characters to appear on one line of the program listing. The minimum width is 60. The maximum acceptable value for PAGEWIDTH is 132.

11.4.16 PRINT/NOPRINT

The PRINT/NOPRINT controls permit or prevent printed output, or select the device or file to receive printed output.

Syntax

```
PRINT[(filename)]
NOPRINT
```

Abbreviation

PR/NOPR

Default

PRINT(*source.LST*)

Type

Primary

Description

The PRINT control directs the compiler to produce printer output (listings), and the NOPRINT control stops the compiler from producing printed output. If you specify neither control, the compiler will produce listings and put them in a file that has the same name as the source input file, only with an LST extension. This new LST file will be created on the same device used for the source file. For example, if your source file is named *progrm* and it is on drive 1 (:F1:*progrm*), and you use neither control, or use only the simple PRINT control (the default), the compiler will create the listing as :F1:*progrm*.LST.

If you specify a PRINT control with a file in parentheses, the compiler will put the listings in the file or device named by file, which must be a legal filename for a file or device.

If you specify the NOPRINT control, the compiler will not produce listings—even if you specify other controls, such as LIST or CODE. If the NOPRINT control is in effect, the compiler will not produce any printed output. In addition, if you specify NOPRINT, error messages will not appear on the console.

11.4.17 REENTRANT

The REENTRANT control indicates that a particular SUBROUTINE or FUNCTION can call itself.

Syntax

```
REENTRANT
```

Abbreviation

RE

Default

none

Type

General

Description

The REENTRANT control indicates that reentrant code be produced for the specified FUNCTION or SUBROUTINE. That is, all local variables contained in these subprograms will be dynamically allocated on the run-time stack and removed at each RETURN statement.

11.4.18 STORAGE

The STORAGE control specifies default lengths, in bytes, applied to INTEGER and/ or LOGICAL data items.

Syntax

STORAGE(INTEGER **intlen*[,LOGICAL**loglen*])

or

STORAGE(LOGICAL **loglen*[,INTEGER**intlen*])

Abbreviation

SR

Default

STORAGE(INTEGER*2,LOGICAL*1)

Type

Primary

Description

The STORAGE control permits the user to specify the default lengths, in bytes, applicable to INTEGER and/or LOGICAL data items that are not explicitly implied by Fortran-86 type-statements or constant specifications.

Each length specification (*intlen* or *loglen*, above) may be 1, 2, or 4. INTEGER**intlen* may be abbreviated as Iintlen, and LOGICAL**loglen* may be abbreviated as Lloglen.

NOTE

The ANSI 1977 allocation requirements for "numeric storage units" imply STORAGE(INTEGER*4,LOGICAL*4).

11.4.19 SUBTITLE

The SUBTITLE control prints a subtitle on each page of printed output.

Syntax

```
SUBTITLE('text')
```

Abbreviation

ST

Default

SUBTITLE(' ')

Type

General

Description

The SUBTITLE control prints a subtitle on every page of printed output. To specify a subtitle, supply a sequence of printable ASCII characters (a string) for text, enclosed within apostrophes.

The compiler places the subtitle text on the subtitle line of each page of listed output, and truncates this subtitle on the right if necessary. You can specify a maximum length of 60 characters, but a narrow pagewidth may restrict this number further.

When a SUBTITLE control appears before the first noncontrol line in the source file, it puts the text on the first page and on all subsequent pages until the compiler encounters another SUBTITLE control. A subsequent SUBTITLE control causes a page eject, and the new text is put on the next page and on all following pages until another SUBTITLE control appears in the source program.

If the NOLIST control is in effect, the compiler saves this text and this text appears again as a subtitle when the listing resumes.

The SUBTITLE control does not apply to the CODE listing.

11.4.20 SYMBOLS/NOSYMBOLS

The SYMBOLS control provides a symbol-table listing of source program identifiers.

Syntax

```
SYMBOLS
NOSYMBOLS
```

Abbreviation

SB/NOSB

Default

NOSYMBOLS

Type

Primary

Description

The SYMBOLS control directs the compiler to produce a symbol-table listing of all identifiers and labels in the source program. The compiler prints an entry for each Fortran-86 constant, type, variable, argument, procedure, function, or label that occurs in the source program, in alphabetical order. The compiler appends this listing to the file that the PRINT control creates.

The NOSYMBOLS control prevents this symbol-table listing. The default setting is NOSYMBOLS.

11.4.21 TITLE

The TITLE control prints a title on each page of printed output.

Syntax

```
TITLE('text')
```

Abbreviation

TT

Default

module name

Type

Primary

Description

The TITLE control prints a title on every page of printed output. To specify a title, supply a sequence of printable ASCII characters (a string) for text, enclosed within apostrophes.

The compiler places the title text on the title line of each page of listed output, and truncates the title on the right, if necessary. You can specify a maximum length of 60 characters, but a narrow pagewidth may restrict this number further.

11.4.22 TYPE/NOTYPE

The control directs the compiler to include type records in the object modules. This allows link-time parameter type checking.

Syntax

```
TYPE
NOTYPE
```

Abbreviation

TY/NOTY

Default

TYPE

Type

Primary

Description

This TYPE records included in the object modules describe attributes of symbols used in the source program, and are used later for type checking by the linker. Type records provide a mechanism of promoting type compatibility between subprograms.

The TYPE control also enables internal type checking among multiple external procedure references.

The NOTYPE control prevents the inclusion of type records in the object module, and suppresses internal type checking.

NOTE

The type checking mechanism produces warning messages that are intended for convenience in debugging new programs. These messages may be ignored if you have observed the ANSI programming rules.

In particular, a valid array argument can produce a type-checking warning if the corresponding actual argument is an array element, or an array with a different dimension specification.

11.4.23 XREF/NOXREF

The XREF/NOXREF controls permit or prevent a symbol and cross reference listing of source program identifiers. The XREF control is equivalent to the SYMBOLS control.

Syntax

```
XREF
NOXREF
```

Abbreviation

XR/NOXR

Default

NOXREF

Type

Primary

Description

The XREF control directs the complier to produce an alphabetical listing of all the symbols defined in the program and their attributes cross-referenced with numbers of all the source statements that reference them. The compiler appends this listing to the file that the PRINT control creates. (See PRINT/NOPRINT, Section 11.4.16). XREF is ignored when NOPRINT is used.

The NOXREF control prevents this symbol-table listing. The default setting is NOXREF.

CHAPTER 12
COMPILER OPERATION

You create a Fortran-86 program by typing instructions into a file using a text editor and submitting the file to the Fortran-86 compiler. The compiler accepts the source code for processing. A single object file results from this compilation. After the linker and locater process the object file, the code is considered executable object code, implying that your Fortran-86 program can be run.

Chapter 1 of this manual describes the software development process and the system specific appendix explains compiler invocation.

12.1 Input Files

You supply the name of the Fortran-86 source program in the invocation line. You can also include other source files by using the INCLUDE control, as described in Section 11.4.8. These files must be standard operating system files containing the text of Fortran-86 statements.

The Fortran-86 compiler expects a source file consisting of a sequence of *program units*, i.e., BLOCK DATA subprograms, FUNCTION subprograms, SUBROUTINE subprograms, and/or a main program. The compiler processes each program unit independently. Comment lines and compiler control lines may appear anywhere in program units, but the compiler assumes that any comments found after an END statement belong to the next program unit.

Ordinarily, program text lines must be in the standard ANSI Fortran 77 format:

- Positions 1 through 5 contain the statement number.
- Position 6 indicates statement continuation.
- Positions 7 through 132 consist of the actual Fortran statement.

The FREEFORM control (see Section 11.4.6). permits you to write source code in a more convenient format for terminal entry following these guidelines:

- If the statement has a label, position 1 must contain the label number.
- If the line is a continuation line, position 1 must contain an ampersand (&).
- If the line is a control line, position 1 must contain a dollar sign ($).
- Actual statements can begin in position 2, or in position 1 if the first character is not C.

Comment lines are the same in both formats; the first character must be either a C or an asterisk (*).

Once you have entered your source code into a text file, you can invoke the compiler, as described earlier, to process your program.

12.2 Output Files

Unless you use specific controls to suppress them, the compiler produces two output files: the object file and the listing file.

The listing file, or PRINT file, contains a listing of the source program and any other printed output generated by the compiler as specified by the listing selection controls described in Chapter 11. The object file contains the actual code in object module format. The system can execute the object file after you use the linking and locating facilities described in Chapter 14. The compiler output files are described in greater detail in Chapter 13.

The listing file and the object file, unless changed by the PRINT or OBJECT controls (see Sections 11.4.14 and 11.4.11), have the same basic name as the source file, with different extensions. The listing file has the extension LST and the object file has the extension OBJ. The compiler creates both files if they do not exist, or overwrites them if they do. By default, the compiler places these files on the same drive as the source file.

The system specific appendix provides examples.

12.3 Work Files

The compiler creates and uses work files during its operation and deletes them upon the completion of compilation. These files are designated :WORK:, so they do not conflict with your files. See Chapter 13, "Compiler Output," for more specific information about Fortran-86 work files.

12.4 Compiler Messages

When you invoke the compiler, it displays the following sign-on message:

```
system FORTRAN-86 COMPILER, Vx.y
```

where *system* is the operating system, *x* is the compiler version number, and *y* is the change number within the version.

When a compilation is finished, the compiler terminates with the following message:

```
m TOTAL ERRORS DETECTED
n TOTAL WARNINGS DETECTED
 END OF FORTRAN-86 COMPILATION
```

Chapter 15 lists all of the compiler errors.

CHAPTER 13 COMPILER OUTPUT

During compilation, the compiler produces a listing of the source program and an object module. Compiler controls can affect both the listing and object files. These controls are described in detail in Chapter 11. This chapter discusses the contents of these files.

13.1 Program Listing

Unless you specified the NOPRINT control (see Section 11.4.14), the program listing file is either the file you defined with a PRINT control or the default listing file.

The listing file contains, minimally, a "sign-on" preface, any syntactic error messages, a compilation summary, and a sign-off message. You modify the listing by specifying different controls. If the LIST control is active, the compiler produces a program source listing. If the CODE control is active, a pseudo-assembly language listing of the source code is also created. If the SYMBOLS control is active, the listing file includes a listing of all symbols used in the program. NOLIST and NOCODE supress these listings, respectively.

If the NOPRINT control is active, no listing file is produced. Any error messages appear on the system console (:CO:).

Paging occurs automatically during the source and symbol-table listings, but you can force a page eject using the EJECT or SUBTITLE controls. The following sections describe each part of the listing file in detail.

13.1.1 Listing Preface

Each page of the listing file has a numbered page header identifying the compiler, the subprogram currently being compiled, the date and time of the compilation, and optionally, a title and subtitle. The compiler truncates the title and subtitle to 60 characters or less depending on the pagewidth setting. The page heading is followed by two blank lines. The following is the Fortran-86 header:

```
system FORTRAN-86 COMPILER title      date/time PAGEnnn
filename           subtitle   modulename
```

where

system	is the name of the operating system.
title	is the name you specified in the TITLE control.
subtitle	is the name you specified in the SUBTITLE control.
date/time	is the running date and the starting time supplied and updated by the operating system.
filename	is the name of your source program.
modulename	is the name of your (sub)program.
nnn	is the number of pages in the PRINT file.

13.1.2 Source Listing

The source listing includes the source code of the module being compiled, any errors detected during compilation, and optional symbol-table and pseudo-assembly listings.

Source lines appear as they do in the Fortran-86 input file with the following additions:

Positions 1-4 contain a statement number for each Fortran-86 statement. The compiler associates each Fortran statement printed or not with a unique statement number, and prints it at the beginning of that statement. Error messages refer to these statement numbers, not to statement labels coded as part of the Fortran-86 program.

If an INCLUDE control inserted a line into the source code, an equal sign (=) and a digit indicating the nesting level of the INCLUDE follow the statement number in positions 5-6.

Position 7 contains a hyphen (-) if the compiler continued the line on another line because of a PAGEWIDTH limitation.

The remainder of the listing line, beginning with position 8, contains the source code as read (or added using the INCLUDE control) from the Fortran-86 text file. However, any ASCII TAB characters are expanded to multiple blanks, as necessary, to reach the next character position, which is a multiple of eight.

13.1.3 Symbol Listing

If you specified the SYMBOLS control, the compiler creates a listing with an entry for each variable, array, function, subroutine and run-time procedure that appears in the source program. These are in ASCII sequence by symbol name or statement number you defined in the program. Each entry includes the following:

- the source identifier (symbol)
- the kind (label, array, etc.)
- the data type (integer, logical, etc.)
- the length in bytes
- the scope (external, common, etc.)
- the address relative to the beginning of the segment
- the statement number of its declaration

Additionally, the compiler produces a separate listing of run-time procedures referenced in the program. The run-time procedure listing provides helpful support for identifying critical areas for reducing program size. Each procedure name has one or more modules associated with it, all of which are required to fulfill the function for which the first module was called. The user can identify these modules, using the Run-Time Module Directory, and can determine their sizes by using the LINK86 map.

13.1.4 Pseudo-Assembly Language Listing

If you specified the CODE control, the compiler generates a pseudo-assembly language equivalent of the compiler-generated object code. The list-formatting controls TITLE, PAGEWIDTH, and PAGELENGTH apply to the CODE listing as well as to the source listing.

The pseudo-assembly listing for each program unit always begins on a new page. A comment line with the statement number of the corresponding source statement will head the code resulting from each source statement.

The code listing conforms to standard assembly-language format of six columns of information, although not all six of these columns will necessarily apply to every line of the listing. The columns of information are

- Relocatable location counter (hexadecimal notation)
- Resultant binary code (hexadecimal notation)
- Label field
- Symbolic operation code (mnemonic notation)
- Symbolic arguments
- Comment field

If you used the CODE control, the compiler generates the appropriate assembly directives to declare local symbols and constants in the listing. An at-sign (@) precedes compiler-generated labels, such as those which mark the beginning and ending of a DO loop. A question mark (?) precedes source-program statement labels to distinguish them from compiler-generated labels and numeric constants. Comments appearing on PUSH and POP instructions indicate the stack depth associated with the stack reference.

Figure 13-1 shows a portion of the pseudo-assembly listing for a sample Fortran-86 program, along with the source lines from which it was generated.

```
FORTRAN-86 COMPILER        GENERATED CODE
:F1:PROG2.FTN

                                             ; STATEMENT # 1
003D  8BEC              MOV     BP,SP
003F  9A00000000        CALL    INITFP
0044  9A00000000        CALL    TQ_001
0049  FB                STI
                                             ; STATEMENT # 5
004A  9A00000000        CALL    GETDAT
                                             ; STATEMENT # 6
004F  9BD9EE            FLDZ                 ; 7
0052  9BDD1E0000        FSTP    DPRES        ; 7
0057  9B                WAIT
                                             ; STATEMENT # 7
0058  9BD9EE            FLDZ                 ; 7
005B  9BDB3E1200        FSTP    TMPRES       ; 7
0060  9B                WAIT
                                             ; STATEMENT # 8
0061  C70610000100      MOV     I,1H
                 @@000000:
0067  8B061000          MOV     AX,I
006B  81F8F401          CMP     AX,1F4H
006F  7E03              JLE     $+5H
0071  E93C00            JMP     @@000001
                                             ; STATEMENT # 9
0074  87D8              XCHG    BX,AX
0076  B80800            MOV     AX,8H
0079  F7EB              IMUL    BX
007B  2E8E063900        MOV     ES,CS:@CONST+39H
0080  9B26DD061000      FLD     ES:RTOTAL
0086  87D8              XCHG    BX,AX
0088  9B26DC7F10        FDIVR   ES:RARRAY[BX-8H]; 7
008D  9BDDD1            FST     @TOS+1H
0090  9BDC060000        FADD    DPRES        ; 7
0095  9BDD1E0000        FSTP    DPRES        ; 7
009A  9B                WAIT
                                             ; STATEMENT # 10
009B  9BDB2E1200        FLD     TMPRES       ; 7
00A0  9BDEC1            FADDP                ; 7
00A3  9BDB3E1200        FSTP    TMPRES
00A8  9B                WAIT
                 ?10:
```

Figure 13-1. Sample Portion of a Code Listing

```
FORTRAN-86 COMPILER        GENERATED CODE
:F1:PROG2.FTN

                                                ; STATEMENT # 11
00A9  FF061000          INC        I
00AD  E9B7FF            JMP        @@000000
            @@000001:
                                                ; STATEMENT # 12
00B0  9BDB2E1200        FLD        TMPRES   ; 7
00B5  9BDD1E0800        FSTP       RESULT   ; 7
00BA  9B                WAIT
                                                ; STATEMENT # 13
00BB  0E                PUSH       CS       ; 1
00BC  07                POP        ES       ; 1
00BD  8D360000          LEA        SI,?100
00C1  B006              MOV        AL,6H
00C3  9A00000000        CALL       FQ_112
00C8  9BDD060800        FLD        RESULT   ; 7
00CD  9A00000000        CALL       FQ_320
00D2  9BDD060000        FLD        DPRES    ; 7
00D7  9A00000000        CALL       FQ_320
00DC  9A00000000        CALL       FQ_901
                                                ; STATEMENT # 15
00E1  9BDD060800        FLD        RESULT   ; 7
00E6  9BDC2E0000        FSUBR      DPRES    ; 7
00EB  9BDD1E0800        FSTP       RESULT   ; 7
00F0  9B                WAIT
                                                ; STATEMENT # 16
00F1  0E                PUSH       CS       ; 1
00F2  07                POP        ES       ; 1
00F3  8D362500          LEA        SI,?200
00F7  B006              MOV        AL,6H
00F9  9A00000000        CALL       FQ_112
00FE  9BDD060800        FLD        RESULT   ; 7
0103  9A00000000        CALL       FQ_320
0108  9A00000000        CALL       FQ_901
                                                ; STATEMENT # 18
010D  9A00000000        CALL       TQ_999

                                                ; STATEMENT # 1
0087  1E                PUSH       DS
0088  2E8E1E8500        MOV        DS,CS:@@DATA$FRAME
008D  55                PUSH       BP
008E  8BEC              MOV        BP,SP
                                                ; STATEMENT # 5
0090  9BD9EE            FLDZ                ; 7
0093  9BDB3E1200        FSTP       TMPTOT   ; 7
0098  9B                WAIT
                                                ; STATEMENT # 6
0099  0E                PUSH       CS       ; 1
009A  07                POP        ES       ; 1
009B  8D360000          LEA        SI,?100
009F  B006              MOV        AL,6H
00A1  9A00000000        CALL       FQ_112
00A6  9A00000000        CALL       FQ_901
                                                ; STATEMENT # 8
00AB  0E                PUSH       CS       ; 1
00AC  07                POP        ES       ; 1
00AD  8D363C00          LEA        SI,?200
00B1  B005              MOV        AL,5H
00B3  9A00000000        CALL       FQ_110
00B8  9A00000000        CALL       FQ_318
00BD  9BDD1E0800        FSTP       RVALUE   ; 7
00C2  9B                WAIT
00C3  9A00000000        CALL       FQ_901
                                                ; STATEMENT # 10
00C8  0E                PUSH       CS       ; 1
00C9  07                POP        ES       ; 1
00CA  8D364000          LEA        SI,?300
00CE  B006              MOV        AL,6H
00D0  9A00000000        CALL       FQ_112
00D5  9A00000000        CALL       FQ_901
                                                ; STATEMENT # 12
00DA  0E                PUSH       CS       ; 1
00DB  07                POP        ES       ; 1
00DC  8D363C00          LEA        SI,?200
00E0  B005              MOV        AL,5H
00E2  9A00000000        CALL       FQ_110
```

Figure 13-1. Sample Portion of a Code Listing (Cont'd.)

```
FORTRAN-86 COMPILER           GENERATED CODE
:F1:PROG2.FTN

00E7  9A00000000         CALL     FQ_318
00EC  9BDD1E0000         FSTP     FACTOR  ; 7
00F1  9B                 WAIT
00F2  9A00000000         CALL     FQ_901
                                          ; STATEMENT # 13
00F7  C70610000100       MOV      I,1H
            @@000002:
00FD  8B061000           MOV      AX,I
0101  81F8F401           CMP      AX,1F4H
0105  7E03               JLE      $+5H
0107  E94300             JMP      @@000003
                                          ; STATEMENT # 14
010A  87D8               XCHG     BX,AX
010C  B80800             MOV      AX,8H
010F  F7EB               IMUL     BX
0111  87D8               XCHG     BX,AX
0113  9BDD060800         FLD      RVALUE  ; 7
0118  2E8E068300         MOV      ES,CS:@CONST+83H
011D  9B26DD5F10         FSTP     ES:RARRAY[BX-8H]; 7

0122  9B                 WAIT
                                          ; STATEMENT # 15
0123  9BDD060800         FLD      RVALUE  ; 7
0128  9BDB2E1200         FLD      TMPTOT  ; 6
012D  9BDEC1             FADDP            ; 6
0130  9BDB3E1200         FSTP     TMPTOT  ; 7
0135  9B                 WAIT
                                          ; STATEMENT # 16
0136  9BDD060000         FLD      FACTOR  ; 7
013B  9BDC0E0800         FMUL     RVALUE  ; 7
0140  9BDD1E0800         FSTP     RVALUE  ; 7
0145  9B                 WAIT
            ?10:
                                          ; STATEMENT # 17
0146  FF061000           INC      I
014A  E9B0FF             JMP      @@000002
            @@000003:
                                          ; STATEMENT # 18
014D  9BDB2E1200         FLD      TMPTOT  ; 7
0152  2E8E068300         MOV      ES,CS:@CONST+83H
0157  9B26DD1E1000       FSTP     ES:RTOTAL; 7
015D  9B                 WAIT
                                          ; STATEMENT # 19
015E  8BE5               MOV      SP,BP
0160  5D                 POP      BP
0161  1F                 POP      DS
0162  CB                 RET
```

Figure 13-1. Sample Portion of a Code Listing (Cont'd.)

13.1.5 Error Message Listing

Error messages for your compiled Fortran-86 program appear after the source listing. The compiler controls PAGEWIDTH, PAGELENGTH, and TITLE apply to the error-message listing as well.

The format for the error messages is as follows:

[STATEMENT *n*][, NEAR *symbol*], *errortype m*: *message*

where

errortype	is either ERROR or WARNING.
m	is the specific error or warning number.
n	is the internal number of the statement containing the error.
symbol	is a pointer to the location of the error within the statement.
message	is the actual error message (see Chapter 15, "Error Messages").

13.1.6 Compilation Summary

The compiler generates the following messages at the end of each program listing:

```
STORAGE REQUIREMENTS FOR MODULE module:

     CODE AREA SIZE                 xxxxH          yyyyD
     VARIABLE AREA SIZE             xxxxH          yyyyD
     MAXIMUM STACK SIZE             xxxxH          yyyyD
     //                             xxxxH          yyyyD
     / SEGMENTNAME/                 xxxxH          yyyyD

 mmm ERRORS DETECTED.
 nnn WARNINGS DETECTED.
 ENTRY POINT IS x.
[FLOATING POINT OPERATIONS WERE GENERATED.]
 COMPILATION OF module status.
```

In this message, *module* is the name of the compiled module. The module size appears in both hexadecimal, *xxxx*, and decimal, *yyyy*. The compiler differentiates between the number of errors, *mmm*, and the number of warnings, *nnn*, showing both. The *status* of the compilation can be completed or aborted if the compiler detected any errors.

When the dictionary exceeds compile-time memory capacity, a temporary spill file is opened on the disk. When this happens, the following message appears on the screen:

```
DICTIONARY SPILLING TO DISK
```

This message indicates that compilation time will be noticeably greater than normal.

13.1.7 Sign-Off Message

The compiler prints the sign-off message, as described in Section 12.5, at the end of the listing.

13.2 Object Files

The Fortran-86 compiler outputs a file containing relocatable object modules. By linking this file with the Fortran-86 run-time libraries and other relocatable files, you can produce a single executable object module.

Each source file submitted to the compiler produces one object file. Each program unit in the source file produces one object module in the object file. Object modules have the same names as their respective program units. For a module of an unnamed main program or BLOCK DATA subprogram, the compiler assigns the names, @ MAIN or @ BLOCKDATA, respectively.

Each object module generated by the compiler will contain one each of the following 8086 segments:

- A CODE segment
- A DATA segment
- A STACK segment

The CODE segment is named *programname* _CODE, and the DATA segment, *programname* _DATA. (Multiple DATA segments are named *programname* _DATA*n* (where *n*=1, 2, ...). Each COMMON block is a separate segment named @ *common-name*, with a single @ for blank COMMON.

Local arrays and COMMON blocks exceeding 64K bytes in size are allocated on multiple, chained segments. The first such segment is named as described above; each successive segment has the same name, but with the suffixes @0FL*n* (where *n*=1, 2, ...).

The following class definitions appear for your convenience in case you want to locate your program with absolute addresses:

- CODE—consisting of all CODE segments (including constants)
- DATA—consisting of all DATA segments
- STACK—consisting of the STACK segment
- COMMON—consisting of all COMMON segments

You specify generation of object files using the OBJECT control (see Section 11.4.11). The compiler will not produce an object file if you specify the NOOBJECT control.

13.3 Work Files

The compiler temporarily allocates work files and deletes them when they are no longer needed or at the termination of the compilation. Up to six work files can be allocated. The system specific appendix provides examples.

CHAPTER 14 LINKING, RELOCATING, AND EXECUTING PROGRAMS

14.1 Introduction

Before you can execute your Fortran-86 program, you must link the object modules and optionally locate them in memory. The compiled modules that make up your final program need not be written in the same language. You can freely link together programs written in Fortran-86, Pascal-86, PL/M-86 or assembly language to make the most efficient use of language features. Additionally, some built-in Fortran-86 functions reside in the run-time support libraries which you must link with your object code before the program can be executed successfully.

Intel provides the utilities necessary for linking your program, locating it in memory, and loading it for execution. These utilities are listed in the system specific appendixes.

The 8086-based linker and locater are described in detail in the *iAPX 86,88 Family Utilities User's Guide.* This guide also provides an overview of 8086 memory addressing techniques, definitions of segments, classes, and groups, discussions of segment, class, and group combining, and descriptions of how the locator binds segments to addresses. The utilities guide also descibes the mechanics of loading and executing programs and the maintenance of program libraries using the 8086 resident library utility and the object-code print utility.

14.2 Memory Allocation

Each Fortran-86 compilation allocates the memory for the program unit in several independent, relocatable segments. They are CODE, DATA, STACK, blank COMMON, and named COMMON.

The CODE segment contains the executable object code for your Fortran-86 program. The compiler also places all data constants in the CODE segment. Format specifications from FORMAT statements are also in this segment.

The compiler allocates memory in DATA segments for all local variables and arrays, except those in subprograms compiled while the REENTRANT control is active. The compiler places temporary storage for intermediate values and copies of argument addresses in the STACK segment.

The blank COMMON segment holds all variables and arrays in blank COMMON blocks. For named COMMON blocks, the compiler allocates all variables and arrays to separate COMMON segments corresponding to the names you supplied for those COMMON blocks.

In addition to the Fortran-86 segments (CODE, DATA, STACK, and COMMON), the relocatable object module may contain other segments. These are segments provided by the Fortran-86 run-time libraries and user modules originally written in other languages.

14.3 Linking Object Modules

The 8086-based linker (LINK86) produces a single output module. While combining modules, the linker adjusts all addresses to be relative to the beginning of the segments in the new output module. The linker also searches libraries for modules that resolve external references in the modules being combined, and includes the new modules in the output file. Throughout this process, the linker generates a link map, and error messages for abnormal conditions.

The output module can be processed by the 8086-based locater (LOC86), which assigns absolute memory locations to the code in the object module. The output file from the locater can be passed again to the linker (LINK86) to be combined with other modules into an expanded output module. The linked module may be executed on the iRMX 86 operating system without locating (LOC86) if the BIND control is used.

14.3.1 Use of Libraries

A library is a file containing object modules. It is created and maintained by the library utility, LIB86. You use the libraries to build your programs by referring to the object modules as external procedures in your programs and linking the libraries to your programs.

The linker treats library files in a special manner. When you specify input modules to the linker, the linker combines them while keeping track of all external references. When a library file is included as input to the linker, the linker searches the library for modules that satisfy these unresolved external references. This means that libraries should be specified to the linker after the input primary modules. If a module has an external reference to another module in the library, the linker searches the library again to try to satisfy the reference. The process continues until all external references are satisfied, or until the linker cannot find any more public symbols to satisfy an external reference.

The library utility is described in detail in the *iAPX 86,88 Family Utilities User's Guide* (121616).

14.3.2 Run-Time Support Libraries

Intel supplies libraries that provide run-time support for Fortran-86 modules. The run-time support is divided into separate libraries so that you can link in the appropriate libraries for your application. You do not have to maintain these libraries using LIB86, since they are already supplied as libraries.

A list of all run-time libraries follows:

CEL87.LIB and EH87.LIB are required to support floating-point and error handling functions.

F86RN0.LIB, F86RN1.LIB, and F86RN2.LIB are required for any run-time support. These libraries provide Fortran run-time support for I/O, internal I/O, intrinsic functions, 32-bit integer arithmetic, character strings, and multiple segment variables.

F86RN3.LIB and F86RN4.LIB are the default logical record system libraries. For more information see the *Run-Time Support iAPX 86,88* (121776).

RTNULL.LIB instead of F86RN3.LIB and F86RN4.LIB to resolve external references when you do not use external I/O or if you intend to provide your own logical record interface.

8087.LIB is required to support floating-point arithmetic with the 8087 Numeric Data Processor. When using the 8087 Emulator use the E8087 and the module

8087.LIB is required to support floating-point arithmetic with the 8087 Numeric Data Processor. When using the 8087 Emulator use the E8087 and the module E8087.LIB instead of the 8087.LIB. If you are not performing any floating-point arithmetic, use 87NULL.LIB. DCON87.LIB is not normally needed for Fortran support. This library provides functions that convert floating-point numbers from ASCII to floating-point representation, and vice-versa. See the *8087 Support Library Reference Manual*, order number 121725, for additional information.

LARGE.LIB is required to execute Fortran-86 programs in the Series III environment when using F86RN3.LIB and F86RN4.LIB. Do not use LARGE.LIB if you linked in RTNULL.LIB (for no run-time support), except when using your own run-time support libraries that rely on the Universal Development System Interface (UDI) or make UDI calls in the program.

LARGE.LIB is required to execute Fortran-86 programs in an iRMX-86 environment. Do not use LARGE.LIB if you linked in RTNULL.LIB (for no runtime support), except when using your own run-time support libraries that rely on the UDI, or if your program makes UDI calls.

14.3.3 Linking with Overlays

In some cases a Fortran-86 program will be too large to fit into the available memory on the system. The overlay feature, provided by LINK86, provides a mechanism for memory usage that allows these programs to fit on the system by allowing different parts of a program to share the same memory. Overlayed programs consist of a single root portion and multiple overlays. The root is in memory for the entire execution of the program. All of the overlays share the same memory, so only one overlay can be in memory at a time.

When linking libraries in overlays, you can use the ASSUMEROOT control of LINK86 to prevent modules that have been linked into the root from being linked into an overlay as well. However, two of the Fortran-86 libraries (F86RN2.LIB and F86RN4.LIB) as well as EH87.LIB contain dummy versions of run-time routines. The run-time libraries are set up only to link in the run-time modules that are actually needed by the program. The dummy modules contained in these libraries resolve external references to routines that are not needed. Normally these routines will not be called and will cause an exception if they are called. However, if a routine is needed by a module in an overlay and the dummy version has already been linked into the root, the real version will not be included because LINK86 cannot distinguish the real module from the dummy module.

Linking the real versions of these modules, whether they are needed or not, avoids this complication. Each of the libraries that contains dummy modules includes a module that has external references to all of the public symbols that have dummy versions. By linking in this module before linking the library containing the real versions of these routines, you can force all of the real routines to be linked into the root. Figure 14-1 illustrates this process in a Fortran-86 program.

14.3.3.1 Overlay Restrictions

The SAVE statement will not save the values of variables in an overlay if the overlay is reloaded. COMMON block values are not saved when a new overlay is loaded (even if the common block is specified in a SAVE statement), unless the COMMON block also is specified in the root.

```
;
;  LINK ROOT
;
LINK86  :f1:root.OBJ, &
            EH87.LIB(TQINSTRUCTION_RETRY), &
            CEL87.LIB,F86RN0.LIB, &
            F86RN2.LIB(INPUT_EDIT_TABLE,OUTPUT_EDIT_TABLE), &
            F86RN1.LIB,F86RN2.LIB, &
            F86RN4.LIB(FORMAT_SEQ_DEVICE_DRIVER, &
                       UNFORMAT_SEQ_DEVICE_DRIVER), &
            F86RN3.LIB,F86RN4.LIB, &
            EH87.LIB,E8087,E8087.LIB,LARGE.LIB   TO  :f1:root.LNK &
        OVERLAY(ROOT)   NOBIND
;
;  LINK OVERLAY 1
;
LINK86    :f1:ovl.OBJ,CEL87.LIB,F86RN0.LIB, &
           F86RN1.LIB,F86RN2.LIB,F86RN3.LIB,F86RN4.LIB, &
           E8087.LIB,LARGE.LIB    TO     :f1:ovl.LNK &
        OVERLAY(OVERLAY1)  ASSUMEROOT(:f1:root.LNK)  NOBIND
;
;  LINK OVERLAY 2
;
LINK86    :f1:ov2.OBJ,CEL87.LIB.F86RN0.LIB, &
           F86RN1.LIB,F86RN2.LIB,F86RN3.LIB,F86RN4.LIB, &
           E8087.LIB,LARGE.LIB    TO     :f1:ov2.LNK &
        OVERLAY(OVERLAY2)   ASSUMEROOT(:f1:root.LNK)  NOBIND
;
;  FINAL LINK
;
LINK86    :f1:root.LNK,:f1:ovl.LNK,:f1:ov2.LNK    TO    root.86   BIND
```

Figure 14-1. Using the Overlay Feature to Link Fortran-86 Object Modules

14.3.4 Linking with Fortran Procedures

The following link operation takes two object modules, MYMOD1.OBJ and MYMOD2.OBJ, links them together, than links in the Fortran run-time libraries to form the output module MYPROG.86. To extend the LINK86 command to the next line without transmitting the command, type the ampersand (&) character before the RETURN key, and continue typing the command on the next line (do not type the ampersand character between letters of a filename). The continued line will start with an angle bracket (>).

Series III:

```
-RUN LINK86 MYMOD1.OBJ, MYMOD2.OBJ, F86RN0.LIB, & <cr>
> F86RN1.LIB, F86RN2.LIB, F86RN3.LIB, F86RN4.LIB, & <cr>
>87NULL.LIB, LARGE.LIB TO MYPROG.86 BIND <cr>
```

Series IV:

```
-LINK86 MYMOD1.OBJ, MYMOD2.OBJ, F86RN0.LIB, & <cr>
> F86RN1.LIB, F86RN2.LIB, F86RN3.LIB, F86RN4.LIB, & <cr>
>87NULL.LIB, LARGE.LIB TO MYPROG.86 BIND <cr>
```

The linker first reads MYMOD1.OBJ and MYMOD2.OBJ for external references and resolves those references. Then, the linker attempts to resolve any more external references in the modules by looking at the public symbols in the libraries F86RN0.LIB, F86RN1.LIB, F86RN2.LIB, F86RN3.LIB, F86RN4.LIB, 87NULL.LIB, and LARGE.LIB. Use the 87NULL.LIB when the modules do not perform real arithmetic. The final output module is MYPROG.86. This module can be loaded and executed on the Series III.

When the modules MYMOD1.OBJ and MYMOD2.OBJ do perform real arithmetic, link them with the 8087 Numeric Data Processor or the 8087 Emulator. The LINK86 command when using the emulator is:

Series III:

```
-RUN LINK86 MYMOD1.OBJ, MYMOD2.OBJ, F86RN0.LIB, &<cr>
> F86RN1.LIB, F86RN.LIB, F86RN3.LIB, F86RN4.LIB, & <cr>
> EH87.LIB, E8087, E8087.LIB, LARGE.LIB TO MYPROG.86 BIND <cr>
```

Series IV:

```
-LINK86 MYMOD1.OBJ, MYMOD2.OBJ, F86RN0.LIB, &<cr>
> F86RN1.LIB, F86RN.LIB, F86RN3.LIB, F86RN4.LIB, & <cr>
> EH87.LIB, E8087, E8087.LIB, LARGE.LIB TO MYPROG.86 BIND <cr>
```

To support real arithmetic when using the 8087, replace E8087 and E8087.LIB with 8087.LIB. EH87.LIB provides exceptions handling support for the 8087 Numeric Data Processor or its emulator. This is the link sequence that should be used in a full-featured operating system. The BIND option used with LINK86 provides an output file that is ready to be executed (if the operating system has an LTL loader).

14.3.5 Linking with Non-Fortran Procedures

The relocatable object modules produced by the Fortran-86 compiler are compatible with those generated by the Pascal-86 compiler, the PL/M-86 compiler, and the 8086/8087/8088 Macro Assembler. You can link together modules written in these iAPX 86,88 family languages. This feature allows you to use Fortran-86 to code those segments of your application to which the features of Fortran-86 are particularly well suited: multidimensional arrays, formatted and direct access I/O, floating-point arithmetic, and/or Fortran-86 intrinsic functions. Other facets of your programming task can be written in another language with no loss of compatibility.

Pascal-86 subprograms must be linked to the first two Pascal-86 run-time libraries before linking to Fortran-86 object files, as follows:

```
LINK86 PASMOD.OBJ, P86RN0.LIB, P86RN1.LIB, &
        TO PASMOD.LNK NOPUBLICS EXCEPT names
```

where

names	are those names that are referenced by the Fortran program.

PASMOD.LNK is then used in the Fortran-86 LINK86 command in the same way as Fortran-86 object files.

A one-LINK-step alternative:

```
LINK86 FTNMOD.OBJ, CEL87.LIB, F86RN0.LIB, F86RN1.LIB, &
       F86RN2.LIB, PASMOD.OBJ, P86RN0.LIB, P86RN1.LIB, &
       F86RN3.LIB, F86RN4.LIB, 8087.LIB, etc.
```

For more specific information about mixing Fortran-86 subprograms with subprograms in other languages, see Appendix H of this manual.

Be sure to invoke the RUN utility (as shown above) when linking in a Series III.

14.4 Locating Object Modules

The 8086-based locater (LOC86) binds locatable segments to absolute memory addresses. The locater creates an absolute output module from a single input module, generates a memory map that summarizes the results of address binding, produces a symbol table that shows the addresses of certain symbols, detects any errors that arise in the locating process, and filters locating information and compiler-generated debugging information. The locating process is described in detail in Chapter 3 of the *iAPX 86,88 Family Utilities User's Guide for 8086-Based Development Systems.*

The output module from the locater is a program that you can load and execute. The system specific appendix provides examples.

The locator includes several controls that enable you to specify exactly where portions of your program will be located in memory. These controls can be specified as part of the command syntax to the locater. This section describes specific considerations for locating Fortran-86 object modules.

The ORDER control allows you to dictate the sequence of segment types in memory. The format of this control is as follows:

```
ORDER(segids)
```

where

segids	is some combination of the segment names CODE, DATA, */commonname/* (for a named COMMON), // (for blank COMMON), and STACK.

If you do not specify the ORDER control, the system locates module segments sequentially in memory in the following order: CODE, STACK, COMMON, and DATA; the term COMMON means all COMMON segments in an arbitrary order.

The ORDER list can be partial; you need not list all module segments. In this case, the locater takes all segments specified in the ORDER control in the order specified. It takes the remaining segments in the default order, after the modules listed in the ORDER control.

14.5 Preconnecting Files

Fortran-86 I/O statements operate on device units that are connected to files on a one-to-one basis. A unit-to-file connection can be made when the file is opened (by the OPEN statement) or by preconnecting the unit to the file at run-time.

In the Series-III run-time environment, Fortran-86 provides the following default preconnections:

Unit	Device
5	console input
6	console output
other	system work file

The system specific appendix provides examples for overriding the default preconnections. You can specify the UNIT load-time control at execution time. The format of the UNIT control is as follows:

```
source (UNITn = path)
```

where

source	is the name of your relocated object code.
n	is a number between 0 and 255.
path	is an operating environment filename.

Note the following examples:

```
PROGRM (UNIT4=:LP:) <cr>
PROGRM.LOC (UNIT1=:CI:,UNIT0=:CO:)<cr>
```

The preconnection feature applies to Fortran-86 programs that have been compiled, linked, and optionally located to run in your system.

When preconnecting a file, the string UNITn may not contain spaces; i.e., UNIT7, not UNIT 7.

14.6 Executing Programs

Your linked (and relocated) program can now be loaded and executed. Your program file could also be used as input to the DEBUG-86 debugger.

To run correctly, a program must be complete; i.e., it must contain all the modules necessary to run. A program must contain modules from the run-time support library described in Section 14.3.2. The system specific appendix provides examples of program execution.

CHAPTER 15
ERRORS AND WARNINGS

This chapter lists all the compiler and run-time error and warning messages. The compiler makes a distinction between errors and warnings, since the latter produces executable object code despite the diagnostic messages.

Operating system error messages can be found in the manuals listed in the system specific appendix. LINK86, LOC86, LIB86, and OH86 error messages can be found in the *iAPX 86,88 Family Utilities User's Guide for 8086-Based Development Systems.*

15.1 Compiler Controls and the Error Listing

The compiler errors and warnings appear in the error message listing on the device specified by the PRINT compiler control. If the NOPRINT control is active, the compiler does not generate an error message listing. Specifying the LIST control causes the compiler to produce a complete listing of the program code, including statements associated with error messages. Using the NOLIST control, however, causes the compiler to list only those statements where errors were detected.

Source program errors are usually not fatal. An error in your source code will be logged in the error message listing and the compiler will continue to process your source file, if possible. You can request that the compiler halt upon encountering one or more errors using the ERRORLIMIT compiler control. See Section 11.4.5 for specifics about this option.

15.2 Compiler Error Messages

The Fortran-86 compiler can issue the following types of error messages:

- Fortran-86 source program errors
- Compiler control errors
- Input/Output errors
- Insufficient memory errors
- Compiler failure errors

15.2.1 Error Format

A more detailed description of the error message listing format can be found in Section 13.1.5 of the Compiler Output chapter. Errors and warnings within Fortran-86 source code are printed in this listing in this format:

`STATEMENT` *n*`[, NEAR` *symbol*`],` *errortype m*`:` *message*

where

errortype	is either error or a warning.
m	is the specific error or warning number.
n	is the internal number of the statement containing the error.
symbol	is a pointer to the location of the error within the statement.
message	is the actual error message.

The compiler summarizes source program error totals at the end of program listing for each program unit, as described in Section 13.1.6.

15.2.2 Error Messages

The following lists the compiler error messages. Each line gives the number and message for each error. If any message appears without a number, call your Intel representative; this indicates a compiler failure.

F001 *‹element›* NEEDED NEAR *‹source text›*

‹element› is required to complete a valid Fortran-86 statement or control. The rest of the statement is not compiled.

F002 INCORRECTLY PLACED PRIMARY CONTROL

F003 UNIMPLEMENTED GENERAL CONTROL

F004 UNIMPLEMENTED PRIMARY CONTROL

F005 INITIAL CONTROL CANNOT BE CHANGED

F006 PARSING TERMINATED BEFORE END OF STATEMENT

F007 UNSUPPORTED STATEMENT

F008 DUPLICATE LABEL

F009 STATEMENT ILLEGAL FOR BLOCK DATA

F010 STATEMENT OUT OF ORDER

F011 NAME ALREADY IN COMMON

F012 NAME CANNOT BE IN COMMON

F013 ARRAY NAME MUST HAVE DIMENSIONS

F014 ONLY DUMMY ARGUMENTS CAN HAVE VARIABLE DIMENSIONS

F015 NAME CANNOT BE AN ARRAY

F016 DUPLICATE DIMENSION SPECIFICATION

F017 NUMBER OF DIMENSIONS EXCEEDS SEVEN

F018 ONLY LAST DIMENSION CAN BE STAR

F019 LOWER BOUND CANNOT BE STAR

F020 NAME CANNOT BE INITIALIZED

F021 ILLEGAL NAME IN DATA EXPRESSION

F022 CONSTANT IN DATA EXPRESSION MUST BE INTEGER

F023 NAME IN CONSTANT LIST IS NOT A CONSTANT

F024	NAME ILLEGAL FOR MEMORY ASSOCIATION
F025	CONSTANT EXPRESSIONS OF THIS DATA TYPE ARE NOT SUPPORTED
F026	NAME CANNOT BE A SYMBOLIC CONSTANT
F027	DUPLICATE DEFINITION OF SYMBOLIC CONSTANT
F028	RIGHT SIDE OF CONSTANT EXPRESSION IS NOT CONSTANT
F029	DUPLICATE DEFINITION OF EXTERNAL PROCEDURE
F030	NAME CANNOT BE AN EXTERNAL PROCEDURE
F031	DUPLICATE DEFINITION OF INTRINSIC PROCEDURE
F032	NAME CANNOT BE AN INTRINSIC PROCEDURE
F033	UNSUPPORTED STATEMENT
F034	ALTERNATE RETURN NOT SUPPORTED
F035	NAME IS ALREADY A DUMMY ARGUMENT
F036	NAME CANNOT BE A DUMMY ARGUMENT
F037	LABEL MISSING ON FORMAT STATEMENT
F038	STATEMENT-FUNCTION DUMMY ARGUMENT MUST BE A NAME
F039	NAME ILLEGAL AS STATEMENT-FUNCTION ARGUMENT
F040	DUPLICATE DUMMY ARGUMENT OF STATEMENT FUNCTION
F041	STATEMENT-FUNCTION DUMMY ARGUMENT CANNOT BE SUBSCRIPTED
F042	ILLEGAL ASSIGNMENT TARGET
F043	ILLEGAL USE OF NAME AS A FUNCTION
F044	ILLEGAL USE OF NAME AS A SUBROUTINE
F045	LENGTH EXPRESSION IS NOT AN INTEGER CONSTANT
F046	EXPRESSION IS NOT CONSTANT
F047	EXPRESSION IS NOT OF TYPE INTEGER
F048	ILLEGAL OPERATOR OR CONSTRUCT
F049	ALTERNATE RETURN IS NOT SUPPORTED
F050	MISSING TERMINATION FOR A DO OR BLOCK IF

F051	MISSING TERMINATION FOR A CONTAINED DO
F052	MISSING ENDIF FOR A CONTAINED BLOCK IF
F054	ILLEGAL STATEMENT FOLLOWING LOGICAL IF
F055	NO MATCHING BLOCK IF
F056	ELSE OR ELSEIF FOLLOWING ELSE IS ILLEGAL
F057	DO VARIABLE IS NOT AN INTEGER VARIABLE
F058	END SPECIFIER ILLEGAL WITH WRITE OR DIRECT ACCESS
F059	UNFORMATTED INTERNAL IO NOT ALLOWED
F060	DIRECT ACCESS NOT ALLOWED FOR INTERNAL OR LIST IO
F061	MISSING UNIT SPECIFIER
F062	MULTIPLE UNIT SPECIFIERS
F063	MULTIPLE FILE SPECIFIERS
F064	MULTIPLE RECORD SPECIFIERS
F065	MULTIPLE IOSTAT SPECIFIERS
F066	MULTIPLE ERR SPECIFIERS
F067	MULTIPLE END SPECIFIERS
F068	INVALID STATUS SPECIFIER
F069	MULTIPLE FILE SPECIFIERS
F070	MULTIPLE RECL SPECIFIERS
F071	INVALID STATUS SPECIFIER
F072	MULTIPLE STATUS SPECIFIERS
F073	INVALID ACCESS SPECIFIER
F074	MULTIPLE ACCESS SPECIFIERS
F075	INVALID FORM SPECIFIER
F076	MULTIPLE FORM SPECIFIERS
F077	INVALID BLANK SPECIFIER
F078	MULTIPLE BLANK SPECIFIERS
F079	INVALID CARRIAGE SPECIFIER
F080	MULTIPLE CARRIAGE SPECIFIERS

F081	WRONG NUMBER OF ARGUMENTS
F082	ARGUMENT MUST BE AN EXTERNAL PROCEDURE
F083	TWO-BYTE RESULT FIELD NEEDED
F084	DIMENSION VARIABLE NOT AN ARGUMENT OR IN COMMON
F085	FORMAT SPECIFIER IS NOT A FORMAT LABEL
F086	FOUR-BYTE FIELD LENGTH REQUIRED
F087	TWO-BYTE FIELD LENGTH REQUIRED
F088	DATA TYPE INTEGER OR LOGICAL REQUIRED
F089	VARIABLE REFERENCE REQUIRED
F090	DATA TYPE INTEGER REQUIRED
F091	ARITHMETIC EXPRESSION REQUIRED
F092	DATA TYPE LOGICAL REQUIRED
F093	CHARACTER DATA TYPE REQUIRED
F094	ILLEGAL USE OF PROCEDURE NAME
F095	SUBSCRIPTS MISSING IN ARRAY REFERENCE
F096	SUBSTRING ALLOWED WITH TYPE CHARACTER ONLY
F097	INCORRECT NUMBER OF SUBSCRIPTS
F098	THIS PROCEDURE CANNOT BE USED AS AN ACTUAL ARGUMENT
F099	WRONG NUMBER OF ARGUMENTS
F100	INCOMPATIBLE DATA TYPE
F101	ARRAY SIZE IS UNKNOWN
F102	RIGHT SIDE IS NOT A CHARACTER EXPRESSION
F103	VARIABLE-LENGTH FUNCTION NOT ALLOWED
F104	VARIABLE-LENGTH CHARACTER STRING NOT ALLOWED
F105	LABEL IS NOT DEFINED AT AN EXECUTABLE STATEMENT
F106	SUBSTRING FOR NONCHARACTER VARIABLE IS IGNORED
F107	ILLEGAL SUBSTRING START IS ASSUMED

F108 SUBSTRING END CANNOT BE LESS THAN SUBSTRING START

F109 SUBSCRIPTS FOR NON-ARRAY IGNORED

F110 WRONG NUMBER OF SUBSCRIPTS-FIRST ELEMENT ASSUMED

F111 SUBSCRIPT VALUE IS LESS THAN LOW BOUND

F112 SUBSCRIPT VALUE EXCEEDS UPPER BOUND

F113 VARIABLE DIMENSION NOT ALLOWED

F114 DUMMY ARGUMENTS ILLEGAL IN INTERRUPT PROCEDURE

F115 ONLY SUBROUTINES AND FUNCTIONS CAN BE REENTRANT

F116 INTERRUPT PROCEDURE MUST BE A SUBROUTINE

F118 BUILTIN OPERAND MUST BE IN CONTIGUOUS STORAGE

The 8087 built-in functions STSW87, LDCW87, STCW87, SAV87, and RST87, do not accept operands in noncontiguous storage. This is because they translate directly into their corresponding 8087 machine instructions. See error F206 for a description of how to avoid this problem.

F119 THIS STATEMENT IS TOO COMPLEX

F120 ONLY INTEGER AND LOGICAL SUPPORTED FOR VALUE ARGUMENTS

F121 UPPER BOUND IS LESS THAN LOWER BOUND

F122 DUPLICATE TYPE SPECIFICATION IGNORED

F123 ILLEGAL USE OF HOLLERITH

F124 ARRAY SIZE EXCEEDS COMPILER CAPACITY

F125 INVALID COMPONENT DATA TYPE

F126 COMPLEX ORDERED COMPARISON ILLEGAL

F127 DUPLICATE PROCEDURE NAME IN INTERFACE CONTROL

F151 END STATEMENT MISSING

F152 END STATEMENT IN INCLUDE IGNORED

F153 MAXIMUM PAGELENGTH IS 255

F154 MINIMUM PAGELENGTH IS 5

F155 MAXIMUM PAGEWIDTH IS 132

F156 MINIMUM PAGEWIDTH IS 60

F157	MORE DATA VARIABLES THAN DATA CONSTANTS
F158	THIS STATEMENT IS TOO COMPLEX
F159	ATTEMPT TO DIVIDE BY 0
F160	OVERFLOW IN CONSTANT DIVISION
F161	THE LINE AT OR AFTER THIS STATEMENT IS TOO LONG
F162	TOO MANY CHARACTERS IN STATEMENT
F163	NONDIGIT IN STATEMENT-LABEL FIELD
F164	FIRST LINE OF A STATEMENT IS A CONTINUATION LINE
F165	MORE THAN 19 CONTINUATION LINES NOT SUPPORTED
F166	LABEL PRESENT ON CONTINUATION LINE--LINE IGNORED
F167	INVALID CHARACTER(S) IN SOURCE AT OR AFTER THIS STATEMENT
F168	CONTROL NEAR 'XXX' CANNOT BE NEGATED
F169	HOLLERITH STRING LONGER THAN 255--TRUNCATED ON RIGHT
F170	ZERO-LENGTH HOLLERITH STRING ILLEGAL
F171	STATEMENT ENDS BEFORE END OF HOLLERITH STRING
F172	TOO MANY NESTED INCLUDE LEVELS
F173	INTERRUPT NUMBER MUST BE BETWEEN 0 AND 255--LOW BYTE USED
F174	UNKNOWN CONTROL IN SOURCE PROGRAM NEAR
F175	THE BLOCK CONTAINING THIS STATEMENT IS TOO COMPLEX
F176	INTRINSIC HAS INCORRECT NUMBER OF OPERANDS
F177	TYPES OF OPERANDS INCOMPATIBLE
F178	TYPE OF ARGUMENT INCOMPATIBLE WITH INTRINSIC
F179	MORE DATA CONSTANTS THAN DATA VARIABLES
F180	THIS STATEMENT IS TOO COMPLEX
F181	ASSIGN VARIABLE MUST BE AT LEAST TWO BYTES LONG
F182	TYPES OF OPERANDS INCOMPATIBLE WITH OPERATION

```
F183  PARAMETER TYPE MISMATCH WITH EARLIER
      INVOCATION OF PROCEDURE

F191  SOURCE FILENAME MISSING

F192  UNKNOWN CONTROL IN COMMAND TAIL

F194  INVALID NUMERIC DIGIT

F201  TOO MANY NAMES TO SORT

F202  EQUIVALENCE OF TWO ITEMS IN DIFFERENT COMMONS
      IGNORED

F203  EQUIVALENCE OF AN ITEM AT TWO DIFFERENT
      LOCATIONS IGNORED

F204  ATTEMPT TO EXTEND COMMON ON LEFT BY
      EQUIVALENCE IGNORED

F205  EQUIVALENCE LIST WITH FEWER THAN TWO LEGAL
      ELEMENTS IGNORED

F206  VARIABLE OR ARRAY ELEMENT ALLOCATED
      NONCONTIGUOUS STORAGE
```

A variable or array element in the program overlaps a 64K-byte segment boundary and therefore must be accessed by an out-of-line run-time procedure when referenced in an executable statement.

This message is not an error or a program restriction. It is issued as an aid to users who want to optimize their program performance.

Special out-of-line handling can be avoided by redefining certain memory sequences so that no single variable or array element overlaps 64K-segment boundaries. To do this:

1. Do not mix data lengths in a numeric/logical COMMON block that exceeds 65,520 (or 64K—16) bytes in length.
2. When a local CHARACTER array exceeds 64K bytes in length, the element length should divide 65,520 evenly. ($65{,}520 = 2^4 \times 3^2 \times 5 \times 7 \times 13$, so an element length of any combination of these factors will avoid this warning.)
3. If mixed data lengths, including odd-length CHARACTER types, are necessary in a COMMON block that exceeds 65,520 bytes in length, reorder the elements or add filler variables so that the 65,521st (131,041st, etc.) byte coincides with the first byte of a variable or array element.
4. If the overlapping variable is the result of a mixed-length EQUIVALENCE specification, change your program to avoid the need for the mixed-length EQUIVALENCE specification.

```
F207  OVERLAP ACTUAL ARGUMENT--SPECIAL COMPILATION
      REQUIRED.
```

A variable or array element that has been allocated noncontiguous storage (or an array containing such an element) has been used as an actual argument for a subroutine or function. Since variables in noncontiguous storage require special handling by the compiler, the subroutine or function indicated must be compiled using the OVERLAP control.

As an alternative to the OVERLAP control, you may redefine the calling program's actual arguments so that no single variable or array element overlaps 64K-segment boundaries. For more information, see the explanation of the compiler warning message F206.

```
F208   CONSTANT/VARIABLE TYPE MISMATCH IN A DATA
       STATEMENT--ENTRIES IGNORED

F209   NONBLANK CHARACTERS FOLLOWING FORMAT
       SPECIFICATIONS IGNORED

F210   FORMAT DOES NOT BEGIN WITH '('

F211   INVALID OR MISSING DELIMITER--',', '/', ':',
       OR ')' NEEDED

F212   UNRECOGNIZABLE FORMAT EDIT DESCRIPTOR FOUND

F213   '-' NOT FOLLOWED BY AN INTEGER

F214   A NEGATIVE INTEGER IS ALLOWED ONLY WITH 'P'

F215   'B' REQUIRES A NONZERO POSITIVE INTEGER WIDTH

F216   'I' REQUIRES A NONZERO POSITIVE INTEGER WIDTH

F217   'I' REQUIRES A NONNEGATIVE INTEGER AFTER THE
       '.'

F218   'Z' REQUIRES A NONZERO POSITIVE INTEGER WIDTH

F219   'L' REQUIRES A NONZERO POSITIVE INTEGER WIDTH

F220   'F' REQUIRES A NONZERO POSITIVE INTEGER WIDTH

F221   'F' REQUIRES A '.' AFTER ITS WIDTH

F222   'F' REQUIRES A NONNEGATIVE INTEGER AFTER THE
       '.'

F223   'D' REQUIRES A '.' AFTER ITS WIDTH

F225   'D' REQUIRES A NONNEGATIVE INTEGER AFTER THE
       '.'

F226   'E' REQUIRES A NONZERO POSITIVE INTEGER WIDTH

F227   'E' REQUIRES A '.' AFTER ITS WIDTH

F228   'E' REQUIRES A NONNEGATIVE INTEGER AFTER THE
       '.'

F229   'E' REQUIRES A NONZERO POSITIVE INTEGER
       EXPONENT 'E' FIELD

F230   'G' REQUIRES A NONZERO POSITIVE INTEGER WIDTH

F231   'G' REQUIRES A '.' AFTER ITS WIDTH
```

F232	'G' REQUIRES A NONNEGATIVE INTEGER AFTER THE '.'
F233	'G' REQUIRES A NONZERO POSITIVE INTEGER EXPONENT 'E' FIELD
F234	A SIGNED INTEGER CONSTANT MUST PRECEDE 'P'
F235	A NONZERO POSITIVE INTEGER CONSTANT MUST PRECEDE 'X'
F236	A NONZERO POSITIVE INTEGER CONSTANT MUST PRECEDE 'H'
F237	CLOSING QUOTE MISSING FOR QUOTED STRING
F238	'H' FORMAT SPECIFIES MORE CHARACTERS THAN ARE AVAILABLE
F239	DECIMAL PART LARGER THAN DESCRIPTOR FIELD WIDTH
F240	ILLEGAL OR UNPRINTABLE FORMAT DESCRIPTOR FOUND
F241	REPEAT NESTING EXCEEDS 3 LEVELS
F242	ILLEGAL CHARACTER IN A QUOTED STRING
F243	'P' FORMAT IS OUT OF RANGE
F244	MORE LEFT PARENTHESES THAN RIGHT
F245	INTEGER SPECIFIED IS OUT OF RANGE ALLOWED IN FORMAT STATEMENTS
F246	THE DECIMAL PART OF AN 'I' IS GREATER THAN ITS WIDTH
F247	DECIMAL AND EXPONENT PARTS LARGER THAN DESCRIPTOR FIELD WIDTH
F251	TOO MANY PROCEDURE NAMES AND LABELS TO SORT
F252	MORE THAN 64K OF DATA OUTSIDE COMMON--64K USED
F253	CODE CROSSES 64K BOUNDARY AT OR AFTER THIS STATEMENT
F254	MORE THAN 64K OF CODE INCL. CONSTANTS--64K USED
F255	MORE THAN 64K OF STACK NEEDED--64K USED
F256	MORE THAN 64K OF PARAMETERS--64K USED
F257	ILLEGAL USE OF DATA STATEMENT IGNORED

F258	DATA CONSTANT EXCEEDS 255 BYTES
F259	TOO MANY ARGUMENTS FOR TYPE CHECKING
F261	TOO MANY ERRORS TO SORT
F271	CONSTANT CONTAINS AN ILLEGAL CHARACTER--BLANK ASSUMED
F272	INTEGER CONSTANT WON'T FIT IN FOUR BYTES--TRUNCATED ON LEFT
F273	LABEL IS GREATER THAN 99999--RIGHTMOST DIGITS TRUNCATED
F274	NULL STRING IS ILLEGAL--' ' ASSUMED
F275	QUOTED STRING LONGER THAN 255 CHARACTERS--TRUNCATED ON RIGHT
F276	LABEL OF ZERO IS ILLEGAL
F277	NAME LONGER THAN 6 CHARACTERS--TRUNCATED ON RIGHT
F278	DIGIT STRING OF MORE THAN FIVE DIGITS IS ILLEGAL
F279	INCOMPATIBLE LENGTHS FOR SYMBOLIC AND ACTUAL CONSTANT
F280	IMPLICIT RANGE INVALID--ONLY FIRST LETTER USED
F281	INCOMPATIBLE DATA TYPE AND LENGTH
F282	LETTER ALREADY GIVEN AN IMPLICIT TYPE
F283	SUBPROGRAM NAME IS ALREADY A SUBPROGRAM NAME
F284	SUBPROGRAM NAME IS ALREADY A COMMON NAME
F285	COMMON NAME IS ALREADY A SUBPROGRAM NAME
F286	LENGTH CANNOT BE STAR
F287	EXPLICIT LENGTH ILLEGAL--DEFAULT USED
F288	NAME CANNOT BE CHARACTER* (*)--CHARACTER*1 USED
F289	THE TYPE OF THIS INTRINSIC FUNCTION IS CHANGED TO ITS DEFAULT
F290	LENGTH SPECIFICATION EXCEEDS 64K--DEFAULT USED
F291	OVER 49 OVERFLOW SEGMENTS ALLOCATED

15.2.3 Compiler Control Error Messages

If the Fortran-86 compiler detects an error in a compiler control (whether in a control line embedded in source code or in the compiler invocation line), the compilation may be halted. If this happens, the compiler issues an error message to both the console and the list file. The form of the message is:

```
***FORTRAN COMPILATION TERMINATED <MESSAGE NUMBER>
```

15.2.4 Compiler Failure Error Messages

Compiler failure messages are indicated by unnumbered error messages.

The following unnumbered compiler error message—

```
***FORTRAN COMPILATION TERMINATED: MAXIMUM VIRTUAL
SYMBOL TABLE SPACE EXCEEDED IN PHASE a
```

indicates that the symbol table space required for the compilation exceeds the maximum size of 256K bytes. This can occur when a source file contains multiple modules. To avoid this error, break up the source file and compile the modules separately.

Fatal compiler failure errors are internal Fortran-86 compiler errors that should never occur. If you encounter one of these errors, please report it to Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051, Attention: Software Support Services.

15.3 Run-Time Errors

Certain Series-III operating system errors may occur that are documented in the *Intellec® Series-III Microcomputer Development System Console Operating Instructions.* Run-time errors that are unique to the Fortran-86 run-time support software are described in this section.

A masked floating-point run-time error can occur without stopping the program. When a run-time error other than a masked floating-point error occurs, the system stops running the program, prints a run-time exception message, and returns control to the operating system.

Run-time system exception messages take the following form:

```
*** RUN-TIME type EXCEPTION code
*** NEAR LOCATION hhhhH:hhhhH
*** JOB ABORTED.
```

The *type* of the run-time exception can be one of the following types:

FORTRAN I/O
I/O
OPERATING ENVIRONMENT
INTEGER ZERO DIVIDE
INTEGER OVERFLOW
RANGE
CHECK

For each type, the *code* is the hexadecimal exception code number for each message. The hexadecimal locations *hhhh*H:*hhhh*H are the values in CS:IP after control returns from the run-time system to the program. Each message is described in the subsequent sections by type and by code number.

15.3.1 Input/Output Exceptions

If a Fortran-86 I/O statement includes the ERR specifier in its control list, the compiler transfers control to the statement designated by ERR when an error is detected. The default error handler is not called in this case.

If you include the IOSTAT specifier in the control list of a Fortran-86 I/O statement, I/O operations return a numerical code as well as the value of a symbol designated by IOSTAT.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1200H
```

An invalid link sequence was specified for the run-time libraries.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1201H
```

Negative system error detected.

```
RUN-TIME FORTRAN I/O EXCEPTION: 120EH
```

Output list specifies more values than can fit into a direct access record.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1210H
```

An initial left parenthesis is required to define a format statement.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1211H
```

Invalid delimiter was found in a FORMAT statement (expecting ",","/", or ")".

```
RUN-TIME FORTRAN I/O EXCEPTION: 1212H
```

An unrecognizable edit descriptor was found in a FORMAT statement.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1213H
```

A nondigit followed a + or a dash (—) in the FORMAT statement (note that "-P" must "-1P").

```
RUN-TIME FORTRAN I/O EXCEPTION: 1214H
```

Only P-format descriptor can follow a negative integer.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1215H
```

B-format descriptor must be followed by a positive integer width field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1216H
```

I-format descriptor must be followed by a positive integer width field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1217H
```
Iw.m-format descriptor must have a positive integer following the decimal point.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1218H
```
Z-format descriptor must be followed by a positive integer width field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1219H
```
L-format descriptor must be followed by a positive integer width field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 121AH
```
F-format descriptor must be followed by a positive integer width field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 121BH
```
F-format descriptor must have a decimal point following the width field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 121CH
```
F-format descriptor must have a nonnegative integer following the decimal point.

```
RUN-TIME FORTRAN I/O EXCEPTION: 121DH
```
D-format descriptor must be followed by a positive integer width field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 121EH
```
D-format descriptor must have a decimal point following the width field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 121FH
```
D-format descriptor must have a nonnegative integer following the decimal point.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1220H
```
E-format descriptor must be followed by a positive integer width field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1221H
```
E-format descriptor must have a decimal point following the width field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1222H
```
E-format descriptor must have a nonnegative integer following the decimal point.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1223H
```
E-format descriptor must have a positive integer following the E in the exponent field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1224H
```
G-format descriptor must be followed by a positive integer width field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1225H
```
G-format descriptor must have a decimal point following the width field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1226H
```
G-format descriptor must have a nonnegative integer following the decimal point.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1227H
```
G-format descriptor must have a positive integer following the E in the exponent field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1228H
```
A signed integer constant must precede P-format descriptor.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1229H
```
A positive integer constant must precede X-format descriptor.

```
RUN-TIME FORTRAN I/O EXCEPTION: 122AH
```
A positive integer constant must precede H-format descriptor.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1231H
```
The closing quote for a quoted string is missing.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1232H
```
H-format descriptor requires more characters than are available.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1233H
```
The width field must be greater than or equal to the decimal field of a floating-point edit descriptor (E, G, D).

```
RUN-TIME FORTRAN I/O EXCEPTION: 1234H
```
A character in the FORMAT statement was found to be outside the set of characters allowed for format edit descriptors.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1235H
```
The nesting of brackets in a FORMAT statement exceeds limit (3).

```
RUN-TIME FORTRAN I/O EXCEPTION: 1236H
```
An illegal character was found within a quoted string.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1237H
```
The integer specified for P-format descriptor was out of range (−2**15,2**15−1).

```
RUN-TIME FORTRAN I/O EXCEPTION: 1239H
```
Integer specified is out of range allowed by FORMAT statements.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1238H
```
More left parentheses than right.

```
RUN-TIME FORTRAN I/O EXCEPTION: 123AH
```
Integer size greater than field width.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1240H
```
H-format descriptor not allowed on input.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1241H
```
A logical data item was expected on input.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1242H
```
An integer data item was expected on input.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1243H
```
A floating-point data item was expected on input.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1244H
```
An invalid logical data field was found on input.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1248H
```
An invalid hexadecimal data field was found on input.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1249H
```
An invalid binary data field was found on input.

```
RUN-TIME FORTRAN I/O EXCEPTION: 124AH
```
A repeatable edit descriptor is missing, causing an infinite loop to occur in the processing of a repeated FORMAT statement.

```
RUN-TIME FORTRAN I/O EXCEPTION: 124BH
```
The scale of an input exponent is out of range.

```
RUN-TIME FORTRAN I/O EXCEPTION: 124CH
```
Quoted string input is invalid.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1251H
```
End of file record was encountered with no END = specified.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1252H
```
An attempt was made to read or write beyond end of record.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1254H
```
The data transfer mode is inconsistent with the file's FORM attribute.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1255H
```
The data transfer mode is inconsistent with the file's ACCESS attribute.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1256H
```
Syntax error in formatted binary or hexadecimal input field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1260H
```
Invalid delimiter in list directed input field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1261H
```
Syntax error in list directed alphanumeric input field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1262H
```
Syntax error in formatted/list directed logical input field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1263H
```
Syntax error in formatted/list directed floating-point input field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1264H
```
Syntax error in formatted/list directed integer input field.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1265H
```
Zero-valued repeat factor not allowed in list-directed input.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1270H
```
An attempt was made to append to an internal file.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1272H
```
The input data transfer conflicts with CARRIAGE = specifier.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1273H
```
The next I/O list element and repeatable edit descriptor do not match.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1274H
```
Invalid repeat specifier in FORMAT statement.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1275H
```
Expected repeatable edit descriptor is missing.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1276H
```

Recursion error. Attempt was made to perform I/O on a file which is active on the same unit.

```
RUN-TIME FORTRAN I/O EXCEPTION: 1282H
```

Attempt to read or write past the ENDFILE record.

```
RUN-TIME FORTRAN I/O EXCEPTION 1283H
```

The integer field on input does not conform to the decimal signed integer syntax.

```
RUN-TIME FORTRAN I/O EXCEPTION 1284H
```

The floating point field on input does not conform to the run-time signed number syntax.

```
RUN-TIME FORTRAN I/O EXCEPTION 1285H
```

The integer field on formatted input defined a signed integer which could not fit into the INTEGER*2 range.

```
RUN-TIME FORTRAN I/O EXCEPTION 1286H
```

The integer field formatted input defined a signed integer which could not fit into the INTEGER*2 range.

```
RUN-TIME FORTRAN I/O EXCEPTION 1287H
```

The floating point field on formatted input defined a signed number whose magnitude was too large to fit into the tempreal range.

```
RUN-TIME FORTRAN I/O EXCEPTION 1288H
```

The floating point field on formatted input defined a signed number whose magnitude was too small to fit into the tempreal range.

```
RUN-TIME FORTRAN I/O EXCEPTION 1289H
```

The integer field on formatted input defined a signed integer which could not fit into the INTEGER*1 range.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12A1H
```

The string passed in the STATUS specifier of an OPEN statement is illegal.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12A2H
```

The string passed in the ACCESS specifier of an OPEN statement is illegal.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12A3H
```

The string passed in the FORM specifier of an OPEN statement is illegal.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12A4H
```

The string passed in the BLANK specifier of an OPEN statement is illegal.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12A5H
```
The string passed in the CARRIAGE specifier of an OPEN statement is illegal.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12A6H
```
A FILE= specifier must be given in the OPEN statement when STATUS='NEW'.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12A7H
```
A FILE= specifier must be given in the OPEN statement when STATUS='OLD'.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12A8H
```
A FILE/ specifier must not be given in the OPEN statement when STATUS/ 'SCRATCH'.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12A9H
```
Of those attributes specified in the OPEN statement, only BLANK=, CARRIAGE=, and/or RECL= can change for an existing file-unit connection.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12AAH
```
The integer value specified for RECL= in the OPEN statement must be positive.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12ABH
```
RECL= must not be specified in the OPEN statement when ACCESS='SEQUENTIAL' and FORM='UNFORMATTED'.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12ACH
```
RECL= must be specified in the OPEN statement when ACCESS='DIRECT'.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12ADH
```
RECL= attribute of an existing connection must not be changed in the OPEN statement unless ACCESS='SEQUENTIAL' and FORM='FORMATTED'.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12AEH
```
BLANK= must not be specified in the OPEN statement for a new connection when FORM='UNFORMATTED'.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12AFH
```
BLANK= must not be specified in the OPEN statement for an existing connection when FORM='UNFORMATTED'.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12B0H
```
CARRIAGE= must not be specified in the OPEN statement for a new connection when FORM='UNFORMATTED'.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12B1H
```
CARRIAGE= must not be specified in the OPEN statement for an existing connection when FORM='UNFORMATTED'.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12B2H
```
The file-unit does not exist.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12C1H
```
KEEP must not be specified for a file whose status prior to execution of the CLOSE statement is SCRATCH.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12C2H
```
The string passed in the STATUS specifier of a CLOSE statement is illegal.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12D1H
```
The external unit specified by a BACKSPACE statement was not connected.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12D2H
```
The external unit specified by a BACKSPACE statement was not connected for sequential access.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12D3H
```
Backspacing over records written using list-directed formatting is illegal.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12E1H
```
The external unit specified by a REWIND statement was not connected.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12E2H
```
The external unit specified by a REWIND statement was not connected for sequential access.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12F1H
```
The external unit specified by an ENDFILE statement was not connected.

```
RUN-TIME FORTRAN I/O EXCEPTION: 12F2H
```
The external unit specified by an ENDFILE statement was not connected for sequential access.

```
RUN-TIME I/O EXCEPTION: 9102H
```
The end of file was encountered when illegal.

```
RUN-TIME I/O EXCEPTION: 9103H
```
The integer field on input does not conform to the decimal signed integer syntax.

```
RUN-TIME I/O EXCEPTION: 9104H
```
The floating-point field on input does not conform to the run-time signed number syntax.

```
RUN-TIME I/O EXCEPTION: 9105H
```

The integer field on formatted input defined a signed integer which could not fit into the INTEGER 2 range.

```
RUN-TIME I/O EXCEPTION: 9106H
```

The integer field on formatted input defined a signed integer which could not fit into the INTEGER 4 range.

```
RUN-TIME I/O EXCEPTION: 9107H
```

The floating-point field on formatted input defined a signed number whose magnitude was too large to fit into the TEMPREAL range.

```
RUN-TIME I/O EXCEPTION: 9108H
```

The floating-point field on formatted input defined a signed number whose magnitude was too small to fit into the TEMPREAL range.

```
RUN-TIME I/O EXCEPTION: 9109H
```

The integer field on formatted input defined a signed integer which could not fit into INTEGER 1 range.

15.3.2 Operating Environment Error

```
RUN-TIME EXCEPTION: 1500H
```

Configuration exception. Call your local Intel representative.

```
RUN-TIME EXCEPTION: 1501H
```

Command line preconnection facility has detected invalid preconnection syntax.

```
RUN-TIME EXCEPTION: 1502H
```

An attempt was made to open a file which should have not already existed.

```
RUN-TIME EXCEPTION: 1503H
```

Configuration error. File not open for write access.

```
RUN-TIME EXCEPTION: 1504H
```

Configuration error. File not open for read access.

```
RUN-TIME EXCEPTION: 1505H
```

More than six file's descriptors were requested from the RTNULL descriptor allocator.

```
RUN-TIME EXCEPTION: 1506H
```

Unformatted sequential record is inconsistent.

```
RUN-TIME EXCEPTION: 1507H
```

Seek out or range. Attempt to seek when offset (i.e., rec_len * rec_num > (2**31)-1).

```
RUN-TIME EXCEPTION: 1508H
```

DIRECT record length too large (maximum allowable: formatted, 65,503, unformatted, 65,503).

15.3.3 Integer Exceptions

```
RUN-TIME INTEGER EXCEPTION: 8000H
```

8-bit, 16-bit, or 32-bit signed integer zero divide.

```
RUN-TIME INTEGER EXCEPTION: 8001H
```

8-bit, 16-bit, or 32-bit signed integer overflow.

15.3.4 Range and Check Exceptions

```
RUN-TIME EXCEPTION: 8017H
```

Compiler generated check exception (e.g., stack overflow).

15.3.5 Floating-Point Function Exceptions

Floating-point function error messages take the following form:

```
*** RUN-TIME FLOATING-POINT function EXCEPTION status
*** NEAR LOCATION hhhhhH
*** JOB ABORTED
```

The *function* can be one of the following:

SIN	ANINT	RMD
COS	NINT	SIGN
TAN	RINT	*y**x*
ASIN	IRINT	*y**i*
ACOS	SINH	MIN
ATAN	COSH	MAX
ATAN2	TANH	CDPRJ
ALOG	SQRT	CPRJ
AL0G10	DIM	CDCMPLX
INT	EXP	CMPLX
AINT	MOD	

The *status* is the hexadecimal value of the 8087 STATUS register and the location *hhhhh* is the 20-bit physical address of the location of the exception. The 8087 STATUS values are described in the *8086 Family User's Manual Numerics Supplement*. General floating-point exceptions are discussed in the next section.

15.3.6 Floating-Point 8087 Exceptions

Floating-point error messages take the following form:

```
*** RUN-TIME 8087 EXCEPTION status
*** INSTR OPCODE op
*** MEMOP ADDRESS hhhhhH
*** NEAR LOCATION hhhhhH
*** JOB ABORTED.
```

The *status* is the hexadecimal value in the 8087 STATUS register. The *op* is the hexadecimal value of the 8087 instruction opcode register. The *hhhhh*H is a hexadecimal 20-bit physical address. The 8087 registers are described in the *8086 Family User's Manual Numerics Supplement.*

There are six possible 8087 floating-point, or exception conditions: invalid operation, denormalized operand, zero divide, overflow, underflow, and precision. Not all exceptions are errors.

This section first discusses the meaning of the six types of exceptions, what conditions cause them, and the actions performed when each exception occurs with the corresponding exception controls unmasked. The *8086 Family User's Manual Numerics Supplement* discusses the unmasked case.

Section 7.6 contains explanations of rounding, denormalized and unnormalized numbers, unnormalized arithmetic, infinity arithmetic, and NaNs. These discussions should suffice for Fortran-86 users; however, if you are also writing modules in other languages to interface with the 8087 chip or emulator, you may wish to see the *8086 Family User's Manual Numerics Supplement* for a fuller explanation of some topics.

NOTE

Fortran-86 presets the 8087 computation modes and exception masks (explained in the *8086 Family User's Manual Numerics Supplement*) to the following recommended settings:

- The infinity arithmetic mode is projective.
- The rounding mode is round-to-nearest.
- The precision mode for intermediate results is 64 bits of precision.
- The denormal arithmetic mode is warning mode.
- All 8087 exception conditions are masked except invalid operation, which is unmasked.
- The 8087 interrupt enable mask bit is zero (interrupt enabled).

You can change the computation modes and exception masks in a Fortran-86 program by using the 8087 control intrinsics (see Section 6.1.2.3). The following discussions assume that you have not changed any of these settings. If you use any of the functions SIN, COS, TAN, ASIN, ACOS, ATAN, ATAN2, EXP, ALOG, ALOG10, SINH, COSH, TANH, *y****x*, *y****i*, NINT, ANINT, MOD, or RMD you must not unmask the precision error and the precision exception bit in the 8087 STATUS word is undefined after the operation is completed.

15.3.6.1 Invalid Operation

An *invalid operation* exception occurs when either an operand is invalid for the specified operation, or the operation itself is invalid. This exception generally indicates a program error such as a reference to an uninitialized variable; so even if you mask all other exceptions, it is recommended that you leave Invalid Operation unmasked. An Invalid Operation exception is signalled when any one of the following conditions occurs:

- One or more of the operands is a Trapping NaN.
- One or more of the operands in the computation sequence was unnormalized or denormalized, and the result cannot be guaranteed because significant information was lost. (Not all operations on unnormalized or denormalized numbers result in loss of significant information; those that do not will not signal Invalid Operation.)
- Any of the following operations is attempted: infinity+infinity (in projective mode), infinity-infinity, 0.0*infinity, infinity*0.0, infinity/infinity, 0.0/0.0, normal number/unnormalized number, normal number/denormalized number (in warning mode).
- In INT, NINT, or IRINT, the operand is too large to fit into the INTEGER format (INTEGER*2 and INTEGER*4 only).
- In comparisons using any of the relational operators, .LT., .LE., .GT., or .GE., the two operands are "'unordered'.

The invalid operation is a "before" error, so that when unmasked, the original operands are available to the exception handler.

The following are specific cases that cause invalid operation exceptions:

- SQRT(x) where x is a negative number, a denormal number (in warning mode), an unnormal number, or $\pm$ infinity (in projective mode).
- SIN(x), COS(x), TAN(x) where x is $\pm$ infinity, or $|x| \geq 2^{-63}$ and x is an unnormal number.
- ARCSIN(x), ARCCOS(x) where x is $\pm$ infinity, or $|x| \geq 2^{-63}$ and unnormal number, or $|x| > 1$.
- ARCTAN(x), EXP(x) $|x| \geq 2^{-63}$ and an unnormal number.
- log (x), log10(x) where x is a negative number, a denormal (in warning mode) or unnormal number, or $\pm$ infinity (in projective mode).
- exp(x) where x is $\pm$ infinity (in projective mode), or $|x| \geq 2^{-63}$ and x is an unnormal number.
- SINH(x), COSH(x), TANH(x) where x is an unnormal number and $|x| \geq 2^{-63}$.
- $\pm$ infinity ***x*, 0**0, and **$\pm$ infinity (all in projective mode).
- $-$ infinity ***x* unless *x* is an INTEGER whole number, $\pm$ infinity **0, and 0**0 (all in affine mode).
- *y****x* where *y* is a negative number and *x* is not a whole number.
- *y****i* where *i* is a negative number, *y* is an unnormal number, and *i* cannot be converted into a 32-bit integer.
- AMOD(*y*,*x*), RMD(*y*,*x*) where *y* is $\pm$ infinity and *x* is unnormal or denormal.

- ATAN2(*y,x*) where *x* and *y* are unnormal numbers and $|y/x| \geq 2^{-63}$, $|x| = |y| = 0$, or $|y| = |y| =$ infinity.
- DIM(*x,y*) where *x* and *y* are infinite (in projective mode).

In some cases, an 8087 invalid exception is raised for valid operations. When not masked (default), the run-time system intercepts the exception before the error handler is invoked and causes program execution to continue normally. These cases are:

- Any otherwise valid arithmetic or conversion operation involving a non-Trapping NaN.
- A comparison between two unordered operands, neither one a Trapping NaN, using the relational operators .EQ. and .NE. See Section 7.7 for descriptions of NaN's and unordered relations.

If the invalid exception is masked at the time of the operation, then the same results occur, but the exception flag is undefined.

15.3.6.2 Denormalized Operand

This exception arises when one or more of the operands is a denormalized number. It can occur if a masked underflow exception has occurred in a previous operation. It is never an error.

The unmasked denormalized exception implements "normalizing mode" arithmetic. The run-time system intercepts these exceptions and takes action as described in Section 7.7.

15.3.6.3 Zero Divide

In a division operation, if the divisor is a normal zero and the dividend is a finite nonzero number, then the zero divide exception occurs. If this exception is masked, the result is infinity. If unmasked, an error occurs and the original operands are available to the exception handler.

Zero divide occurs when an infinity is introduced by an operation that does not overflow. Infinity is the exact answer of the zero divide. The following specific cases result in operation exceptions:

- LOG(0)
- LOG10(0)
- 0***x*, where *x* is negative
- 0***i*, where *i* is negative

15.3.6.4 Overflow

If a rounded result is finite but its exponent is too large to represent in the result floating-point format, the overflow exception occurs. If this exception is masked, an overflow yields infinity, and the precision exception also occurs.

For the operations EXP, SINH, COSH, *y****x*, and *y****i*, overflow is a "before" error. Consequently, when it is unmasked, the original operands are available to the exception handler.

For the operations "+", "—", "*", "/", and DIM, overflow is an "after" error. Consequently, when it is unmasked, a result with a wrapped exponent is available to the exception handler.

15.3.6.5 Underflow

The underflow exception occurs when either of the following conditions arises:

- A rounded result has too small an exponent to be represented in the result floating-point format without normalizing.
- An intermediate product or quotient, where neither operand is a normal zero, is indistinguishable from a normal zero. (This cannot occur with normalized operands.)

If the Underflow exception is masked, the result is a correctly rounded denormalized number or zero.

For the operations *y****x* and *y****i*, underflow is a "before" error. Consequently, when it is unmasked, the original operands are available to the exception handler.

For the operations "+", "—', "*", "/", DIM, ATAN(*y*,*x*), AMOD, and RMD, underflow is an "after" error. Consequently, when it is unmasked, a result with a wrapped exponent is available to the exception handler.

15.3.6.6 Precision

If the correctly rounded result of an operation is not the same as the unrounded value, the precision exception occurs. If this exception is masked, no special action is performed; the correctly rounded result is delivered.

APPENDIX A DIFFERENCES BETWEEN FORTRAN-86 AND OTHER VERSIONS OF FORTRAN

This appendix lists the differences between Fortran-86 and other versions of Fortran. Specifically, the appendix describes the:

- Features of Fortran-86 that are not part of the American National Standards Institute (ANSI) Fortran 77
- Deviations from the ANS-1978 Standard
- Features of Fortran-86 that are different from Fortran-80

The number that appears after each feature listed in Sections A.1 and A.3 refers to the section or chapter of this manual where the feature is described.

A.1 Extensions to Fortran 77

- Binary-, octal-, and hexadecimal-based INTEGER constants. (5.1.1)
- INTEGER values with storage-unit lengths of 1 and 2 bytes. (5.1.1)
- The TEMPREAL data type. (5.1.2)
- A REAL*8 data type that is equivalent to the DOUBLE PRECISION data type. (5.1.2.2)
- LOGICAL values with storage unit lengths of 1 and 2 bytes. (5.1.3)
- Values of different types and lengths within the same storage sequence. (5.10.1)
- 8087 intrinsics. (6.1.2.3)
- The intrinsic functions INT1, INT2, INT4, RINT, IRINT, IDRINT, and TREAL. (6.1.2.2)
- The RMD intrinsic function. (6.1.2.2)
- The %VAL function. (6.1.2.6)
- Bitwise Boolean operations. (7.5)
- Implicit length extensions for INTEGER, REAL, or LOGICAL expressions in assignment statements. (8.1.1)
- A format descriptor to suppress a carriage return on a terminal output device at the end of a record. (9.4.1.2)
- Port-I/O intrinsics for byte and word values. (6.1.1.1)
- The B and Z edit descriptors in the FORMAT statement. (9.4.1.1)
- The CARRIAGE specifier and the RECL specifier for sequential, formatted access in an OPEN statement. (9)
- Hollerith format specifications in INTEGER, REAL, LOGICAL, and DOUBLE PRECISION arrays. Hollerith data-type constants. (Appendix F)
- COMPLEX*16 data type.
- Operations and intrinsic functions for COMPLEX and COMPLEX*16 data.

A.2 Deviations from the ANS-1978 Standard

- **The ENTRY and alternate return features** are not supported.
- The FORMAT edit descriptors **T, TR, TL, S, SS, SP, I***w.m*, and colon are not supported.
- **The INQUIRE statement** is not supported.
- **REAL and DOUBLE PRECISION control expressions** for DO and computed GO TO are not supported.
- **The PARAMETER statement** is restricted to simple constants of any data type, or expressions of type INTEGER. Conversions between INTEGER and floating-point constants are not supported.
- **IOSTAT variables** must be of type INTEGER*2.
- **The Fortran-86 source line size** is not limited to 72 characters; up to 132 source characters per line are accepted by the compiler. This feature is designed to simplify program entry using a video terminal.
- **The DATA statement** may not imply conversion between INTEGER and floating-point constants.

A.3 Differences Between Fortran-80 and Fortran-86

- DATA statements can appear anywhere after the specification statements. (2.2.1)
- The DOUBLE PRECISION data type. (5.1.2.3)
- The TEMPREAL data type. (5.1.2.4)
- CHARACTER data-type functions and substrings. (5.4)
- The PARAMETER statement. (5.3)
- Lower and upper bounds for array dimensions. (5.4.1)
- Generic intrinsic-function names. (6.1.2.2)
- The intrinsic functions INT1, INT2, INT3, RINT, IRINT, IDRINT, RMD, and TREAL. (6.1.2.2)
- Statement functions. (6.1.2.4)
- The %VAL function. (6.1.2.6)
- The D and G edit descriptors. (9.4.1.1)
- Port I/O intrinsics for byte and word values. (6.1.1.1)
- 8087 intrinsics (6.1.2.3)
- New execution-environment interfaces. (Appendix I)
- Changed OPEN-statement semantics. (9.2.1)
- Revised error messages. (15)
- The COMPLEX data type. (5.1.2.5)

APPENDIX B PROCESSOR-DEPENDENT FEATURES OF FORTRAN-86

The following Fortran-86 features are dependent on the 8086, 8087, and 8088 microprocessors on which Fortran-86 programs run. Following each entry is a chapter or section reference where the feature is described in this manual.

- Equivalence of upper- and lowercase letters in the character set. (3.2.2)
- Values of different types and lengths within the same storage sequence. (5.10.1)
- Port-I/O intrinsics for byte and word values. (6.1.1.1)
- Interrupt procedures with the INTERRUPT control. (11.4.10)
- The %VAL function. (6.1.2.6)
- 8087 control intrinsics. (6.1.2.3)
- Reentrant subprograms with the REENTRANT control. (11.4.18)
- Unit preconnection. (14.5)
- The size and structure of storage allocation for variables. (Appendix G)

APPENDIX C COMPILER CAPACITY

This appendix lists the limits imposed on Fortran-86 programs by either Fortran-86 or its environment.

- The compiler accepts up to 19 continuation lines.
- An INTEGER*1 value must be within the range −128 to +127.
- An INTEGER*2 value must be within the range −32,768 to +32,767.
- An INTEGER*4 value must be within the range −2,147,483,648 to +2,147,483,647.
- A REAL value must have magnitude approximately in the range $|1.2 * 10^{(-38)}|$ to $|3.4 * 10^{(38)}|$.
- A DOUBLE PRECISION value must have magnitude approximately in the range $|3.4 * 10^{(-308)}|$ to $|1.8 * 10^{(308)}|$.
- A TEMPREAL value must have the magnitude approximately in the range $|3.4 * 10^{(-4932)}|$ to $|1.2 * 10^{(4932)}|$.
- INTEGER operations addition, subtraction, multiplication, division, and exponentiation are performed modulo 256 for two INTEGER*1 values, modulo 65,536 for two INTEGER*2 values and modulo 4,294,967,296 otherwise.
- The compiler performs INTEGER assignment modulo 256, modulo 65,536, or modulo 4,294,967,296 if the target variable has the data type INTEGER*1, INTEGER*2, or INTEGER*4, respectively.
- Subscript values are taken modulo 65,536 for arrays declared to be less than 65,536 bytes in length; otherwise modulo 4,294,967,296 applies.
- The variable specified in an IOSTAT option must be INTEGER*2.
- The maximum record lengths for I/O are as follows: 65535 for unformatted sequential, 32767 for formatted direct, and 32767 for unformatted direct. There is no limit to the record length for formatted sequential.

APPENDIX D LANGUAGE SUMMARY

This appendix summarizes the Fortran-86 statements, and special punctuation symbols.

D.1 Statement Summary

ASSIGN Statement

Syntax

ASSIGN *stl* TO *name*

Function

Assign a statement label *stl* to an integer variable *name*

Category

Executable

Assignment Statement

Syntax

name = *exp*

Function

Assign the value of an expression *exp* to a variable *name*

Type

Arithmetic, Logical, Character

Category

Executable

BACKSPACE Statement

Syntax

BACKSPACE *unit*
BACKSPACE *arg-list*

Function

Position file connected to *unit* before preceding record

where

unit	is the unit specifier.	
arg-list	is	
	`[UNIT=]`*unit*	unit specifier
	`IOSTAT=`*stname*	I/O status specifier
	`ERR=`*stl*	error specifier
BACKSPACE	is for sequential files only.	

Category

Executable

BLOCK DATA Statement

Syntax

`BLOCK DATA`[*name*]

Function

Identify and optionally *name* a BLOCK DATA subprogram.

Category

Nonexecutable

CALL Statement

Syntax

`CALL` *name*[`(`[*arg*[`,` *arg*]...;`)`]

Function

Call the subroutine, *name* with actual argument(s) *arg*.

Category

Executable

CHARACTER Statement

Syntax

`CHARACTER`[**len*]*name*[**len*][, *name*[**len*]]...

Function

Specify *name* and *len* for character type variable or array.

Category

Nonexecutable, specification, type

CLOSE Statement

Syntax

CLOSE(*close-list*)

Function

Close the file described by *close-list*

where

close-list	is	
	[UNIT=]*unit*	unit specifier
	IOSTAT=*stname*	I/O status specifier
	ERR=*stl*	error specifier
	STATUS=*stat*	file disposition specifier

Category

Executable

Comment Line Statement

Syntax

The character 'C' or asterisk (*) in position 1; any other characters in positions 2-72.

Function

Program documentation

Category

Nonexecutable

COMMON Statement

Syntax

COMMON[/*name*]/]*nlist*[[,]/*name*/*nlist*]...

Function

Name and define the contents of COMMON block(s), *name*. If *name* is not specified, a blank COMMON is defined.

Category

Nonexecutable, specification

COMPLEX and COMPLEX*16 Type Statement

Syntax

```
COMPLEX [*len], name[*len], name[*len],...
```

Function

Declares a variable name, function name, or dummy procedure to be of type COMPLEX with length len.

Category

Nonexecutable, specification type.

CONTINUE Statement

Syntax

```
CONTINUE
```

Function

No effect unless this is the terminal statement of a DO loop; then action depends on the DO variable.

DATA Statement

Syntax

```
DATA nlist/clist...
```

Function

Assign values in *clist* to the items in *nlist*.

Category

Nonexecutable

DIMENSION Statement

Syntax

`DIMENSION` *array*`(`*d*`)`[, *array*`(`*d*`)`]...

Function

Name *array*(s) and define dimension(s) *d*.

Category

Nonexecutable, specification

DO Statement

Syntax

`DO` *stl*[,]*var*`=`*e1* , *e2*[, *e3*]

Function

Define the beginning of DO loop and set up loop counters

where

stl	is label of last (executable) statement in DO loop.
var	is DO loop index variable.
e1	is initial loop index value.
e2	is loop termination value.
e3	loop increment/decrement value.

Category

Executable

DOUBLE PRECISION Statement

Syntax

`DOUBLE PRECISION` *name*[, *name*]...

Function

Specify name(s) for a double precision type variable or array.

Category

Nonexecutable, specification, type

ELSE Statement

Syntax

```
ELSE
```

Function

Provides alternate execution path from IF or ELSE IF.

Category

Executable, block IF

ELSE IF Statement

Syntax

ELSE IF(*exp*)THEN

Function

Continue execution if expression *exp* is TRUE.

Category

Executable, Block IF

END Statement

Syntax

```
END
```

Function

Terminate main program; return from subprogram; mark end of program unit.

Category

Executable

END IF Statement

Syntax

```
END IF
```

Function

Mark end of IF block; continue execution.

Category

Executable, block IF

ENDFILE Statement

Syntax

```
ENDFILE unit
ENDFILE(arg-list)
```

Function

Write end-of-file record on file connected to *unit*

where

unit	is the unit specifier.	
arg-list	is	
	[UNIT=]*unit*	unit specifier
	IOSTAT=*stname*	I/O status specifier
	ERR=*stl*	error specifier
ENDFILE	is for sequential files only.	

Category

Executable

EQUIVALENCE Statement

Syntax

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

Function

Allow entries in *nlist* to share the same storage area.

Category

Nonexecutable, specification

EXTERNAL Statement

Syntax

```
EXTERNAL name[,name]...
```

Function

Allows the name of an external/dummy procedure name to be used as an actual argument.

Category

Nonexecutable, specification

FORMAT Statement

Syntax

stl FORMAT([*flist*])

Function

Specifies the format of formatted I/O data where *flist* includes the following repeatable and nonrepeatable edit descriptors

Repeatable		Nonrepeatable	
I*w*	integer	'*string*'	literal
F*w,d*	real	*n*H*string*	Hollerith
E*w,d*[E*e*]	real	*n*X	record position
D*w,d*	real	/	record termination
G*w,d*[E*e*]	real	*k*P	scale factor
L*w*	logical	BN	blank
A[*w*]	alphanumeric	BZ	blank
B*w*	binary	$	alternate-record
Z*w*	hexadecimal		termination

Category

Nonexecutable

FUNCTION Statement

Syntax

[*type*]FUNCTION *name*([*arg*[, *arg*]...])

Function

Name the FUNCTION subprogram and define its type and dummy argument(s).

Category

Nonexecutable

GO TO Statement

Syntax

GO TO *stl*
GO TO (*stl*[, *stl*]...)*exp*
GO TO *name*[(*stl*[, *stl*]...)]

Function

Transfer control to statement labelled *stl* or ASSIGNED to variable *name*. The first branches unconditionally; the second branches based on the value of the integer expression *exp*; the third branches unconditionally, but statement label corresponding to *name* must be included in list.

Category

Executable

IF Statement

Syntax

```
IF(exp)s1,s2,s3
IF(exp)stmt
IF(exp)THEN
```

Function

Transfer control to a specified statement or perform specified action(s) based on the value of the expression *exp*. In the first format, *exp* is an arithmetic expression and *s1*, *s2*, and *s3* are statement labels; control passes to:

s1 if $exp<0$
s2 if $exp=0$
s3 if $exp>0$

In the second format, the statement *stmt* is executed if the logical expression is TRUE. Third format introduces IF block; statements following IF—THEN are executed if logical expression is TRUE.

Category

Executable

IMPLICIT Statement

Syntax

```
IMPLICIT ntype(let[let]...)...
```

Function

Define implicit typing for variable names whose first letter is *let* or in the range *let-let*.

Category

Nonexecutable, specification

INTEGER Statement

Syntax

INTEGER[**len*]*name*[**len*][*name*[**len*]]...

Function

Define *name* to be of type integer with length *len*.

Category

Nonexecutable, specification, type

INTRINSIC Statement

Syntax

INTRINSIC *name*[, *name*]...

Function

Allow intrinsic function(s) to be used as actual argument(s).

Category

Nonexecutable, specification

LOGICAL Statement

Syntax

LOGICAL[**len*]*name*[**len*][, *name*[**len*]]...

Function

Define *name* to be of type logical with length *len*

Category

Nonexecutable, specification, type

OPEN Statement

Syntax

OPEN(*OPEN-LIST*)

Function

Open the specified file with *open-list* consisting of the following:

`[UNIT=]`*unit*	unit specifier
`IOSTAT=`*stname*	I/O status specifier
`ERR=`*stl*	error specifier
`FILE=`*fname*	filename specifier
`STATUS=`*stat*	file status specifier
`ACCESS=`*acc*	access method specifier
`FORM=`*fmat*	formatting specifier
`RECL=`*reclen*	record length specifier
`BLANK=`*blnk*	blank specifier
`CARRIAGE=`*car*	carriage control specifier

Category

Executable

PAUSE Statement

Syntax

PAUSE[*msg*]

Function

Halt program execution; resume under control of external signal; *msg* is 1-5 digits or a character constant.

Category

Executable

PARAMETER Statement

Syntax

`PARAMETER(`*name*`=`*exp...*`)`

Function

Assigns a *name* to a constant expression *exp.*

Category

Nonexecutable, specification

PRINT Statement

Syntax

`PRINT` *f*[, *outlist*]

Function

Output items in *outlist* to preconnected unit in format specified by *f*.

Category

Executable

PROGRAM Statement

Syntax

PROGRAM *name*

Function

Optionally name main-program unit. If missing, the compiler will assign @MAIN as the program name.

Category

Nonexecutable

READ Statement

Syntax

READ(*ctl-list*)[*inlist*]
READ *f*[, *inlist*]

Function

Input items in *inlist* as directed by specified controls in *ctl-list*

[UNIT=]*unit*	unit specifier
[FMT=]*f*	format specifier
REC=*recno*	record number specifier
IOSTAT=*stname*	I/O status specifier
ERR=*stl*	error specifier
END=stl	end-of-file specifier

Second format is for preconnected units; *f* is the format specifier.

Category

Executable

REAL Statement

Syntax

REAL[**len*]*name*[**len*][, *name*[**len*]]...

Function

Define *name* to be of type real with length *len*.

Category

Nonexecutable, specification, type

RETURN Statement

Syntax

```
RETURN
```

Function

Return from FUNCTION or SUBROUTINE subprogram.

Category

Executable

REWIND Statement

Syntax

```
REWIND unit
REWIND(arg-list)
```

Function

Reposition file connected to *unit* at its initial point with *arg-list* including:

`[UNIT=]`*unit*	unit specifier
`IOSTAT=`*stname*	I/O status specifier
`ERR=`*stl*	error specifier

REWIND is for sequential files only.

Category

Executable

SAVE Statement

Syntax

```
SAVE name[,name]...
```

Function

Save data in *name* on return from subprogram.

Category

Nonexecutable, specification

Statement Function Statement

Syntax

name ([*arg* [, *arg*]...]) = *exp*

Function

Define function *name*

Category

Nonexecutable

STOP Statement

Syntax

STOP[*msg*]

Function

Terminate program execution, with optional message, *msg*.

Category

Executable

SUBROUTINE Statement

Syntax

SUBROUTINE *name*[([*arg* [, *arg*]...])]

Function

Define SUBROUTINE subprogram *name* with dummy argument(s) *arg*.

Category

Nonexecutable

TEMPREAL Statement

Syntax

TEMPREAL *name*[, *name*]...

Function

Define *name* to be of type tempreal.

Category

Nonexecutable, specification, type

WRITE Statement

Syntax

WRITE(*ctl-list*)[*outlist*]

[UNIT=]*unit*	unit specifier
[FMT=]*f*	format specifier
REC=*recno*	record number specifier
IOSTAT=*stname*	I/O status specifier
ERR=*stl*	error specifier

D.2 Symbol Summary

Table D-1 lists the arithmetic operators and their meanings.

Table D-1. Arithmetic Operators

Operator	Meaning
**	Exponentiation
/	Division
*	Multiplication
+	Addition
−	Subtraction

Table D-2 lists the relational operators and their meanings.

Table D-2. 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

Table D-3 lists the logical operators and their meanings.

Table D-3. Logical Operators

Operator	Meaning
.NOT.	Logical Negation
.AND.	Logical Conjunction
.OR.	Logical Inclusive Disjunction
.EQV.	Logical Equivalence
.NEQV.	Logical Nonequivalence

APPENDIX E CHARACTER SET AND COLLATING SEQUENCE

ASCII CHARACTER	HEX	FORTRAN-86 CHARACTER	ASCII CHARACTER	HEX	FORTRAN-86 CHARACTER
NUL	00	no	@	40	no
SOH	01	no	A	41	yes
STX	02	no	B	42	yes
ETX	03	no	C	43	yes
EOT	04	no	D	44	yes
ENQ	05	no	E	45	yes
ACK	06	no	F	46	yes
BEL	07	no	G	47	yes
BS	08	no	H	48	yes
HT	09	no	I	49	yes
LF	0A	no	J	4A	yes
VT	0B	no	K	4B	yes
FF	0C	no	L	4C	yes
CR	0D	no	M	4D	yes
SO	0E	no	N	4E	yes
SI	0F	no	O	4F	yes
DLE	10	no	P	50	yes
DCI	11	no	Q	51	yes
DC2	12	no	R	52	yes
DC3	13	no	S	53	yes
DC4	14	no	T	54	yes
NAK	15	no	U	55	yes
SYN	16	no	V	56	yes
ETB	17	no	W	57	yes
CAN	18	no	X	58	yes
EM	19	no	Y	59	yes
SUB	1A	no	Z	5A	yes
ESC	1B	no	[	5B	no
FS	1C	no	\	5C	no
GS	1D	no	]	5D	no
RS	1E	no	∧(↑)	5E	no
US	1F	no	—	5F	yes
space	20	yes	`	60	no
!	21	no	a	61	yes
“	22	no	b	62	yes
#	23	yes	c	63	yes
$	24	yes	d	64	yes
%	25	no	e	65	yes
&	26	no	f	66	yes
'	27	yes	g	67	yes
(	28	yes	h	68	yes
)	29	yes	i	69	yes
*	2A	yes	j	6A	yes
+	2B	yes	k	6B	yes
,	2C	yes	l	6C	yes
−	2D	yes	m	6D	yes
.	2E	yes	n	6E	yes
/	2F	yes	o	6F	yes
0	30	yes	p	70	yes
1	31	yes	q	71	yes
2	32	yes	r	72	yes
3	33	yes	s	73	yes
4	34	yes	t	74	yes
5	35	yes	u	75	yes
6	36	yes	v	76	yes
7	37	yes	w	77	yes
8	38	yes	x	78	yes
9	39	yes	y	79	yes
:	3A	yes	z	7A	yes
;	3B	no	{	7B	no
<	3C	no	|	7C	no
=	3D	yes	}	7D	no
>	3E	no	≃	7E	no
?	3F	no	DEL	7F	no

APPENDIX F HOLLERITH DATA TYPE

This appendix describes the Hollerith data type that is a carryover from Fortran 66. The character data type provides better processing capability but the Hollerith type has been retained for compatability.

F.1 Hollerith as a Data Type

Although Hollerith is a data type, a symbolic name cannot be of type Hollerith. You identify Hollerith data (other than Hollerith constants) using an INTEGER, floating-point (REAL, DOUBLE PRECISION, TEMPREAL, COMPLEX, COMPLEX*16), or LOGICAL type name. You cannot use type CHARACTER.

You can define INTEGER, floating-point or LOGICAL items with a Hollerith value using either DATA or READ statements. Equivalenced items become associated with that Hollerith value also. When this definition occurs, the defined item loses its INTEGER, floating-point, or LOGICAL characteristic.

F.2 Hollerith Constants

The format of a Hollerith constant is

*n*H*h1h2...hn*

where

n	is a nonzero, unsigned, integer constant.
h	is any representable character.

Blanks are significant in the character string following the H.

Hollerith constants can appear only in DATA statements and in the argument list of CALL statements.

F.2.1 Hollerith Constants in DATA Statements

A Hollerith constant may appear in the *clist* of a DATA statement. The corresponding argument in *nlist* must be type INTEGER, floating-point, or LOGICAL.

For an argument of type INTEGER, floating-point or LOGICAL, the number of characters *n* in the corresponding Hollerith constant must be less than or equal to *g* (where *g* is the length of the argument in bytes). If *n* is less than *g*, the compiler initializes the argument with the *n* Hollerith characters extended on the right with *g–n* blank characters.

Each Hollerith character initializes exactly one variable or array element.

F.2.2 Hollerith Constants in CALL Statements

An actual argument in a CALL statement can be a Hollerith constant, as long as the corresponding dummy argument has type INTEGER, floating-point, or LOGICAL. This is an exception to the rule that actual and dummy arguments must agree in type. The length of the dummy argument, however, must agree with the length of the actual argument.

F.3 Hollerith Format Specification

A format specification can be an array name of type INTEGER, floating-point, or LOGICAL. In this case, the leftmost characters of the specified entity must contain Hollerith data constituting a legal format specification. Blank characters may precede the format specification and data may follow the right parenthesis ending the specification with no affect.

A Hollerith format specification must not contain an apostrophe edit descriptor or an H edit descriptor.

F.4 "A" Editing of Hollerith Data

You can use the A*w* edit descriptor with Hollerith data if the corresponding I/O list item has type INTEGER, floating-point, or LOGICAL.

Editing is the same as for A*w* editing of character data, except that *n* is the maximum number of characters that the system can store in the list item.

APPENDIX G RUN-TIME DATA REPRESENTATIONS

The Fortran-86 compiler determines the amount of storage needed at run time for each data type, and the run-time support software allocates the storage when you execute the Fortran program. This appendix describes the storage necessary for each data type.

G.1 Storage Units

There are two types of storage units: numeric storage units and character storage units. A numeric storage unit is one, two, four, or ten bytes depending on the length of the specified data type. The standard length is four bytes. A character storage unit is always one byte. A storage sequence is a consecutive series of either numeric storage units or character storage units depending on the type of the data..

G.2 Data Types

Fortran-86 supports the following data types: INTEGER, REAL, DOUBLE PRECISION, TEMPREAL, COMPLEX, COMPLEX*16, CHARACTER, and LOGICAL. Table G-1 summarizes the storage necessary for each data type.

Table G-1. Summary of Storage Units

Data Type	Number of Units	Length of Unit	Bytes
INTEGER*1	1	1 byte	1
INTEGER*2	1	2 bytes	2
INTEGER*4	1	4 bytes	4
REAL*4	1	4 bytes	4
REAL*8	2	4 bytes	8
DOUBLE PRECISION	2	4 bytes	8
TEMPREAL	1	10 bytes	10
COMPLEX*8	2	4 bytes	8
COMPLEX*16	4	4 bytes	16
LOGICAL*1	1	1 byte	1
LOCIGAL*2	1	2 bytes	2
LOGICAL*4	1	4 bytes	4
CHARACTER*n (0≤n≤256)	n	1 byte	n

APPENDIX H LINKING TO SUBPROGRAMS WRITTEN IN OTHER LANGUAGES

This appendix describes the calling conventions used by iAPX 86,88 family languages. These calling conventions are standardized so that a program written in Fortran-86 can communicate with procedures, subroutines, and subprograms written in other iAPX 86,88 family languages.

NOTE

The information contained in this appendix is dependent on current implementations of the Fortran-86, PL/M-86, C-86, and Pascal-86 compilers. As such, it is subject to change with any new version of one of these compilers. Programmers using this appendix are urged to carefully document assumptions based on this information to enable upgrading to new versions as they are released, if necessary. Any changes will be reflected in the respective language user's guides.

As a Fortran-86 programmer linking PL/M-86, C-86, or Pascal-86 procedures with Fortran-86, you need to know the PL/M-86, C-86, and Pascal-86 data types that match Fortran-86 data types and the order and number of arguments to supply for the C-86, PL/M-86, or Pascal-86 parameters, described in Section H.2. You must also know how to link subprograms, as described in Chapter 14.

PL/M-86, C-86, and Pascal-86 procedures linking with Fortran-86 procedures must be compiled under the LARGE model of segmentation.

As a Fortran-86 programmer calling 8086/8087/8088 Macro Assembly Language subroutines, you need to know the calling conventions of the stack and register usage and the corresponding data types, described in this appendix, in order to write an assembly language subroutine that can pick up the data your Fortran-86 program passes to it. The same information is necessary for a macro assembly language programmer calling Fortran-86 subprograms. Refer to the *8086/8087/8088 Macro Assembler Operating Instructions for 8086-Based Development Systems*, Appendix B, for more information about linking to the macro assembly language programs and for examples of linking such programs to PL/M-86 programs.

H.1 Introduction

A Fortran-86 program consists of a main program and any number of subprograms. Not all of these program units have to be written in Fortran-86. You can choose the appropriate language for each subprogram as long as you link the subprograms properly with LINK86, the 8086-based linker. Since the iAPX 86,88 family languages follow the same calling conventions, control will pass to a subprogram called correctly. However, the called subprogram may not be able to deal intelligently with the data passed to it, because different languages treat data structures differently.

NOTE

Subprogram is a term used in Fortran-86 referring to both subroutines and functions. *Procedure* is the term used in Pascal-86, C-86, and PL/M-86. The assembly language term is *subroutine*. In this appendix, the word *subprogram* denotes any entity written in any iAPX 86,88 language that can call a Fortran-86 subroutine or function, or be called from a Fortran-86 subprogram.

If you want to link your Fortran-86 application with a subprogram written in Pascal-86 or PL/M-86, Section H.2 should be sufficient for your needs. However, if your main program is written in PL/M-86, you must also know how to initialize the Fortran-86 run-time environment described in Section H.5.

Writing assembly-language subprograms to be called from Fortran-86 programs requires an understanding of this entire appendix.

The calling convention for programs written in C differs from the calling conventions used for Fortran-86, Pascal-86, and PL/M-86. As a result, a primary control named INTERFACE has been provided that allows Fortran-86 programs to call procedures and functions written in C. This control allows procedures written in C to call Fortran-86 programs.

H.2 Calling Sequence

The calling convention for the invocation of a subprogram is essentially the same for Fortran-86, Pascal-86, and PL/M-86 (LARGE model of segmentation), for most equivalent argument types. The arguments are pushed on the 8086 or 8087 stack in left-to-right order, and then the subprogram is invoked with an 8086 intersegment call instruction. If the subprogram is a function, the returned value is delivered in predefined 8086 registers, on the top of the 8087 stack, or via an additional reference parameter on the 8086 stack, depending on the value's data type.

You can see the pseudo-assembly listing of this sequence if you specify the CODE control when compiling a Fortran-86 program that contains a reference to an external subroutine or function.

The called subprogram has the responsibility of saving certain 8086 registers and restoring them before returning to the caller. The subprogram also removes the arguments from the stack. A pseudo-assembly listing (CODE control) of a Fortran-86 SUBROUTINE or FUNCTION will illustrate these instruction sequences, which are similar to those generated for Pascal-86 and PL/M-86 subprograms.

H.2.1 Arguments

There are two methods of passing arguments to other subprograms: by value and by reference. The first method, *by value*, passes the actual value of the argument to the subprogram. With the second method, *by reference*, the address of the argument is passed to the subprogram, and the called subprogram must use the address to locate the data associated with the argument. The called program must know which method is being used for each argument.

In Fortran-86, arguments for subprograms are passed by reference on the 8086 stack. PL/M-86 subprograms linking with Fortran-86 must use long (doubleword) pointers to pass or accept arguments. Pascal-86 arguments must be specified as VAR parameters when communicating with standard Fortran-86 subprograms.

Fortran-86 provides the nonstandard %VAL function (see Section 6.2.1.7) that creates or accepts a value argument for certain simple data types. While the %VAL method is useful for linking with existing non-Fortran subprograms, the reference method is standard and strongly recommended to ensure software portability.

In the following sections, arguments are assumed to be passed by reference unless otherwise qualified. Pointers are always long (double words).

H.2.2 Returned Values

The methods of returning values from function subprograms is consistent across Fortran-86, PL/M-86, and Pascal-86 for all supported data types. The following rules apply:

1. All floating-point data types are returned on the top of the 8087 stack. In the case of COMPLEX values, the real component is pushed onto the stack first and the imaginary part is pushed last.
2. A character string (Fortran only) is returned via a CHARACTER argument for the target location provided by the calling program. The calling sequence is the same as that of a SUBROUTINE call with the target string location specified as the first argument (see Section H.2.3.4). The calling subprogram determines the length of the returned string.
3. All other data types allowable as returned values are returned in 8086 registers depending on their length: 1-byte values in AL, 2-byte values in AX, and 4-byte values in DX/AX.

H.2.3 Data Types

Data-type compatibility between Fortran-86 and PL/M-86, C-86, or Pascal-86 varies considerably due to the characteristics of these languages and their implementation. ASM-86, having the weakest typing of all, can be considered to be fully compatible with Fortran-86 as long as Fortran-86 data-type conventions are followed.

H.2.3.1 Floating-Point Data Types

Fortran-86's REAL*4 is identical to REAL in both PL/M-86 and Pascal-86. REAL*8 and DOUBLE PRECISION are supported in Pascal-86 and C-86, but not in PL/M-86. TEMPREAL is supported in Pascal-86 but not in PL/M-86 or C-86. TEMPREAL is supported in Pascal-86 but not in PL/M-86 or C-86. Floating-point values in ASM-86 must have data formats as defined by the *8086,87 Family User's Guide Numerics Supplement.* In addition, Fortran-86 distinguishes Trapping NaN's from nontrapping NaN's by the most significant bit of the significand.

Use of the %VAL function with floating-point variables is not supported by floating-point data types.

H.2.3.2 Integer Data Types

INTEGER*2 in Fortran-86 is equivalent to INTEGER in both PL/M-86 and Pascal-86, and INTEGER*1 is compatible with the Pascal-86 subrange −128...+127 and with PL/M-86's BYTE for positive values that are less than 128. INTEGER*4 is supported by Pascal-86 but not PL/M-86. INTEGER*4 is compatible with the PL/M-86 DWORD for positive values, and LONGINT for Pascal-86. ASM-86 subprograms can support all Fortran INTEGER types when only signed operations are used.

Any Fortran INTEGER type may be passed by value on the 8086 stack using the %VAL function.

INTEGER data types used in bitwise boolean operations are compatible with Pascal-86's SET type, if the field lengths and bit sequences are carefully observed. See the *Pascal-86 User's Guide* for implementaion details.

H.2.3.3 Logical Data Types

With Fortran's LOGICAL data types, only the least significant bit is relevent (0 for .FALSE., 1 for .TRUE.), and the remaining bits are undefined. LOGICAL*1 in Fortran-86 is the same as PL/M-86's BYTE data type used in boolean expressions. While Pascal-86's BOOLEAN type is fully acceptable to a Fortran-86 subprogram as a LOGICAL*1 dummy argument or returned value, the reverse is not supported.

The problem of passing or returning a LOGICAL*1 value to a Pascal-86 subprogram is that Pascal-86 requires all high-order bits to be zero (see Table H-1), whereas these bits are unpredictable in Fortran-86. Use of INTEGER*1 containing the integer 0 (.FALSE.) or 1 (.TRUE.) is a way to bypass this restriction.

All Fortran-86 LOGICAL data types may be passed using the %VAL function, except that LOGICAL*4 must not be used with %VAL when linking with Pascal-86.

Table H-1. Fortran-86 Data Types and Their Equivalents in Pascal-86, C-86, PL/M-86, and ASM-86

Fortran-86	Pascal-86	PL/M-86	ASM-86	C-86
REAL*4	REAL LONGREAL TEMPREAL	REAL	DD (8087 single precision	real
REAL*8 or DOUBLE PRECISION			DQ (8087 double precision)	double
TEMPREAL			DT (8087 extended precision)	
INTEGER*1	[0...127]	BYTE (1)	DB (signed)	int
INTEGER*2	INTEGER	INTEGER	DW (signed)	long int
INTEGER*4			DD (signed)	
LOGICAL*1	BOOLEAN (2)	BYTE (2)	DB (2)	
LOGICAL*2			DW (2)	
LOGICAL*4			DD(2)	
CHARACTER*n	{ array [1...*n*] of CHAR, INTEGER } (2)	{ BYTE (*n*), INTEGER } (3)	{ DB nDUP, word } (3)	int char[*n*]

COMPLEX*8	Record
	Realpart: Real;
	Imagpart: Real;
	End;
		Structure(
		Realpart Real
		Imagpart Real)
			STRUC
			REALPART DD/DQ
			IMAGPART DD/DQ
			ENDS
				struct {
				real REALPART;
				real IMAGPART }

COMPLEX*16	Record
	REALPART: LONGREAL;
	IMAGPART: LONGREAL;
	End;
		Structure(
		Realpart Real
		Imagpart Real)
			STRUC
			REALPART DD/DQ
			IMAGPART DD/DQ
			ENDS
				struct {
				double REALPART;
				double IMAGPART }

NOTES:

(1) For values 0 through 127 only.

(2) Only rightmost significant bit; Remaining bits are undefined, except for Pascal-86, which requires them to be zero.

(3) See Section H.2.3.4

H.2.3.4 Character Data Types

Fortran-86 character-string arguments and returned values are passed in a unique manner that is not directly supported by PL/M-86 or Pascal-86. Familiarity with Fortran-86 conventions, however, will enable you to pass or accept Fortran character strings to or from non-Fortran subprograms.

A Fortran-86 character-string argument has two components: the address of the string and its length. For each CHARACTER argument, a word containing the string length is placed (by value) immediately after the corresponding string address on the 8086 stack. Note that this description also applies to the argument inserted by the compiler to receive the returned value of a CHARACTER function (see Section H.2.2). In both cases, the calling subprogram specifies the string length.

CAUTION

All Fortran arguments of type CHARACTER[**n*] are passed in the same manner. CHARACTER*1 is not the same as PL/M-86's BYTE or Pascal-86's CHAR.

Example:

A Fortran function is defined as follows:

```
CHARACTER*8 FUNCTION CHFUN(A)
CHARACTER*(*)A
        .
        .
        .
```

A PL/M-86 program can invoke this function using the following procedure declaration:

```
CHFUN: PROCEDURE (RES,RLEN,ARG,ALEN)EXTERNAL;
DECLARE (RES,ARG) POINTER;
DECLARE (RLEN,ALEN) INTEGER;
END;
```

In this example, the character strings pointed to by RES and ARG are BYTE arrays whose lengths are specified by the caller in RLEN and ALEN, respectively. Note that any string lengths defined in the function for arguments and returned values are ignored.

Use of %VAL with character strings is not supported in Fortran-86.

H.2.3.5 Arrays and Structures as Arguments

Fortran-86 array arguments are fully compatible with those of PL/M-86 and Pascal-86, as long as the component data types are compatible. The argument consists of a long pointer (two words) on the 8086 stack. Fortran-86 has no structure or record data type except for LARGE ARRAY support.

NOTE

For multidimensional arrays, Fortran dimensions are specified in reverse sequence from those of Pascal-86 and PL/M-86.

Use of the %VAL function is not supported for arrays.

H.2.3.6 Procedures as Arguments

Procedure arguments are fully compatible between Fortran-86 and PL/M-86 (LARGE model). They are passed by reference using a long pointer (two words) on the 8086 stack. Use of the %VAL function is not supported.

Procedure arguments cannot be passed between Pascal-86 and Fortran-86.

H.2.4 Further Linkage Considerations

Fortran-86 subprograms always assume that the 8087 stack is completely empty on entry to the subprogram. On return it will contain one floating-point returned value, with the exception of COMPLEX data, which returns two values. PL/M programs must not call Fortran-86 functions within floating-point expressions or parameter lists since PL/M-86 does not conform to this convention.

H.2.5 C Language Considerations

The INTERFACE control handles differences in calling conventions, making it possible to link modules written in C with those written in Fortran-86, Pascal-86, PL/M-86, or ASM86.

H.3 Register Usage

A Fortran-86 subprogram assumes that all 8086 and 8087 registers and flags are volatile and need not be saved/restored before returning, except for the following:

- SS stack-segment register (never changed)
- CS code-segment register (restored by RETURN)
- DS data-segment register (saved on entry, restored on return)
- BP stack-base pointer (saved on entry, restored on return)
- SP top-of-stack pointer (restored, and arguments deleted, on return)

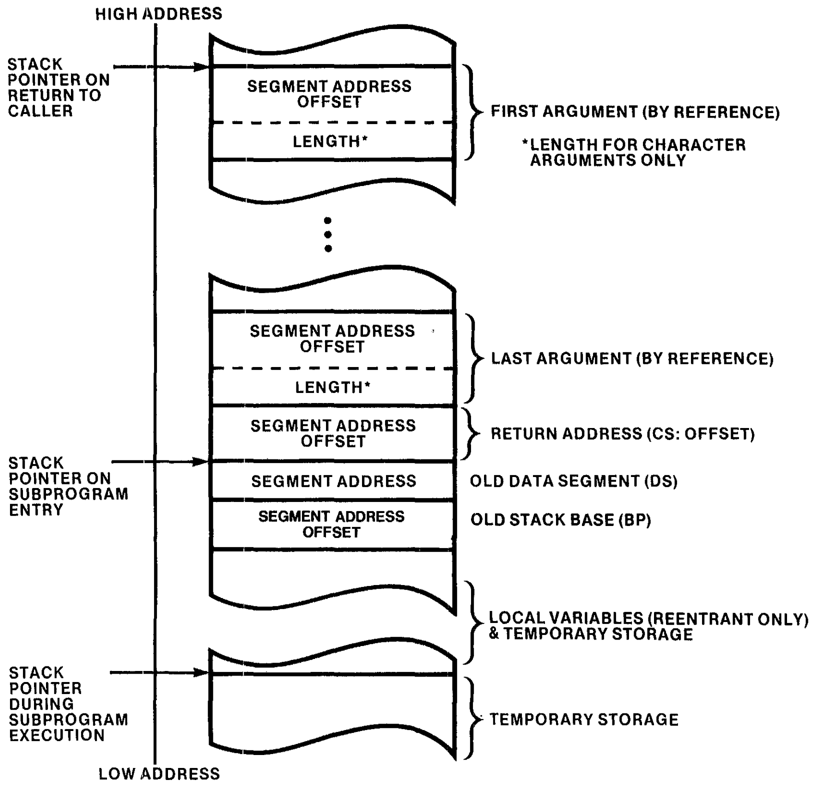
The 8087 stack is assumed to be completely empty on entry, and will contain one floating-point value on return, with the exception of COMPLEX data, which returns two values.

Assembly-language subprograms called by Fortran-86 programs are expected to conform to these Fortran-86 conventions. It is recommended that you compile sample Fortran-86 subprograms with the CODE control as an illustration before writing your ASM-86 subprogram.

H.4 Stack Usage

Each 8086 stack position holds one word. Arguments passed by reference normally take two words, the segment address and offset. Character arguments require a third word to pass the length (value). Arguments passed by value take one or two words, depending on the data length. One-byte arguments have an undefined high-order byte.

Figure H-1 shows the 8086 stack layout for a Fortran-86 subprogram.


HIGH ADDRESS
STACK POINTER ON RETURN TO CALLER
SEGMENT ADDRESS OFFSET
LENGTH*
FIRST ARGUMENT (BY REFERENCE)
*LENGTH FOR CHARACTER ARGUMENTS ONLY
SEGMENT ADDRESS OFFSET
LENGTH*
LAST ARGUMENT (BY REFERENCE)
SEGMENT ADDRESS OFFSET
RETURN ADDRESS (CS: OFFSET)
STACK POINTER ON SUBPROGRAM ENTRY
SEGMENT ADDRESS
OLD DATA SEGMENT (DS)
SEGMENT ADDRESS OFFSET
OLD STACK BASE (BP)
LOCAL VARIABLES (REENTRANT ONLY) & TEMPORARY STORAGE
STACK POINTER DURING SUBPROGRAM EXECUTION
TEMPORARY STORAGE
LOW ADDRESS

NOTE: REFERENCE ARGUMENTS CAN BE REPLACED BY ACTUAL VALUES WHEN USING %VAL FUNCTION.

Figure H-1. 8086 Stack Layout During Execution of a Fortran-86 Subprogram 121570-7

All the elements past the return address are pushed on the stack by the called program, and need to be saved only when they are changed by the called subprogram. The arguments are removed on return using the RET *n* instruction. See the *8086/8087/8088 Macro Assembler Operating Instructions for 8086-Based Development Systems* for further details of stack management.

H.5 Initialization of the Fortran-86 Run-Time Environment

If your application program consists of a main program written in PL/M-86 or ASM-86 and one or more subprograms written in Fortran-86, you must explicitly initialize Fortran-86's run-time environment. Fortran-86 and Pascal-86 share the same environment.

Compiling a sample Fortran program using the CODE control illustrates this initialization.

Each main program must execute the following two instructions before invocation of any Fortran-86 subprogram:

```
CALL INITFP
CALL TQ_001
```

Execution of your program is terminated using:

```
CALL TQ_999
```

The procedure TQ_001 initializes global I/O tables and error-handling facilities for both Fortran-86 and Pascal-86. If called more than once during program execution, TQ_001 will normally destroy the previous status of the I/O system. TQ_999 closes Fortran-86 and Pascal-86 files, and halts execution of the program.

INITFP initializes the floating-point environment. If your application does not perform any floating-point operations, you should still include this call to allow for future changes. See Section 14.2.2 for a description of the libraries that resolve these external references, and instructions on how to configure your object programs at link time.

Figure H-2 lists a sample ASM-86 program that calls Fortran-86 subprograms.

```
8086/8087/8088 MACRO ASSEMBLER    ASMEX                                          09/01/80  PAGE    1

SERIES-III 8086/8087/8088 MACRO ASSEMBLER V1.0 ASSEMBLY OF MODULE ASMEX
OBJECT MODULE PLACED IN :F1:ASMEX.OBJ
INVOCATION LINE CONTROLS:  PRINT(:F1:ASMEX.LST) OBJECT(:F1:ASMEX.OBJ)

LOC  OBJ                  LINE     SOURCE

                                   NAME    ASMEX
                                   ;
                                   ;This program demonstrates procedure linkage to FORTRAN-86,
                                   ;focusing on the parameter passing conventions.
                                   ;
                                   ;This procedure takes four arguments for four parameters:
                                   ;a character variable, an integer*2 value, an integer (or logical)
                                   ;variable, and an integer function.
                                   ;They are pushed onto the stack in that order.
                                   ;They must be popped at exit (with RET instruction).
                                   ;
                                   ;The prologue code saves BP, and points BP to the
                                   ;structure defined below.  After prologue executes,
                                   ;stack looks like this:
                                   ;
                                   ;       high memory
                                   ;      ----------------
                                   ;      | (segment)    |}
                                   ;      ----PARM1-------}
                                   ;      | (offset)     |}--->argument A
                                   ;      ----PARM1-------}
```

Figure H-2. Sample ASM-86 Program

```
                            22     ;      | (size)        ||
                            23     ;      ----------------
                            24     ;      |   PARM2       |---->argument B
                            25     ;      ----------------
                            26     ;      | (segment)     ||
                            27     ;      ----PARM3-------}--->argument C
                            28     ;      | (offset)      ||
                            29     ;      ----------------
                            30     ;      | (segment)     ||
                            31     ;      ----PARM4-------}--->argument FUNC
                            32     ;      | (offset)      ||
                            33     ;      ----------------
                            34     ;      | old DS        |
                            35     ;      ----------------  }saved in prologue
                            36     ;      | old BP        |
                            37     ;      ---------------- <---SP, BP point to here
                            38     ;  low memory
                            39     ;
                            40 +1  $EJECT
                            41     ;The required structure definition is:
                            42     ;
----                        43     DSA      STRUC
                            44
0000                        45     OLD_BP    DW          ?    ;Prologue code saves BP here.
0002                        46     OLD_DS    DW          ?    ;Prologue code saves DS here.
0004                        47     RETURN    DD          ?    ;A double word for FAR procedures.
0008                        48     PARM4     DD          ?    ;Pointer to code of FUNC function.
000C                        49     PARM3     DD          ?    ;Pointer to integer or logical variable.
0010                        50     PARM2     DW          ?    ;A FORTRAN-86 Integer*2 value.
                            51 +1                              ;(parameter passing with $VAL is not recommended)
0012                        52     LEN       DW          ?    ;A FORTRAN-86 Integer*2 variable. (Length)
0014                        53     PARM1     DD          ?    ;  (Pointer to integer or logical value).
                            54
----                        55     DSA      ENDS
                            56
                            57     ;Inside the subprogram, value arguments are accessed simply
                            58     ;by using a structure reference, with BP as the base, and the
                            59     ;appropriate field name as the qualifier; example: [BP].PARM3.
                            60     ;
                            61     ;NOTE: The structure fields for the arguments are declared in
                            62     ;       reverse order in which they were pushed, due to the fact
                            63     ;       that the 8086 stack grows towards low memory.
                            64     ;
                            65     ;The saved value of BP and the return address must be declared
                            66     ;in the structure, since these two items are pushed between the
                            67     ;arguments and the spot pointed to by BP.
                            68     ;
                            69
----                        70     SUBPRG_DATA SEGMENT              ;not combinable
0000 ??                     71     A_LOCAL    DB          ?        ;local variables go here
----                        72     SUBPRG_DATA ENDS
                            73

8086/8087/8088 MACRO ASSEMBLER    ASMEX                                 09/01/80  PAGE    2


LOC  OBJ                   LINE    SOURCE

----                        74     SUBPRG_CODE SEGMENT              ;not combinable
                            75
                            76     ;
                            77     ;SUBPRG does nothing except call the function PARM4 and access
                            78     ;the first three arguments.  The prologue
                            79     ;code saves BP, and then copies SP to BP, allowing the value
                            80     ;arguments to be picked up conveniently with the BP register.
                            81
                            82             PUBLIC      SUBPRG
                            83             ASSUME      CS:SUBPRG_CODE, DS:SUBPRG_DATA
0000                        84     SUBPRG  PROC        FAR
0000 1E                     85             PUSH        DS                ;Prologue code, preserve DS.
0001 55                     86             PUSH        BP                ;Preserve BP for FORTRAN-86.
0002 8BEC                   87             MOV         BP,SP
0004 B8----          R      88             MOV         AX, SUBPRG_DATA   ;Address local data seg.
0007 8ED8                   89             MOV         DS,AX             ;with DS.
                            90
                            91                 ;Call the function argument PARM4. Result is in the register(s).
                            92                 ;1 byte => AL, 2 byte => AX, 4 byte => DX:AX.
                            93
0009 FF5C08                 94             CALL        [SI].PARM4        ;Indirect call to PARM4.
                            95
```

Figure H-2. Sample ASM-86 Program (Cont'd.)

```
000C 8B4E12             MOV     CX,[BP].LEN         ;Length of PARM1 is at BP+12h.
000F C45E14             LES     BX,[BP].PARM1       ;Ptr. to PARM1 is at BP+14h.
0012 268A07             MOV     AL,ES:[BX]          ;First byte of PARM1.
0015 8B5610             MOV     DX,[BP].PARM2       ;PARM2 is at BP+10h.
0018 C45E0C             LES     BX,[BP].PARM3       ;Ptr. to PARM3 is at BP+0Ch.
001B 268A1F             MOV     BL,ES:[BX]          ;Assume PARM1 is 1 byte (or 2 or 4).

001E 5D                 POP     BP
001F 1F                 POP     DS
0020 CA1000             RET     16       ;Return and pop 16 parameter bytes.

                SUBPRG      ENDP
----            SUBPRG_CODE ENDS

                        END

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Figure H-2. Sample ASM-86 Program (Cont'd.)

APPENDIX I
RUN-TIME INTERFACE

This appendix describes the run-time system supporting Fortran-86. It also describes how to run your application object code on your target system and run-time interrupt processing.

I.1 Run-Time Support Overview

The run-time libraries map language-dependent operations into the operating system format. Figure I-1 shows how your application program exists in your system with the run-time libraries.

I.1.1 Application Object Code Independence

In order to allow your application program developed in an Intel operating system environment to run in your 8086-based target system without modification, a Universal Development System Interface (UDI) has been provided. UDI is the specification for handshaking between programs (including run-time libraries) and operating systems. The specification includes calling conventions and data types that are defined as the primitives described in the *Run-Time Support Manual for iAPX 86,88 Applications.*

You must provide a library, using the UDI specification, that sits between the application (including run-time libraries) and the operating system as in Figure I-2.

Figure I-3 shows the Series-III Development System UDI Libraries. Figure I-4 shows the iRMX 86 UDI Libraries.

Note that both the run-time libraries and the application object code may make UDI calls to the Series-III operating system.

When you move your application from one operating system to another, link your application program and run-time libraries to the UDI libraries to support the operating system.

If you provide your own 8086-based operating system, you must write your own UDI library for your operating system.

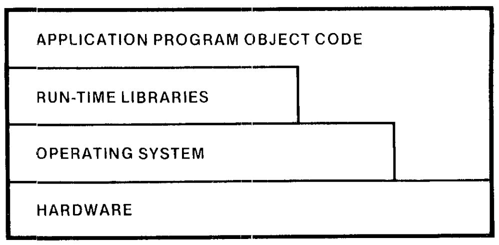


Figure I-1. Application Program and Run-Time Libraries in User System 121570-8

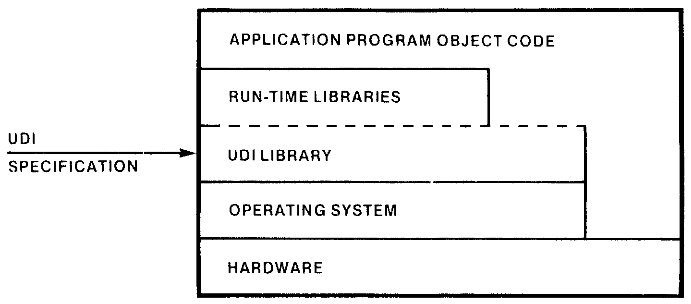


Figure I-2. Use of UDI Library 121570-9

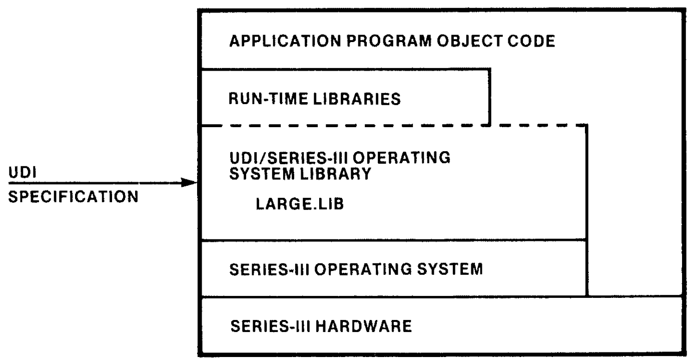

APPLICATION PROGRAM OBJECT CODE
RUN-TIME LIBRARIES
UDI SPECIFICATION
UDI/SERIES-III OPERATING SYSTEM LIBRARY
LARGE.LIB
SERIES-III OPERATING SYSTEM
SERIES-III HARDWARE

Figure I-3. UDI Libraries in Series III Development 121570-10

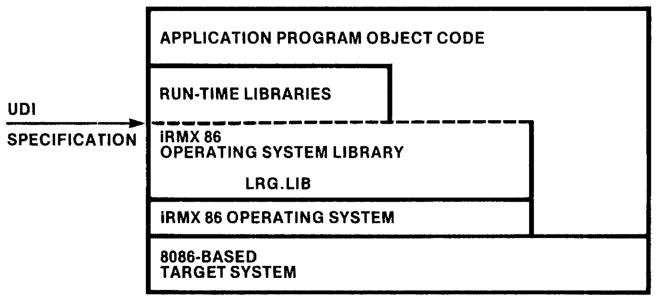


Figure I-4. UDI Libraries with iRMX™ 86 Operating System 121570-11

I.1.2 Low End Application

It is also possible to use a logical record interface instead of UDI for device drivers or simple operating systems as shown later in Figure I-6. (See Section I.3 for details.)

I.2 Run-Time Libraries

There are two types of run-time libraries: I/O and numeric support.

I.2.1 I/O Run-Time Libraries

The Fortran-86 I/O Run-Time Libraries have the format F86RNx.LIB and include:

F86RN0.LIB	
F86RN1.LIB	Formatting and I/O Libraries
F86RN2.LIB	
F86RN3.LIB	Default Logical Record
F86RN4.LIB	System Libraries

I.2.2 Numerics Run-Time Libraries

The numerics libraries support the 8087 (8087.LIB) or the 8087 Emulator (E8087.LIB). Common functions for high-level numerics processing are contained in a separate library, CEL.LIB. In addition, 87ERH.LIB handles 8087 exceptions.

I.2.3 Summary

Figure I-5 shows the run-time libraries and how they interface to the operating system and hardware.

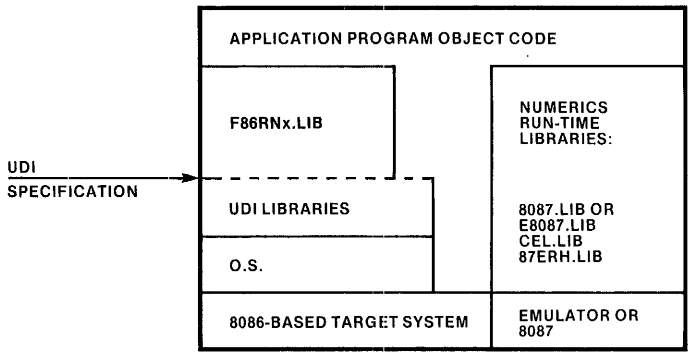

APPLICATION PROGRAM OBJECT CODE
F86RNx.LIB
NUMERICS RUN-TIME LIBRARIES:
UDI SPECIFICATION
UDI LIBRARIES
O.S.
8087.LIB OR E8087.LIB CEL.LIB 87ERH.LIB
8086-BASED TARGET SYSTEM
EMULATOR OR 8087

Figure I-5. I/O and Numerics Run-Time Libraries in System 121570-12

I.3 Logical Record Interface

For information on Logical Record Interface, see the *Run-Time Support for iAPX 86, 88*, 121776.

I.4 Run-Time Interrupt Processing

The discussion in this section does not apply to programs that run in an iRMX 86 environment. To implement run-time interrupt processing on an iRMX 86-based system, your programs must invoke iRMX 86 system calls. Refer to the *iRMX™ 86 Nucleus Reference Manual* for more information.

There are two interrupt pins on the 8086 processor: the "non-maskable interrupt" pin (NMI) and the "maskable interrupt" pin (INTR). The "non-maskable interrupt" cannot be ignored by the processor, whereas the "maskable interrupt" can be enabled or disabled.

Each "maskable interrupt" has an *interrupt number* that designates the type of interrupt. Interrupt numbers range from 0 to 255. Interrupt number 0 is reserved for integer divide by zero errors. Interrupt numbers 1 through 3 are reserved for single stepping, "non-maskable interrupts," and the INT instruction, respectively. Interrupt number 4 is reserved for integer overflow, and integer number 5 is reserved for compiler range checks. The run-time system uses interrupts 16 through 31. Interrupt number 16 is reserved for emulated real arithmetic exceptions, and interrupt number 17 is reserved for other compiler checks. For interrupts reserved for the Series-III system, see the *Intellec® Series III Microcomputer Development System Programmer's Reference Manual.*

You can use any other interrupt numbers for your own procedures. However, if you are overriding the default procedures associated with a specific number, you must use that number for you procedure.

An interrupt occurs when the CPU receives a signal on its "maskable interrupt" pin from some peripheral device. The CPU only responds, however, if interrupts are enabled. The "main program prologue" (code inserted by the compiler at the beginning of the main program) enables interrupts.

If interrupts are enabled, the following actions take place:

1. The CPU issues an "acknowledge interrupt" signal and waits for the interrupting device to send an interrupt number.
2. The CPU flag registers are placed on the stack (occupying two bytes of stack storage).
3. Interrupts are disabled by clearing the IF flag.
4. Single stepping is disabled by clearing the TF flag.
5. The CPU activates the interrupt procedure corresponding to the interrupt number sent by the interrupting device.

You can specify Fortran-86 procedures as interrupt procedures using the INTERRUPT control (11.4.10). You can assign an interrupt number to each interrupt procedure using the SETINT built-in procedure (Chapter 6). These interrupt numbers form an *interrupt vector*, that is, an absolutely-located array of entries beginning at location 0. Thus, the *n*th entry is at location 4 times *n*, and contains the address of the interrupt procedure associated with interrupt number *n*. Each entry is a four-byte value containing a segment address and an offset.

The CPU uses the interrupt vector entry to make a long indirect call to activate the appropriate procedure. At this point, the current code segment address (CS register contents) and instruction offset (IP register contents) are saved on the stack.

If an interrupt procedure terminates normally, the interrupt mechanism and registers are reset to the condition that existed prior to the activation of the procedure.

Figure I-6 shows the stack layout at the point where the procedure is activated.

I.4.1 Interrupt Procedure Preface and Epilogue

At the beginning of each interrupt procedure, before the usual procedure prologue inserted by the compiler, the compiler inserts an *interrupt procedure preface* that performs the following actions:

1. Push the ES register contents onto the stack.
2. Push the DS register contents onto the stack.
3. Load the DS register with a new data segment address taken from the current code segment (i.e., the segment containing the interrupt procedure).
4. Push the AX register contents onto the stack.
5. Push the CX register contents onto the stack.
6. Push the DX register contents onto the stack.
7. Push the BX register contents onto the stack.
8. Push the SI register contents onto the stack.
9. Push the DI register contents onto the stack.

Figure I-7 shows the stack layout at the point where the procedure prologue starts.

10. Perform a call to transfer control to the normal procedure prologue.

Figure I-8 shows the stack layout after the procedure prologue is executed and the code compiled when the interrupt procedure body starts executing.

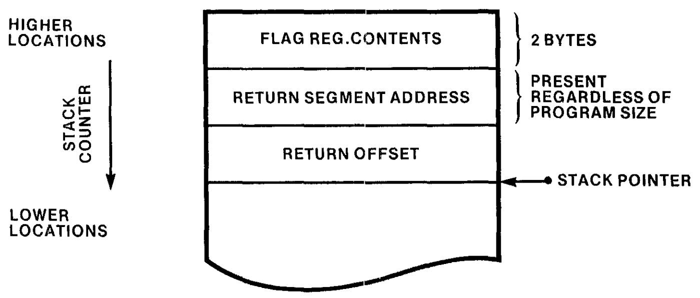


Figure I-6. 8086 Stack Layout When Interrupt Procedure Gains Control 121570-14

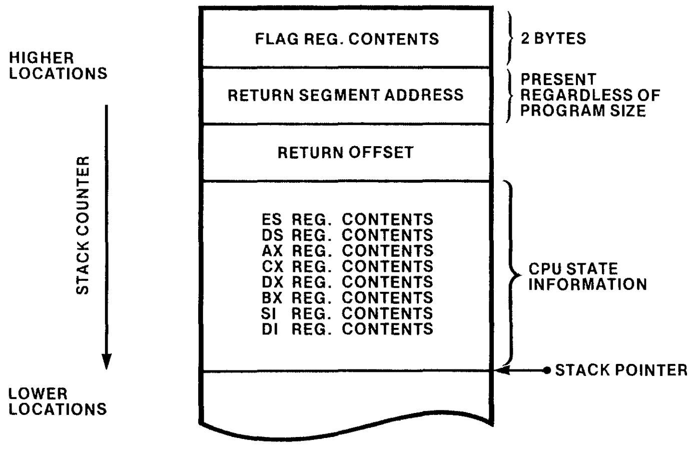


Figure I-7. 8086 Stack Layout After Interrupt Procedure Preface and Before Procedure Prologue 121570-15

When the interrupt procedure body finishes, a RET instruction returns execution to the *interrupt procedure epilogue*, which continues with the following steps.

11. Pop the stack into the DI register.
12. Pop the stack into the SI register.
13. Pop the stack into the BX register.
14. Pop the stack into the DX register.
15. Pop the stack into the CX register.
16. Pop the stack into the AX register.
17. Pop the stack into the DS register.
18. Pop the stack into the ES register.
19. Execute an IRET instruction to return from the interrupt procedure. This restores the IP, CS, and flag register contents from the stack.

At this point the stack is restored to the state it was in before the interrupt occurred, and processing continues normally.

The INTERRUPT compiler control allows you to associate an interrupt number with an interrupt procedure during compile-time. However, you can declare procedures as interrupt procedures without associating them to interrupt numbers creating the interrupt vector at a later time.

Similarly, you can have a library of interrupt procedures that are not yet associated with an interrupt vector. You can then link any program to these procedures with a separately created interrupt vector.

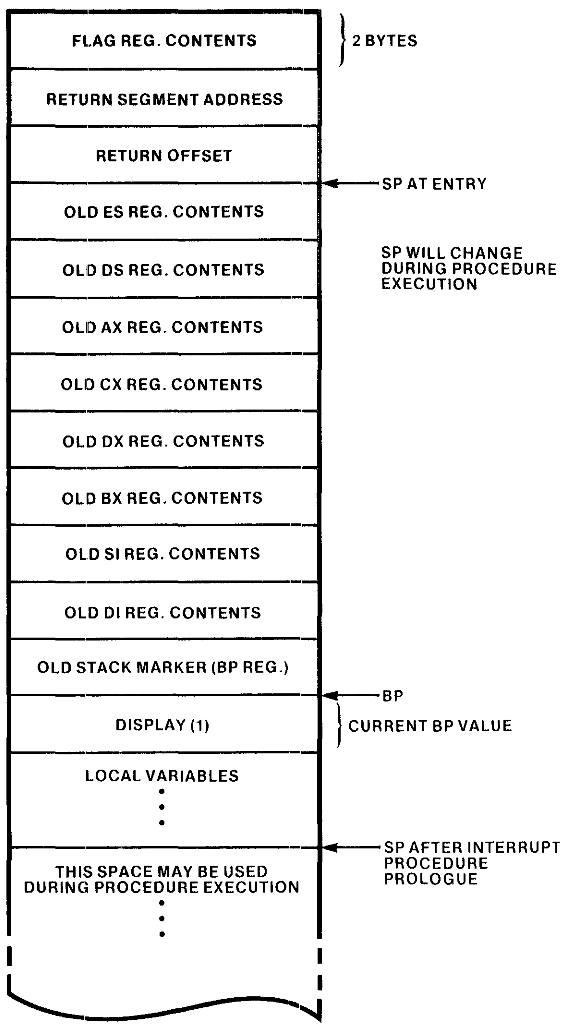


Figure I-8. 8086 Stack Layout During Execution of Interrupt Procedure Body 121570-16

NOTE

An interrupt procedure that uses any of the intrinsic functions EXP, ALOG, SIN, COS, TAN, ARCSIN, ARCCOS, or ARCTAN (functions in the CEL.LIB run-time library) must allocate 50 bytes of 8086 stack space for each level of recursion.

I.4.2 Interrupt Handling for Real Arithmetic Errors

The run-time system (8087 emulator or 8087 processor interface libraries) use interrupt 16 for real arithmetic error handling. If you are using the emulator, you must reserve interrupt 16 for that purpose. If you are using the 8087 processor, you must connect either (1) the 8087 processor to the 8086 interrupt 16, or (2) the 8087 processor to some other interrupt, then link in an assembly language routine to redirect the interrupts from the 8087.

If you are connecting the 8087 processor to an interrupt other than 16, assemble an 8086/8087/8088 assembly language routine like the one given in Figure I-9, and link it in with your program and the interface libraries. The routine in Figure I-9 may be used if the 8087 processor is wired to interrupt 7.

You may modify the routine in Figure I-9 for interrupt *n* (*n* must be greater than or equal to 4) simply by changing the SEGMENT and ORG directives. Calculate the operands for ORG and SEGMENT by first calculating the location of the CONVERT_PROC procedure (4**n*H), and then using the rightmost hexadecimal digit for the ORG operand and the rest of the hexadecimal digits for the SEGMENT operand.

Since the routine in Figure I-9 does not call any external modules, its position in the LINK86 argument list does not matter. Assuming that your compiled Fortran-86 program is called MYMOD1.OBJ, and the assembled routine to redirect interrupts is called INT7.OBJ, you could use the following LINK86 invocation on a Series III development system:

```
-RUN LINK86 MYMOD1.OBJ, INT7.OBJ, CEL.LIB,&<cr>
**F86RN0.LIB, F86RN1.LIB, F86RN2.LIB, F86RN3.LIB,&<cr>
**F86RN4.LIB, EH87.LIB, 8087.LIB, LARGE.LIB<cr>
```

Since the 8087 processor activates the real arithmetic interrupt number when a real arithmetic exception occurs, you can override the default exception handler by providing your own interrupt procedure for the real arithmetic interrupt number. The 8087 exception conditions are described in Chapter 15.

NOTE

The 8087 processor and emulator handle exceptions in the same manner. However, an 8086/8087 implementation may include some external interrupt masking device such as an 8259A. In this case, the emulator cannot simulate the function of the 8259A. When using the 8087 emulator, if an exception that is not masked on the emulated 8087 occurs, and the 8086 interrupt is enabled, a real arithmetic interrupt (interrupt 16) will occur after the emulation of any 8087 instruction. In other words, the 8087 emulator assumes that the 8259A interrupts are enabled.

```
INT_7_SEG           SEGMENT     AT 1H
                    ORG         0CH
                    DD          CONVERT_PROC
INT_7_SEG           ENDS

CONVERT_PROC_S      SEGMENT
CONVERT_PROC        PROC        FAR
                    INT         16
                    IRET
CONVERT_PROC        ENDDP
CONVERT_PROC_S      ENDS
```

Figure I-9. Routine to Redirect Interrupts

APPENDIX J
ADDITIONAL INFORMATION FOR SERIES III AND SERIES IV OPERATING SYSTEM USERS

This appendix contains information that is specific to the Intellec Series III and Series IV Microcomputer Development Systems. It covers the following areas:

- Program development environment
- Compiler invocation and file usage
- Sample link, locate, and execute operations
- Examples of Fortran-86 compiler invocation
- Interrupt handling
- Related publications

This appendix assumes that you have Series III or Series IV system up and running, and that you have a suitable copy of the Fortran-86 compiler. Chapter 1 of this manual leads you through a complete program development sequence using a sample Fortran program supplied with the compiler. Details on the operating system environment are provided in the *Intellec® Series III Microcomputer Development System Console Operating Instructions* (121609), and the *Intellec® Series IV Operating and Programming Guide* (121753).

J.1 Program Development Environment

To run the Fortran-86 Compiler in the Series III system, you must have the following hardware and software:

- Intellec Series III development system
- Intellec Series III operating system (RUN command)
- 192K of RAM memory (standard with the Series III system)
- At least one storage device. (The product is delivered on a flexible disk; therefore the installation of the compiler always requires a single- or double-density disk drive.)

A system with a printer is recommended for producing hard-copy output listings. This system may be separate from the system used to compile programs.

To run the Fortran-86 compiler in the Series IV system, you must have the following hardware and software:

- Intellec Series IV development system
- Intellec Series IV operating system
- 192K of RAM memory (standard with the Series IV system)
- At least one storage device. (The product is delivered on a flexible disk; therefore installation of the compiler always requires a single- or double-density disk drive.)

J.2 Compiler Installation

Compiler installation is described in Chapter 1 of this book.

J.3 Program Disk Contents

The Series III and Series IV Fortran-86 software packages include one double density and one single density disk. Each of these disks contains the following files:

FORT86.86	RTNULL.LIB	E8087	PROG3.FTN
F86RN0.LIB	CEL87.LIB	DCON87.LIB	PROG4.FTN
F86RN1.LIB	EH87.LIB	PROG1A.FTN	PROG5.FTN
F86RN2.LIB	8087.LIB	PROG1B.FTN	
F86RN3.LIB	87NULL.LIB	PROG1C.FTN	
F86RN4.LIB	E8087.LIB	PROG2.FTN	

The file named FORT86 contains the Fortran-86 compiler. The files F86RN0.LIB, F86RN1.LIB, F86RN2.LIB, F86RN3.LIB, F86RN4.LIB, RTNULL.LIB, 8087.LIB, CEL87.LIB, EH87.LIB, and 87NULL.LIB contain the run-time support libraries and modules. DCON87.LIB provides functions that convert floating-point values from binary to ASCII representation, and vice versa. The remaining programs with the extension .FTN are example programs described in Chapter 10 of this manual and Section J.8 of this appendix.

J.4 Compiler Operation

The Fortran-86 compiler is a program that translates your Fortran instructions into object code modules that can be linked and located for execution.

You create a Fortran program by typing instructions into a file using the CREDIT text editor, and submitting the file to the Fortran-86 compiler. The file you submit is called a *source file*, and the file containing the compiled program is called an *object file.* (The content of the object file is also known as *object code.*) In Fortran-86 you can compile *parts* of a program, and each separate compilation is known as an *object module.*

The following discussions assume that you have a Series III or Series IV system up and running, and that you have a suitable copy of the Fortran-86 compiler. Chapter 1 of this manual leads you through a complete program development sequence using a sample Fortran program supplied with the compiler. Details on the operating system environment are provided in the *Intellec® Series III Microcomputer Development System Console Operating Instructions* and the *Intellec® Series IV Operating and Programming Guide.*

J.4.1 Series III Invocation

Invoke the Fortran-86 compiler with the RUN command. The RUN command loads and executes any program specifically in the 8086 environment for the Series III system. The following is a sample compiler invocation:

```
-RUN FORT86 PROG1.SRC XREF<cr>
```

The name FORT86 is the name of the compiler as supplied, without the extension (i.e., the full name is FORT86.86, but you don't supply the .86 extension in the invocation line). PROG1.SRC is the name of the source file that contains the Fortran instructions. XREF is a primary control that tells the compiler to generate a cross-reference listing of source program identifiers (XREF is described in Chapter 11). The XREF control, like all other compiler controls, is optional for the invocation line.

The above example assumes that the compiler and the source program PROG1.SRC reside on drive 0 (:F0:). If PROG1.SRC is on drive 1, the invocation line is:

```
-RUN FORT86 :F1:PROG1.SRC XREF<cr>
```

The *invocation line* takes this general form:

RUN [:F*d*:]FORT86 [:F*d*:] *source* TO [*controls*]

where

RUN	is the name of the command to execute the compiler in the Series III environment.
:F*d*:	specifies which directory FORT86.86 and/or *source* resides in, if not in directory :F0:. The *source* file does not have to be in the same directory as the compiler.
FORT86	is the name you use for the compiler FORT86.86.
source	is the name of the source file containing the Fortran program.
controls	are optional primary or general compiler controls described in Chapter 11. You can have many controls in the invocation line with a space between each control, and you can extend the invocation line by using the ampersand (&) as a continuation character to replace a space.
<cr>	stands for the RETURN key on the keyboard.

The following are some examples:

```
-RUN :F1:FORT86 :F1:MYPROG PRINT(:LP:) TITLE(:TEST24:)<cr>
```

In this example, both FORT86.86 and MYPROG are on drive 1. PRINT and TITLE are compiler controls.

```
-RUN FORT86 :F1:KLUDGE.SRC NOPRINT<cr>
```

In this example, FORT86.86 is on drive 0, but KLUDGE.SRC, the source program, is on drive 1. NOPRINT is a compiler control that prevents all printed output (except error messages) usually generated by the compiler.

NOTE

The RUN command assigns the extension 86 to the filename it executes, if it is specified without an extension. You must specify the filename's extension if it is not 86. If you specify a filename that has no extension, specify a period (.) after the name in the RUN invocation line. For example, if you rename FORT86.86 to COMPIL, include a period after the name COMPIL (i.e., COMPIL.) when you invoke it using RUN. If you choose a new name with a new extension, specify both the new name and the new extension on the RUN invocation line.

J.4.2 Series IV Invocation

The following is a sample compiler invocation in the Series IV environment:

```
-FORT86 PROG1.SRC XREF<cr>
```

The name FORT86 is the name of the compiler as supplied, without the extension (i.e., the full name is FORT86.86, but you don't supply the .86 extension in the invocation line). PROG1.SRC is the name of the source file that contains the Fortran instructions. XREF is a primary control that tells the compiler to generate a cross-reference listing of source program identifiers (XREF is described in Chapter 11). The XREF control, like all other compiler controls, is optional for the invocation line.

The above example assumes that the compiler and the source program PROG1.SRC reside on drive 0 (:F0:). If PROG1.SRC is on drive 1, the invocation line is:

```
-FORT86 :F1:PROG1.SRC XREF<cr>
```

The invocation line takes this general form:

```
[:Fd:] FORT 86 [:FD:] SOURCE TO [CONTROLS]
```

where

:F*d*:	specifies which directory FORT86.86 and/or *source* resides in, if not in directory :F0:. The *source* file does not have to be in the same directory as the compiler.
FORT86	is the name you use for the compiler FORT86.86.
source	is the name of the source file containing the Fortran program.
controls	are optional primary or general compiler controls described in Chapter 11. You can have many controls in the invocation line with a space between each control, and you can extend the invocation line by using the ampersand (&) as a continuation character to replace a space.
< cr >	stands for the RETURN key on the keyboard.

The following are some examples:

```
-:F1:FORT86 :F1:MYPROG PRINT(:LP:) TITLE(:TEST24:)<cr>
```

In this example, both FORT86.86 and MYPROG are on drive 1. PRINT and TITLE are compiler controls.

```
-FORT86 :F1:KLUDGE.SRC NOPRINT<cr>
```

In this example, FORT86.86 is on drive 0, but KLUDGE.SRC, the source program, is on drive 1. NOPRINT is a compiler control that prevents all printed output (except error messages) usually generated by the compiler.

J.4.3 Files Used by the Compiler

J.4.3.1 Input Files

You supply the Fortran source program name for *source* in the invocation line (see the previous section). You can also include other source files by using the INCLUDE control, as described in Chapter 11. These files must be standard ISIS files containing text of Fortran instructions.

J.4.3.2 Output Files

By default, the compiler produces two output files, unless you use specific controls to suppress or redirect them: the *listing* file and the *object* file. Also by default, error messages appear in the listing file.

The listing file (sometimes called the PRINT file) contains a listing of the source program, plus any other printed output generated by the compiler as specified by the listing selection controls described in Chapter 11. The object file (sometimes called the object code file or object module) contains the actual code in object module format, which can eventually be executed (after you use the linking and locating facilities described in Chapter 14. These files are described in more detail in Chapter 13.

The listing file and the object file have the same name as the source file, except that the listing file has the extension LST, and the object file has the extension OBJ. The files are created if they do not exist, or overwritten if they do exist, and they appear in the same directory as the source file. You can optionally change the names and/or directories for the listing and object files by using the PRINT and OBJECT controls, respectively (described in Chapter 11).

For example, if you invoke the compiler on a Series III using the line—

```
-RUN FORT86 :F1:MYPROG<cr>
```

or on a Series IV using the line—

```
FORT86 :F1:MYPROG<cr>
```

the compiler creates (or overwrites) the file MYPROG.LST in directory 1 to contain the listing, and the file MYPROG.OBJ in directory 1 to contain the object module.

You can optionally direct certain sections of printed output to files other than the default listing file described above. In addition to using the PRINT control to specify another file as the listing file, you can specify a different file to receive error messages by using the ERRORPRINT control. Chapter 11 gives details on the use of these controls.

J.4.3.3 Work Files

When operating under the Series III operating system, the compiler creates and uses *work files* during its operation, and deletes them at the completion of compilation. These files are designated :WORK: files and they cannot conflict with your files.

The Series III operating system provides a mechanism to select the directory where work files can be temporarily stored. The default directory is directory 1 (:F1:), but you can select another directory using the RUN WORK command, as in this example:

```
-RUN WORK :F0:<cr>
```

This example selects directory :F0: as the directory to hold work files.

J.4.4 Compiler Messages

When you invoke the compiler, it displays the sign-on message:

```
SERIES-III FORTRAN-86 COMPILER, Vx.y
```

where

x	is the version number of the compiler.
y	is the change number within the version.

When a compilation is finished, the compiler terminates with the message:

```
m TOTAL ERRORS DETECTED
n TOTAL WARNINGS DETECTED
```

where

m	is the total number of errors detected.
n	is the total number of warnings detected.

J.4.4.1 Insufficient Memory Error Messages

The compiler issues a warning message when the compiler dictionary overflows onto external memory. Along with this warning, the compiler indicates the point where the overflow occurred:

```
DICTIONARY OVERFLOW ONTO WORK FILE WHILE PROCESSING SYMBOL symbol
```

Additional information will appear in the PRINT file.

J.5 Linking, Locating, and Executing

The linker (LINK86) links object modules and outputs a file. The locator (LOC86) assigns absolute addresses to modules to locate them in actual memory. The loader (RUN) loads and executes the final program. Additionally, the LIB86 utility enables you to create and maintain your own library file of compiled (or translated) object modules for use with other programs.

The following is a list of the software provided for building executable Fortran-86 programs with a Series III development system:

FORT86.86—the Fortran-86 compiler

F86RN0.LIB, F86RN1.LIB, F86RN2.LIB, F86RN3.LIB, F86RN4.LIB, and RTNULL.LIB—the run-time support libraries

LARGE.LIB—the Series III operating system interface library

CEL87.LIB—the floating-point intrinsic function library

EH87.LIB—the floating-point exception handler library

8087.LIB—the 8087 numeric processor extension (NPX) interface library

E8087, and the E8087.LIB — the 8087 Emulator and interface library

87NULL.LIB — the support library that resolves floating-point references if no floating point arithmetic is used

LINK86, CREF86.86, LOC86, LIB86, and OH86.86 — the 8086-based utilities

J.5.1 Series III Sample Link Operations

The following link operation takes two object modules, MYMOD1.OBJ and MYMOD2.OBJ, links them together, then links in the Fortran run-time libraries to form the output module MYPROG.86. To extend the LINK86 command to the next line without transmitting the command, type the ampersand (&) character before the RETURN key, and continue typing the command on the next line (do not type the ampersand character between letters of a filename). The continued line will start with an angle bracket (>).

```
-RUN LINK86 MYMOD1.OBJ, MYMOD2.OBJ, F86RN0.LIB, & <cr>
> F86RN1.LIB, F86RN2.LIB, F86RN3.LIB, F86RN4.LIB, & <cr>
> 87NULL.LIB, LARGE.LIB TO MYPROG.86 BIND <cr>
```

The linker first reads MYMOD1.OBJ and MYMOD2.OBJ for external references and resolves those references. Then, the linker attempts to resolve any more external references in the modules by looking at the public symbols in the libraries F86RN0.LIB, F86RN1.LIB, F86RN2.LIB, F86RN3.LIB, F86RN4.LIB, 87NULL.LIB, and LARGE.LIB. Use the 87NULL.LIB when the modules do not perform real arithmetic. The final output module is MYPROG.86. This module can be loaded and executed on the Series III.

When the modules MYMOD1.OBJ and MYMOD2.OBJ do perform real arithmetic, link them with the 8087 Numeric Data Processor or the 8087 Emulator. The LINK86 command when using the emulator is:

```
-RUN LINK86 MYMOD1.OBJ, MYMOD2.OBJ, F86RN0.LIB, & <cr>
> F86RN1.LIB, F86RN2.LIB, F86RN3.LIB, F86RN4.LIB, & <cr>
> EH87.LIB, E8087, E8087.LIB, LARGE.LIB TO MYPROG.86 BIND <cr>
```

To support real arithmetic when using the 8087, replace E8087 and E8087.LIB with 8087.LIB. EH87.LIB provides exceptions handling support for the 8087 Numeric Data Processor or its emulator. This is the link sequence that should be used in a full-featured operating system. The BIND option used with LINK86 provides an output file that is ready to be executed (if the operating system has an LTL loader).

J.5.2 Series IV Sample Link Operations

The following link operation takes two object modules, MYMOD1.OBJ and MYMOD2.OBJ, links them together, then links in the Fortran run-time libraries to form the output module MYPROG.86. To extend the LINK86 command to the next line without transmitting the command, type the ampersand (&) character before the RETURN key, and continue typing the command on the next line (do not type the ampersand character between letters of a filename). The continued line will start with an angle bracket (>).

```
-LINK86 MYMOD1.OBJ, MYMOD2.OBJ, F86RN0.LIB, & <cr>
> F86RN1.LIB, F86RN2.LIB, F86RN3.LIB, F86RN4.LIB, & <cr>
> 87NULL.LIB, LARGE.LIB TO MYPROG.86 BIND <cr>
```

The linker first reads MYMOD1.OBJ and MYMOD2.OBJ for external references and resolves those references. Then, the linker attempts to resolve any more external references in the modules by looking at the public symbols in the libraries F86RN0.LIB, F86RN1.LIB, F86RN2.LIB, F86RN3.LIB, F86RN4.LIB, 87NULL.LIB, and LARGE.LIB. Use the 87NULL.LIB when the modules do not perform real arithmetic. The final output module is MYPROG.86. This module can be loaded and executed on the Series III.

When the modules MYMOD1.OBJ and MYMOD2.OBJ do perform real arithmetic, link them with the 8087 Numeric Data Processor or the 8087 Emulator. The LINK86 command when using the emulator is:

```
-LINK86 MYMOD1.OBJ, MYMOD2.OBJ, F86RN0.LIB, & <cr>
>  F86RN1.LIB, F86RN2.LIB, F86RN3.LIB, F86RN4.LIB, & <cr>
>  EH87.LIB, E8087, E8087.LIB, LARGE.LIB TO MYPROG.86 BIND <cr>
```

To support real arithmetic when using the 8087, replace E8087 and E8087.LIB with 8087.LIB. EH87.LIB provides exceptions handling support for the 8087 Numeric Data Processor or its emulator. This is the link sequence that should be used in a full-featured operating system. The BIND option used with LINK86 provides an output file that is ready to be executed (if the operating system has an LTL loader).

J.5.3 Examples

The following examples show how to execute Fortran-86 programs in different environments.

1. To execute a Fortran-86 program in a bare machine (or minimal operating system) environment link in the run-time libraries F86RN0.LIB, F86RN1.LIB, and F86RN2.LIB. If the program requires numerics support and you are using the 8087 microprocessor, the link command is

   ```
   -LINK86 MYPROG.OBJ, CEL87.LIB, F86RN0.LIB, F86RN1.LIB, & <cr>
   >  F86RN2.LIB, RTNULL.LIB, 8087.LIB & <cr>
   >  TO MYPROG.86 BIND <cr>
   ```

   In this example, string and 32-bit integer operations are fully supported. Fortran input/output is not supported; if used, LINK86 will generate an UNRESOLVED EXTERNALS warning.

   When an 8087 exception occurs during program execution, RTNULL.LIB halts execution without an error message. Since there are no external references between RTNULL.LIB and EH87.LIB, the 8087 exception handler will never be invoked. Consequently, do not use RTNULL.LIB when 8087 exceptions are expected.

   Note that the BIND option was not used. In this environment the programs will usually be located using LOC86 and burned into ROM, or loaded with a simple absolute loader.

2. This example links a program using only internal I/O.

   Series III:

   ```
   -RUN LINK86 MYPROG.OBJ, F86RN0.LIB, F86RN1.LIB & <cr>
   >  F86RN2.LIB, RTNULL.LIB, 87NULL.LIB, & <cr>
   >  TO MYPROG.86 <cr>
   ```

   Series IV:

   ```
   -LINK86 MYPROG.OBJ, F86RN0.LIB, F86RN1.LIB & <cr>
   >  F86RN2.LIB, RTNULL.LIB, 87NULL.LIB, & <cr>
   >  TO MYPROG.86 <cr>
   ```

3. This example links a program that does internal I/O and floating-point arithmetic with 8087 emulator support.

   Series III:

   ```
   -RUN LINK86 MYPROG.OBJ, CEL87.LIB, F86RN0.LIB, F86RN1.LIB, & <cr>
   >  F86RN2.LIB, RTNULL.LIB, EH87.LIB, E8087, E8087.LIB & <cr>
   >  TO MYPROG.86 <cr>
   ```

Series IV:

```
-LINK86 MYPROG.OBJ, CEL87.LIB, F86RN0.LIB, F86RN1.LIB, & <cr>
> F86RN2.LIB, RTNULL.LIB, EH87.LIB, E8087, E8087.LIB & <cr>
> TO MYPROG.86 <cr>
```

J.5.4 Sample Locate Operations

The following is a sample locate operation using the default settings for controls.

Series III:

```
-RUN LOC86 SAMPL.1.LNK<cr>
```

Series IV:

```
-LOC86 SAMPL.1.LNK<cr>
```

This sample locate operation binds the logical segments of SAMPL1.LNK to addresses beginning at 00200H (H is for hexadecimal), the default. The output module is called SAMPL1 (the root name of the input module without the LNK extension). Unless you specify a TO clause, the output module (the absolutely located program) will always have the same root name as the input module.

The following is a sample locate operation using the ORDER and ADDRESSES controls.

Series III:

```
-RUN LOC86 SAMPL2.LNK &<cr>
**ORDER(CLASSES(CODE,STACK,DATA)) &<cr>
**ADDRESSES(CLASSES(CODE(20000H),STACK(4F000H)))<cr>
```

Series IV:

```
-LOC86 SAMPL2.LNK &<cr>
**ORDER(CLASSES(CODE,STACK,DATA)) &<cr>
**ADDRESSES(CLASSES(CODE(20000H),STACK(4F000H)))<cr>
```

In the invocation line, you can use the ampersand character (&) to continue a long line without executing it.

This sample locate operation collected together the logical segments by class names in the order specified in the ORDER control. The locater then assigned addresses as specified in the ADDRESSES control to the logical segments collected into the CODE and STACK classes. The DATA class received its address assignment from the default algorithm.

J.5.5 Executing Programs on a Series III

The output module from the locater can be loaded and executed in the 8086 environment by using the Series III RUN command. Position-independent (PIC) and loadtime locatable (LTL) modules produced by LINK86 with the BIND option can also be loaded and executed by the RUN command. These modules could also be used as input to the DEBUG-86 debugger or a similar debugging tool.

To run correctly, a program must be complete; i.e., it must contain all modules necessary to run. For example, in order to run in the Series III 8086 environment with run-time support, a program must contain modules from the run-time support libraries. To run in a foreign environment, you must supply your own run-time support and follow the guidelines in Chapter 14 and Appendixes H and K.

To run a complete program in the Series III 8087 environment, simply use the RUN command. In the example below, both the RUN program and the SAMPL1 program are in directory :F0:. To refer to any program in a different directory, specify the directory in the format :F*d*:.

```
-RUN SAMPL1.<cr>
```

Note that in the example, SAMPL1 appears with a period at the end. This period tells the RUN command not to look for an .86 extension. If the program were named "SAMPL1.86", you would not put a period at the end:

```
-RUN SAMPL1<cr>
```

If your program's name has an extension other than .86, you must specify the extension with the name. If its name has an .86 extension, you need not specify it. If its name has no extension, you must specify the final period.

NOTE

If you use the BIND option with LINK86 on a module that is ready to be processed by the RUN loader, and you do not specify its name in a TO clause, the linker will use the root name (and device) of the first file specified as input, but will not append the LNK extension.

J.5.6 Executing Programs on a Series IV

Output modules from the locater as well as position-independent code (PIC) and loadtime locatable (LTL) modules produced by LINK86 with the BIND option can be loaded and executed in the Series IV environment.

To run correctly, a program must be complete, i.e., it must contain all modules necessary to run. For example, to run in the Series IV 8080 environment with run-time support, a program must contain modules from the run-time support libraries.

To run in a foreign environment, you must supply your own run-time support by following the instructions in Chapter 14 and Appendixes H and K.

To run a complete program in a Series IV 8087 environment, enter the name of the program on the command line as shown:

```
-SAMPL1.<cr>
```

The period at the end of the program name indicates that the program does not have an 86 extension.

If the program has an extension other than .86, specify the extension by name.

NOTE

When the BIND option is used with LINK86 on a module that is ready to be processed, and the name of the program is not specified in a TO clause, the linker uses the root name (and device) of the first file specified as input, but does not append the LNK extension.

J.7 Specific Compiler Controls

This appendix includes a fold-out page for system-specific examples of most of the Fortran-86 compiler controls. This page is designed to be opened out and used in conjunction with the corresponding text in Chapter 10.

J.8 Interrupt Handling on the Series III and Series IV

The Intellec Series III maps the eight Multibus interrupt lines (INT0 through INT7) onto interrupt vector entries numbered 56 through 63; therefore, your application may not use these for software interrupts. Interrupt vector entries available for user software include 64 through 183. Refer to the *Intellec® Series III Microcomputer Development System Programmer's Reference Manual* or the *Intellec® Series IV Operating and Programming Guide* for details.

J.9 Related Publications

Below is a list of other Intel publications you are likely to need to use Fortran-86. Most of them describe related Intel products. The manual order number for each publication is given immediately following the title.

For a list of non-Intel publications that may be useful to you, see the Bibliography at the end of this manual.

- *Fortran-86 Pocket Reference*, 121571

  A companion to this manual, providing summary information for quick reference.

- *A Guide to the Intellec® Series III Microcomputer Development System*, 121632

  A guide to the use of the Series III and associated tools as a total development solution for your iAPX 86 and iAPX 88 microcomputer applications. This tutorial manual takes you through hands-on sessions with the Series III operating system, the CREDIT text editor, the Fortran-86 compiler, the iAPX 86, 88 Family Utilities, the DEBUG-86 applications debugger, and the ICE-86A In-Circuit Emulator.

- *Intellec® Series III Microcomputer Development System Product Overview*, 121575

  A summary description of the set of manuals that describe the Intellec Series III development system and its supporting hardware and software. This brief manual includes a description of each manual related to the Series III, plus a glossary of terms used in the manuals.

- *Intellec® Series III Microcomputer Development System Console Operating Instructions*, 121609
  Intellec® Series III Microcomputer Development System Pocket Reference, 121610

  Instructions for using the console features of the Series III, including the DEBUG-86 applications debugger. The *Console Operating Instructions* provides complete instructions, and the *Pocket Reference* gives a summary of this information.

- *Intellec® Series III Microcomputer Development System Programmer's Reference Manual*, 121618

  Instructions for calling system routines from user programs for both microprocessor environments, MCS-80/85 and iAPX 86, in the Series III.

- *Intellec® Series IV Microcomputer Development System Overview*, 121752

- *Intellec® Series IV Operating and Programming Guide*, 121753
- *ISIS-II CREDIT™ CRT-Based Text Editor User's Guide*, 9800902
  CREDIT™ CRT-Based Text Editor Pocket Reference, 9800903

  Instructions for using CREDIT, the CRT-based text editor supplied with the Series III. The User's Guide provides complete operating instructions, and the Pocket Reference summarizes this information for quick reference.
- *AEDIT-80 Text Editor User's Guide*, 121756 Instructions for using AEDIT-80.
- *iAPX 86,88 Family Utilities User's Guide*, 121616
  iAPX 86,88 Family Utilities Pocket Reference, 121669

  Instructions for using the 8087-based utility programs LINK86, LIB86, LOC86, CREF86, and OH86 in 8086-based development environments to prepare compiled or assembled programs for execution. The *User's Guide* provides complete operating instructions, and the *Pocket Reference* summarizes this information for quick reference.
- *ASM86 Language Reference Manual*, 121703
  ASM86 Macro Assembler Operating Instructions, 121628
  ASM86 Macro Assembler Pocket Reference, 121674

  Instructions for using the ASM86 in 8086-based development environments. The *Language Reference Manual* gives a complete description of the assembly language; the *Operating Instructions* gives complete instructions for operating the assembler; and the *Pocket Reference* provides summary information for quick reference. You need these publications if you are coding some of your routines in assembly language:

  PL/M-86 User's Guide, 121636
  PL/M-86 Pocket Reference, 121662
  Pascal-86 User's Guide, 121540
  Pascal-86 Pocket Reference, 121541

  Instructions for using the PL/M and Pascal-86 languages and compilers in iAPX 86-based development environments. The *User's Guide* gives a complete description of the language and compiler (or translator), and the *Pocket Reference* provides summary information for quick reference. You need these publications if you are coding some of your programs in PL/M-86 or Pascal-86:
- *PSCOPE High-Level Program Debugger User's Guide*, 121790

  Instructions for using PSCOPE, the symbolic debugger for high-level language programs. The *User's Guide* provides complete operating instructions.

  ICE™-86A In-Circuit Emulator Operating Instructions for ISIS-II Users, 9800714
  ICE™-86A Pocket Reference, 9800838
  ICE™-88 In-Circuit Emulator Operating Instructions for ISIS-II Users, 9800949
  ICE™-88 Pocket Reference, 9800950

  Instructions for using the ICE-86A and ICE-88 In-Circuit Emulators for hardware and software development. The *Operating Instructions* manuals give complete user descriptions of the In-Circuit Emulators, and the *Pocket Reference* guides provide summary information for quick reference. You need the corresponding publications if you are using the ICE-86A or ICE-88 emulator.
- *The iAPX 86,88 User's Manual*, 210201-001

  This manual contains general reference information, application notes, and data sheets describing the 8086, 8087, 8088, and 8089 microprocessors and their use.

  Extensive discussions of hardware and development software (including PL/M-86, assembly language, LINK86, and LOC86), plus numerous examples of system designs and programs, are included.

- *8087 Support Library Reference Manual*, 121725

  This manual contains specific information on the 8087 support libraries that are available. It includes full descriptions of the DCON87.LIB, CEL87.LIB, and EH87.LIB, as well as a discussion of the IEEE math standard.

- *Run-Time Support Manual for iAPX 86,88 Applications*, 121776

  This manual describes in detail the run-time interface needed to run programs on the iAPX 86,88 family of microprocessors. It includes a description of the run-time libraries required by high-level language compilers, the concepts behind Intel's various operating system environments, the specifications for Intel's Universal Development Interface (UDI), and the definition of the Logical Record Interface (LRI).

10.1 Examples

Example 10.1.1 Program 1A (PROG1A.FTN)

Link the program to the libraries 87NULL.LIB and LARGE.LIB.

Example 10.1.2 Program 1B (PROG1B.FTN)

Link the program to the libraries 87NULL.LIB and LARGE.LIB.

Example 10.1.3 Program 1C (PROG1C.FTN)

Link the program to the libraries 87NULL.LIB and LARGE.LIB.

10.2 TEMPREAL Example

Example 10.2.1 Program 2 (PROG2.FTN)

Link the program to the libraries CEL87.LIB, EH87.LIB, LARGE.LIB, and either 8087.LIB or E8087 and E8087.LIB.

10.3 $INTERRUPT Example

Example 10.3.1 Program 3 (PROG3.FTN)

Link the program to the libraries 87NULL.LIB and LARGE.LIB.

10.4 $REENTRANT Example

Example 10.4.1 Program 4 (PROG4.FTN)

Link the program to the libraries 87NULL.LIB and LARGE.LIB.

10.5 Function Subprogram Example

Example 10.5.1 Program 5 (PROG5.FTN)

Link the program to the libraries CEL87.LIB, EH87.LIB, LARGE.LIB, and either 8087.LIB or E8087 and E8087.LIB.

APPENDIX K ADDITIONAL INFORMATION FOR iRMX™ 86 OPERATING SYSTEM USERS

This appendix contains information that is specific to the iRMX 86 Operating System. It covers the following areas:

- Program development environment
- Compiler invocation and file usage
- Sample link, locate, and execute operations
- Examples of Fortran-86 compiler invocation with an iRMX 86-based system
- Related publications

This appendix assumes that you have an iRMX 86-based system up and running, and that you have a suitable copy of the Fortran-86 compiler. Chapter 1 of this manual leads you through a complete program development sequence using a sample Fortran program supplied with the compiler. Details on the operating system environment are provided in the *iRMX™ 86 Human Interface Reference Manual.*

K.1 Program Development Environment

To run the Fortran-86 compiler in the iRMX 86-based system, you must have the following hardware and software:

- The iRMX 86 Human Interface (and other iRMX 86 layers necessary to support the Human Interface)
- At least 153K of free space (RAM memory over the operating system requirements)
- At least one mass storage device. (The product is delivered on a flexible disk; therefore, the installation of the compiler always requires a single- or double-density disk drive.)

A system with a printer is recommended for producing hard-copy output listings. This system may be separate from the system used to compile programs.

K.2 Compiler Installation

Compiler installation is described in Chapter 1 of this book.

K.3 Program Disk Contents

The iRMX 86 Fortran-86 software package includes one double density and one single density disk. Each of these disks contains the following files:

FORT86	RTNULL.LIB	E8087	PROG3.FTN
F86RN0.LIB	CEL87.LIB	DCON87.LIB	PROG4.FTN
F86RN1.LIB	EH87.LIB	PROG1A.FTN	PROG5.FTN
F86RN2.LIB	8087.LIB	PROG1B.FTN	
F86RN3.LIB	87NULL.LIB	PROG1C.FTN	
F86RN4.LIB	E8087.LIB	PROG2.FTN	

The file named FORT86 contains the Fortran-86 compiler. The files F86RN0.LIB, F86RN1.LIB, F86RN2.LIB, F86RN3.LIB, F86RN4.LIB, RTNULL.LIB, 8087.LIB, CEL87.LIB, EH87.LIB, and 87NULL.LIB contain the run-time support libraries and modules. DCON87.LIB provides functions that convert floating-point values from binary to ASCII representation, and vice versa. The remaining programs with the extension .FTN are example programs described in Chapter 10 of this manual and Section K.8 of this appendix.

K.4 Compiler Operation

The Fortran-86 compiler is a program that translates your Fortran instructions into object modules that can be linked and located for execution.

To create a Fortran program, type the instructions into a file using a text editor, and submit the file to the Fortran-86 compiler. The original file is called a source file, and the file containing the compiled program is called an object file. (The content of the object file is also known as object code.) In Fortran-86 you can compile parts of a program: each separate compilation is known as an object module.

K.4.1 Invoking the Compiler on an iRMX™ 86-Based System

The command line to invoke the Fortran-86 compiler on an iRMX 86-based system is

```
-dir FORT86 sourcepath controls <cr>
```

where

-	is the prompt.
dir	is the pathname of the directory that contains the compiler.
FORT86	is the name of the compiler as supplied by Intel.
sourcepath	is the pathname of the file containing the Fortran-86 source module.
controls	are optional primary or general compiler controls described in Chapter 11. When using more than one control in the invocation line, use a space between each control.
(&)	acts as a continuation character that replaces a space.
<cr>	represents the RETURN key on the keyboard.

The following is a sample invocation:

```
-FORT86 PROGRM.FTN SYMBOLS <cr>
```

where

PROGRM.FTN	is the name of the source file that contains the Fortran source program.
SYMBOLS	is a compiler control that tells the compiler to generate a symbol-table listing of source-program identifiers in addition to the object module and listing file.

The preceding sample invocation line assumes that both the compiler and the source program reside in the default directory (:$:). You can specify different devices and different directories, however, by prefixing the compiler name and the source file name with additional pathname components.

In the following example, the compiler resides on a device whose logical name is :FD1:, and the source file resides on the default device in a subdirectory of the :PROG: directory.

```
-:FD1:FORT86 :PROG:FTNPROGS/PROGRM.F86 SYMBOLS <cr>
```

Refer to the *iRMX™ 86 Human Interface Reference Manual* for more information about the iRMX 86 file naming conventions.

K.4.2 Files Used by the Compiler

The compiler uses three kinds of files: input files, output files and work files.

K.4.2.1 Input Files

You supply the Fortran source program name for the source in the invocation line previously listed. To include other source files uses the INCLUDE control, as described in Chapter 11. These files must be standard files containing the text of Fortran instructions.

K.4.2.2 Output Files

Unless specific controls are used to suppress the files, the compiler produces two output files: the object file and the listing file.

The object file contains the actual code in object module format. The system can execute the object file after the linking and locating operations are completed (see Chapter 14).

The listing file, or PRINT file, contains a listing of the source program and any other printed output generated by the compiler. (The listing selection controls are described in Chapter 11.)

The listing file and the object file unless changed by the PRINT or OBJECT controls have the same file name as the source file, but with a different extension. The listing file has the extension LST and the object file has the extension OBJ.

If the files do not exist, the compiler creates the files—*flname*.LST and *flname*.OBJ. If files with these names do exist and they are in the same directory as the source file, the compiler overwrites them.

For example, if you invoke the compiler on an iRMX 86-based system with the command

```
-FORT86 :PROG:FTNPROGS/PROGRM <cr>
```

the compiler places the listing in a file with the pathname :PROG:FTNPROGS/ PROGRM.LST. It places the object module in a file with pathname :PROG:FTNPROGS/PROGRM.OBJ.

The compiler output files are described in greater detail in Chapter 13.

K.4.2.3 Work Files

The compiler creates and uses work files during its operation and deletes them upon the completion of compilation.

During configuration of the iRMX 86 Operating System, you can select a location for compiler work files. To do this, assign the logical name :WORK: to a device or to a directory on a device. The compiler automatically creates its work files within the :WORK: directory.

The :WORK: directory is the default in iRMX 86-based systems.

See Chapter 13, "Compiler Output," for more information.

K.4.3 Compiler Messages

The sign-on message for the Fortran-86 compiler is

```
iRMX 86 FORTRAN COMPILER, Vx.y
```

where

x	is the version number of the compiler.
y	is the change number within the version.

When a compilation is finished, the compiler terminates with the message

```
m TOTAL ERRORS DETECTED

n TOTAL WARNINGS DETECTED
```

where

m	is the total number of errors detected.
n	is the total number of warnings detected.

Other iRMX 86 error messages can be found in the *iRMX™ 86 Human Interface Reference Manual.*

K.5 Linking, Locating, and Executing in an iRMX™ 86-Based Environment

The linker (LINK86) links object modules and outputs a file. The locator (LOC86) assigns absolute addresses to modules to locate them in actual memory. The operating system loads and executes the final program. Additionally, the LIB86 utility enables you to create and maintain your own library file of compiled (or translated) object modules for use with other programs.

A list of the software provided for building executable Fortran-86 programs follows:

FORT86—the Fortran-86 compiler

F86RN0.LIB, F86RN1.LIB, F86RN2.LIB, F86RN3.LIB, F86RN4.LIB, and RTNULL.LIB—the run-time support libraries

CEL87.LIB—the floating-point intrinsic function library

EH87.LIB—the floating-point error handler

8087.LIB—the 8087 numeric processor extension (NPX) interface library

87NULL.LIB—the support library that resolves references if no 8087 processor is used

LRG.LIB—the Universal Development Interface (UDI) library

LINK86, CREF86, LOC86, LIB86, and OH86—the 8086-based utilities

K.5.1 Sample Link Operations

The following link operation takes two object modules, MYMOD1.OBJ and MYMOD2.OBJ, links them together, then links in the Fortran run-time libraries to form the output module MYPROG.86. To extend the LINK86 command to the next line without transmitting the command, type the ampersand (&) continuation character before the RETURN key, and continue typing the command on the next line. The continued line will start with two asterisks (**).

```
-LINK86 MYMOD1.OBJ, MYMOD2.OBJ, F86RN0.LIB, & <cr>
**F86RN1.LIB, F86RN2.LIB, F86RN3.LIB, F86RN4.LIB, & <cr>
**87NULL.LIB, LRG.LIB TO MYPROG.86 BIND <cr>
```

The linker first reads MYMOD1.OBJ and MYMOD2.OBJ for external references and resolves those references. Then the linker attempts to resolve any other external references in the modules by looking at the public symbols in the libraries F86RN0.LIB, F86RN1.LIB, F86RN2.LIB, F86RN3.LIB, F86RN4.LIB, 87NULL.LIB, and LRG.LIB. Use the 87NULL.LIB when the modules do not perform real arithmetic. The final output module is MYPROG.86. This module can be loaded and executed in the iRMX 86 environment.

When the modules MYMOD1.OBJ and MYMOD2.OBJ do perform real arithmetic, link them with the 8087 Numeric Data Processor libraries. The LINK86 command is

```
-LINK86 MYMOD1.OBJ, MYMOD2.OBJ, CEL87.LIB, F86RN0.LIB, & <cr>
**F86RN1.LIB, F86RN2.LIB, F86RN3.LIB, F86RN4.LIB, & <cr>
**EH87.LIB, 8087.LIB, LRG.LIB TO MYPROG BIND <cr>
```

K.5.2 Examples

The following examples show how to execute Fortran-86 programs in different environments:

1. To execute a Fortran-86 program in a full-featured operating system environment, link in all of the Fortran-86 run-time support libraries. If the application also requires support for floating-point arithmetic, link in the appropriate numerics libraries. For example, the link sequence for the 8087 microprocessor is

   ```
   LINK86 MYPROG.OBJ, CEL87.LIB, F86RN0.LIB, F86RN1.LIB, & <cr>
   **F86RN2.LIB, F86RN3.LIB, F86RN4.LIB, EH87.LIB, 8087.LIB, & <cr>
   **LGR.LIB TO MYPROG.86 BIND <cr>
   ```

   By using the BIND option with LINK86, the output file is ready to be executed, assuming that the operating system has an LTL loader.

2. To execute a Fortran-86 program and produce code for a bare machine (or minimal operating system) environment link in the run-time libraries F86RN0.LIB, F86RN1.LIB, and F86RN2.LIB. If the program requires numerics support and you are using the 8087 chip, the link command is

```
LINK86 MYPROG.OBJ, F86RN0.LIB, F86RN1.LIB, & <cr>
**F86RN2.LIB, RTNULL.LIB, EH87.LIB, 8087.LIB, &<cr>
**TO MYPROG.86 BIND <cr>
```

   In this example, string and 32-bit integer operations are fully supported. Fortran input/output is not supported; if used, LINK86 will generate an UNRESOLVED EXTERNALS warning.

   When linking in numerics support and an 8087 exception occurs, RTNULL.LIB will simply execute a HLT instruction. Since there are no external references between RTNULL.LIB and EH87.LIB, the exception handler will never be called. Consequently, it should not be included in the link sequence.

3. This example links a program using internal I/O only.

```
-LINK86 MYPROG.OBJ, F86RN0.LIB, F86RN1.LIB, & <cr>
**F86RN2.LIB, RTNULL.LIB, 87NULL.LIB, TO MYPROG.86 BIND <cr>
```

4. This example does only internal I/O and floating point arithmetic.

```
-LINK86 MYPROG.OBJ, CEL87.LIB, F86RN0.LIB, F86RN1.LIB, F86RN2.LIB, & <cr>
**RTNULL.LIB, 87ERH.LIB, E8087, E8087.LIB TO MYPROG.86 BIND <cr>
```

K.6 Locating Object Modules

Chapter 14 discusses object module location. To locate, load, and execute a module in an iRMX 86 environment, you must reserve memory during the iRMX 86 configuration process. If the memory is not reserved, the operating system will assign the memory to other tasks as dynamic memory.

The following is a sample locate operation using the default settings for controls:

```
LOC86 SAMPL1.LNK<cr>
```

This sample locate operation binds the logical segments of SAMPL1.LNK to addresses beginning at the default 00200H (hexadecimal). The output module is called SAMPL1 (the root name of the input module without the LNK extension). Unless specified with a TO clause, the output module (the absolutely located program) will always have the same root name as the input module.

The following sample operation locates a program using the ORDER and ADDRESSES control:

```
LOC86 SAMPL2.LNK, &<cr>
**ORDER (CLASSES (CODE, STACK, DATA)) &<cr>
**ADDRESSES(CLASSES (CODE (20000H),STACK (4F000H)))<cr>
```

In the invocation line, use the ampersand character (&) to continue a long line without executing it.

This sample locate operation collected the logical segments by class names in the order specified in the ORDER control. The locator then assigned addresses specified in the ADDRESSES control to the logical segments collected into the CODE and STACK classes. The DATA class received its address assignment for the default algorithm.

K.7 Preconnecting Files

When running a program on an iRMX 86-based system you can also use the UNIT control to override the default preconnections. The format of the UNIT control in an iRMX 86-based system is

source (UNIT*n* = *path*)

where

source	is the pathname of your relocated object code.
n	is a number between 0 and 255.
path	is a logical name or pathname for a file or device.

Chapter 14 discusses preconnecting files in more detail.

K.8 Executing Programs in an iRMX™ 86 Environment

To execute a complete program in an iRMX 86 environment, enter the pathname of the program file. For example, the following command locates and executes a file named PROG:

```
-PROG.86<cr>
```

Since the iRMX 86 Operating System searches several directories for files to execute, PROG could reside in the default directory (:$:), the program directory (:PROG:), or some other directory. The directories searched and the order of search are iRMX 86 configuration parameters. However, if you are unsure, enter the complete pathname. For example, the following command:

```
-:PROG:PROGRM
```

loads and executes the file PROGRM residing in the :PROG: directory.

K.9 iRMX™ 86 Specific Examples

The last page of this appendix (the fold-out) lists the run-time libraries needed to execute the examples found in Chapter 10 on an iRMX 86-based system.

K.10 Related Publications

For information on the iRMX 86 operating system, see the following manuals:

iRMX™ 86 Human Interface Reference Manual, 9803202
iRMX™ 86 Nucleus Reference Manual, 9803122
EDIT Reference Manual, 143587

10.1 I/O Examples

Example 10.1.1 Program 1A (PROG1A.FTN)

Link the program to the libraries 87NULL.LIB and LRG.LIB.

Example 10.1.2 Program 1B (PROG1B.FTN)

Link the program to the libraries 87NULL.LIB and LRG.LIB.

Example 10.1.3 Program 1C (PROG1C.FTN)

Link the program to the libraries 87NULL.LIB and LRG.LIB.

10.2 TEMPREAL Example

Example 10.2.1 Program 2 (PROG2.FTN)

Link the program to the libraries CEL87.LIB, EH87.LIB, 8087.LIB, and LRG.LIB.

10.3 $INTERRUPT Example

Example 10.3.1 Program 3 (PROG3.FTN)

Do not execute this program on an iRMX 86 Operating System. The iRMX 86 Operating System implements its own form of interrupt processing. All programs that run in an iRMX 86 environment must use iRMX 86 system calls to set up interrupt processing routines.

10.4 $REENTRANT Example

Example 10.4.1 Program 4 (PROG4.FTN)

Link the program to the libraries 87NULL.LIB and LRG.LIB.

10.5 Function Subprogram Example

Example 10.5.1 Program 5 (PROG5.FTN)

Link the program to the libraries CEL87.LIB, EH87.LIB, 8087.LIB, and LRG.LIB.

INDEX

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Intel Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

2. Does the publication cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating).______

NAME ______ DATE ______

TITLE ______

COMPANY NAME/DEPARTMENT ______

ADDRESS ______

CITY ______ STATE ______ ZIP CODE ______

(COUNTRY)

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS ...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN U.S.A.

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

intel®

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.