IBM RT PC

VS FORTRAN User's Guide

Programming Family

Personal
Computer
Software

SH23-0129

IBM RT PC

VS FORTRAN User's Guide

Programming Family

Personal
Computer
Software

First Edition (March 1987)

The information in this manual applies to Version 1 of IBM RT PC VS FORTRAN for use with Release 2.1 of the AIX Operating System; and it applies to all subsequent releases and modifications until otherwise indicated in new editions or Technical Newsletters.

Changes are made periodically to the information herein; these changes will be incorporated in new editions of this publication.

Preface

This manual is a user's guide to compiling FORTRAN programs using IBM RT PC VS FORTRAN on an RT Personal Computer (RT PC[1]) operating on the AIX[2] Operating System.

For a formal description of IBM RT PC VS FORTRAN, see the *RT PC VS FORTRAN Reference Manual.*

Contents:

- Chapter 1 — "Introduction" describes the highlights of IBM RT PC VS FORTRAN. The chapter also contains a diagram showing the main steps in creating a FORTRAN program under the AIX Operating System, and a diagram illustrating the compilation process.
- Chapter 2 — "The Compiler" provides the information necessary to compile FORTRAN programs, and describes each of the available command-line options and compiler directives.
- Chapter 3 — "Opening Files for Input and Output" describes how to open files for input and output under the AIX Operating System.
- Chapter 4 — "Data Representations" describes how IBM RT PC VS FORTRAN represents data storage.
- Chapter 5 — "Mixing Languages" describes the procedures to follow when mixing program elements written in IBM RT PC VS FORTRAN, IBM RT PC VS Pascal, and IBM RT PC C Version 1. It also illustrates the mechanism for passing parameters to subroutines and functions.

---

1 RT PC is a trademark of IBM Corporation

2 AIX is a trademark of IBM Corporation

- Chapter 6 — "The Disassembler" describes how to translate binary code modules into assembly language equivalents.

- Appendix A — "Messages" lists the compile-time and run-time messages.

- Appendix B — "ASCII Character Set" lists the decimal, octal, and hexadecimal values for the American National Standard ASCII characters.

- Appendix C — "Migrating Programs" describes the limitations and uncertainties to be aware of when compiling code written in IBM VS FORTRAN Version 2, IBM RT PC FORTRAN 77 Version 1.1, and VAX[3] FORTRAN Version 3.

Related Publications

You may want to refer to the following IBM RT PC publications for additional information:

- *VS FORTRAN Reference Manual*, SH23-0130, describes the FORTRAN 77 programming language as implemented on the RT PC.

- *VS Pascal User's Guide*, SH23-0127, describes the procedures for compiling and running RT PC VS Pascal programs under the AIX Operating System.

- *VS Pascal Reference Manual*, SH23-0128, describes the Pascal programming language as implemented on the RT PC.

- *VS Language/Operating System Interface Library*, SH23-0131, describes the system routines that can be called from FORTRAN and Pascal programs.

---

[3] VAX is a trademark of Digital Equipment Corporation

- *Concepts*, GC23-0784, gives an overview of the RT PC hardware, the AIX Operating System, and supporting publications.

- *Installing and Customizing the AIX Operating System*, SV21-8001, provides step-by-step instructions for installing and customizing the AIX Operating System, including instructions for adding devices to and deleting them from the system and for defining device characteristics. This book also explains how to create, delete, and change AIX and non-AIX minidisks.

- *Messages Reference*, SV21-8002, lists messages displayed by the RT PC and explains how to respond to the messages.

- *Usability Services Guide* and *Usability Services Reference*, SV21-8003, show how to create and print text files, work with directories, start application programs, and do other basic tasks.

- *Using and Managing the AIX Operating System*, SV21-8004, contains information on using AIX Operating System commands, working with the file system, developing shell procedures, and performing such system-management tasks as creating and mounting file systems, backing up the system, and repairing file-system damage.

- *AIX Operating System Commands Reference*, SV21-8005, lists and describes the AIX Operating System commands.

- *C Language Guide and Reference*, SV21-8008, provides information for writing, compiling, and running C language programs.

- *AIX Operating System Technical Reference*, SV21-8009, describes the system calls and subroutines a programmer would use to write application programs. This book also provides information about the AIX Operating System file system, special files, miscellaneous files, and the writing of device drivers.

- *AIX Operating System Programming Tools and Interfaces*, SV21-8010, describes the programming environment of the AIX Operating System and includes information about the use of operating system tools to develop, compile, and debug programs.

- *AIX Operating System DOS Services Reference*, SV21-8012, provides step-by-step information for using the AIX Operating System shell. In addition, this book describes the DOS system services.

- *User Setup Guide*, SV21-8020, provides instructions for setting up and connecting devices to system units. It also gives procedures for installing the AIX Operating System and for testing the setup.

- *Guide to Operations*, SV21-8021, describes system units, displays, console keyboard, and other devices that can be attached to the RT PC. This guide also includes procedures for operating the hardware and for moving system units.

- *Problem Determination Guide*, SV21-8022, provides instructions for running diagnostic routines for hardware and problem-determination procedures for software.

You may also want to consult the IBM RT PC FORTRAN 77 Version 1.1 publications.

Contents

Chapter 1. Introduction

IBM RT PC VS FORTRAN is an easy-to-use, high-level programming language for the RT Personal Computer. It compiles source code in FORTRAN as defined by IBM VS FORTRAN Version 2, IBM RT PC FORTRAN 77 Version 1.1, ANSI Standard FORTRAN 77, and VAX FORTRAN Version 3.

In addition to excellent performance, IBM RT PC VS FORTRAN offers these enhanced functions:

- Automated installation
- Source compatibility with IBM VS FORTRAN Version 2[1]
- Source compatibility with IBM RT PC FORTRAN 77 Version 1.1[1]
- Source compatibility with ANSI Standard FORTRAN 77
- Source compatibility with VAX FORTRAN Version 3[1]
- Optimized executable code
- Excellent compile-time performance
- An operating system interface library
- No significant limit on program size
- No significant limit on data size
- Separate unit compilation
- Access to command-line options
- Common development/debugging environment
- Detailed screen messages
- Easy inter-language linkages with FORTRAN and C.

You may select one of four compiler modes: IBM mode, R1 mode, AN mode, or VX mode. You may work in the mode you need or with which you are most familiar.

---

1 See Appendix C, "Migrating Programs" for limitations.

IBM Mode

This is the default mode of the compiler, and it allows you to compile code written in IBM VS FORTRAN Version 2 (see Appendix C, "Migrating Programs" for limitations).

You may develop and run IBM mode programs entirely on the RT PC. As a cost-effective development tool, you may develop and run IBM mode programs on an independent RT PC workstation and then move the programs to a mainframe that uses VS FORTRAN Version 2.

You may also take programs written in IBM VS FORTRAN Version 2 from a mainframe and run them on your RT PC.

IBM mode contains all of the ANSI Standard FORTRAN 77 requirements; you may use ANSI Standard FORTRAN 77 code in IBM mode, and can improve it using IBM mode enhancements.

R1 Mode

This mode allows you to compile code written in IBM RT PC FORTRAN 77 Version 1.1 (see Appendix C, "Migrating Programs" for limitations). You can take code written in this version of FORTRAN and recompile it in IBM RT PC VS FORTRAN in order to take advantage of its improvements and additional features.

AN Mode

This mode allows you to compile code written in ANSI Standard FORTRAN 77. Code that is to adhere to this definition of FORTRAN can be compiled in this mode; during program compilation, you are warned when any extension to this definition is used.

VX Mode

This mode allows you to compile code written in VAX FORTRAN Version 3 (see Appendix C, "Migrating Programs" for limitations). You may take programs written in VAX FORTRAN Version 3 from a mainframe and run them on your RT PC.

An additional advantage of IBM RT PC VS FORTRAN is that you have the ability to mix modes in creating an executable program. However, each separate unit compilation may use only a single mode.

You should note that some programs may produce different results when run on the RT PC compared to other machines because of differences in machine architecture, operating systems, or compiler implementations. These differences, along with the limitations of each mode, are noted in Appendix C, "Migrating Programs."

FORTRAN Programs Under AIX

As illustrated in Figure 1-1 on page 1-4, the four main steps in creating an executable FORTRAN program under the AIX Operating System are:

1. Create your program using a text editor and store it with a ".for" or ".f" extension.

2. Compile your source program to generate a binary file.

3. Link the output with the AIX system linker "cc" to create an executable file.

4. Run the program.

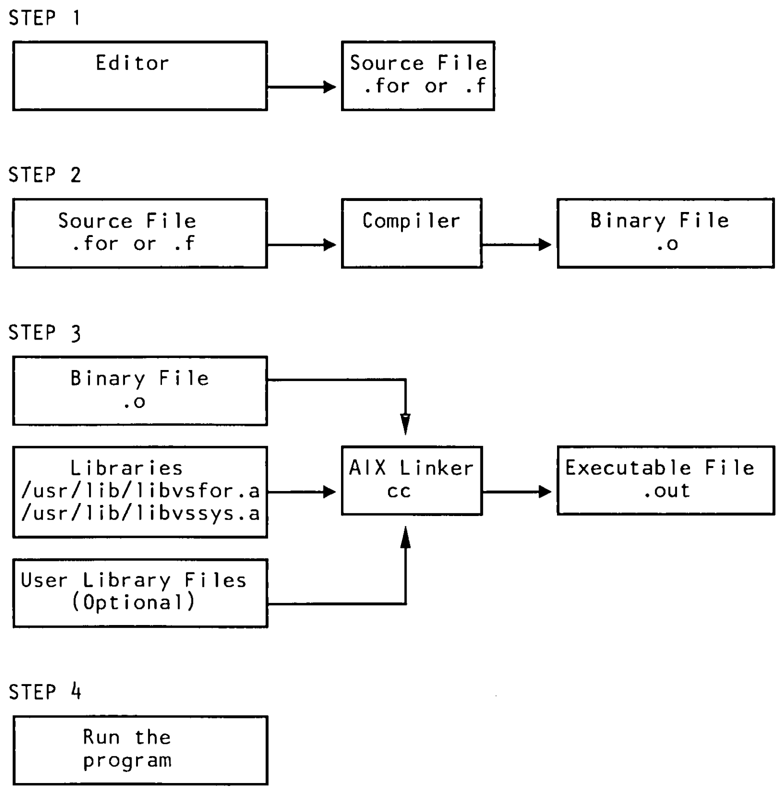

Figure 1-1. Creating a FORTRAN Program Under AIX

Compilation Process

As illustrated in Figure 1-2 on page 1-6, the compiler follows these steps when invoked:

1. The source file is passed to "vsfort", which produces intermediate code with an ".i" extension.

2. The ".i" file is passed to "vspass2", the code generator.

3. Code is then passed to "vspass3".

   a. If the "g+" command-line option was set when the code was passed to "vsfort", then "vspass3" creates both a ".dbg" file that can be used with the Diassembler (disasm) and a binary file (.o file). The ".o" file is passed to the AIX linker (cc) which creates an executable file. The executable file can be debugged by the Symbolic Debugger (sdb).

      Note: If both the "d+" and the "g+" command-line options are set, regardless of their order on the command line, the "g+" option has the higher priority.

   b. If the "d+" command-line option was set when the code was passed to "vsfort", then "vspass3" creates both a ".dbg" file that can be used with the Disassembler (disasm) and a binary file (.o file). The ".o" file is passed to the AIX linker (cc) which creates an executable file. The executable file cannot be debugged by the Symbolic Debugger (sdb).

   c. If neither command-line option was set when the code was passed to "vsfort", then "vspass3" creates a binary file (.o file). The ".o" file is passed to the AIX linker (cc) which creates an executable file. The executable file cannot be debugged by the Symbolic Debugger (sdb).

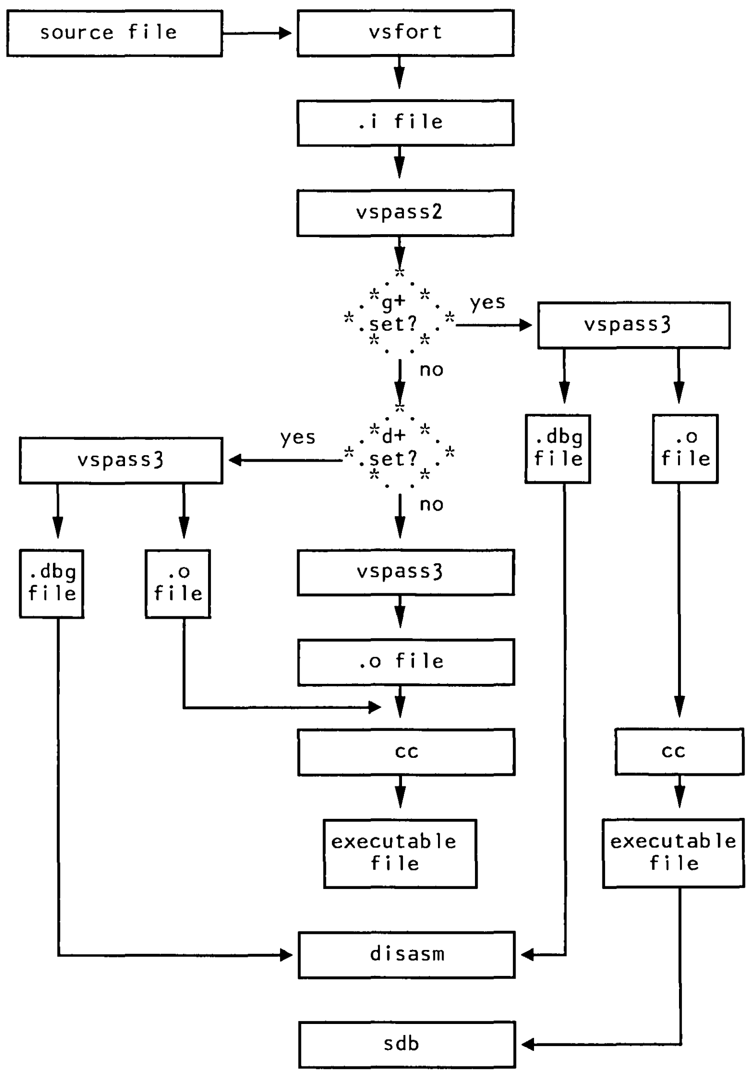

Figure 1-2. Compilation Process

Methods of Presentation

In this guide:

- Italicized letters and words represent variables, for which user-supplied information is substituted. For example, the word "*sourcefl*" in a format could be coded as "myprog1".
- Brackets "[]" indicate that an item is optional. For example, the specification "[*option*]" in a format could be left blank or coded as "a+" as needed.
- An ellipsis (...) indicates that the preceding specification can, optionally, be repeated. For example, the specification "[*option*] ..." in a format could be left blank or coded as "a+" or "a+ d+" or "a+ d+ f+" as needed.
- All other words, letters, and symbols are to be coded as shown.
- The general rule in FORTRAN for spaces (blanks) is that they have no significance in statements, and are used to improve readability. Space and blank are synonymous in this manual.
- The phrase "FORTRAN 77" refers to ANSI Standard FORTRAN 77.
- The phrase "FORTRAN 66" refers to ANSI Standard FORTRAN 66.

Chapter 2. The Compiler

FORTRAN source code is compiled on the RT PC by executing the "vsf" compiler, which produces binary code from the FORTRAN source code. The binary code is then linked using the AIX Operating System linker "cc".

This chapter describes how to execute the compiler, and includes a description of each of the command-line options and compiler directives available in IBM, R1, VX, and AN modes. Compiler option modification and optimization capabilities are also described.

Note: For more information on the AIX Operating System linker, see the *AIX Operating System Commands Reference* manual.

Invoking the Compiler

The format for running the compiler from the command line is:

```
vsf sourcefl [ option ] ...
```

sourcefl
: is the name of a source file that has a ".for" or ".f" extension. If the extension is not specified, the compiler searches for a "*sourcefl*.for" file, and if not found, then searches for a "*sourcefl*.f" file. This is the only argument required for the compiler's operation.

option
: is any of the command-line options listed in "Command-Line Options" on page 2-3. If no options are specified, the compiler:

- writes error messages to the standard output device
- generates calls to a compatible software library
- gives local variables AUTOMATIC implementation (except those appearing in SAVE statements or those initialized in DATA statements)
- reads source programs in fixed-form format
- compiles in IBM mode
- produces warning messages.

Example:

```
C  SAMPLE PROGRAM "sample.for"
1     INTEGER I
2     REAL REAL
3     REAL IFINAL
4     DOUBLE PRECISION EXACT
5     FORMAT(1X,5F8.3)
6     DO 11,I=1,5
7     REAL=3.12*I
8     EXACT=REAL/2
9     IFINAL=(EXACT*2)*(I-1)+REAL
10    WRITE (6,5) IFINAL
11    CONTINUE
12    STOP
13    END
```

To compile this program, enter `vsf sample`. This command invokes a shell script named "vsf", which runs the compiler.

The screen displays the message:

```
0 errors. 13 lines.  File sample.for
0 warnings.
```

The program now runs whenever `sample` is entered. The output from this program is:

```
sample
    3.120
   12.480
   28.080
   49.920
   78.000

Programmed STOP
```

Command-Line Options

Command-line options are provided to change any of the compiler's default settings. Figure 2-1 on page 2-8 lists all of the options and indicates the modes in which each one can be used.

a+ CONDITIONAL COMPILATION
instructs the compiler to also compile lines that have an uppercase or lowercase D in column 1.

This option is available in VX mode.

d+ DISASSEMBLER INFORMATION
instructs the compiler to put disassembler information in the ".dbg" file. This option is required if this module is to be disassembled using the RT PC Disassembler. This option also prepares the binary files for profiling. For more information on profiling, see the *AIX Operating System Commands Reference* manual.

This option is available in all modes.

e*filename* ERROR FILE
instructs the compiler to place its error output in *filename*. If the e*filename* option is not specified, error messages are written to the standard output device.

This option is available in all modes.

f+ FLOATING-POINT HARDWARE
instructs the compiler to generate in-line calls to floating-point hardware. The floating-point hardware is required at run time but is optional at compile time.

This option is available in all modes.

g+ DEBUGGER INFORMATION
instructs the compiler to put debugger information in the executable file. This option is required if this module is to be debugged using the RT PC Symbolic Debugger.

This option is available in all modes.

h+ STATIC IMPLEMENTATION
instructs the compiler to give local variables STATIC implementation.

This option is available in all modes.

i*n1,n2,...nn* CONDITIONAL INCLUDE
instructs the compiler to selectively activate the INCLUDE statement within the FORTRAN source code during compilation. If the number specified in an INCLUDE statement's optional *nnn* parameter is also specified in this option's *n1,n2,...nn* list, the contents of the file specified in the INCLUDE statement are included in the compilation. The variables *nnn* and *n1,n2,...nn* are numbers from 1 through 255, and there are no default values. The INCLUDE statement is described in "Compiler Directives" on page 2-9.

This option is available in IBM mode.

k- FREE-FORM FORMAT
instructs the compiler to read the input source program in free-form format. Formats are described in the *RT PC VS FORTRAN Reference Manual.*

This option is available in IBM mode.

l*filename*	LISTING FILE instructs the compiler to place its listing output in *filename*. If the l*filename* option is not specified, a listing file is not generated. This option is available in all modes.
l+	LIST TO STANDARD OUTPUT DEVICE instructs the compiler to generate a listing to the standard output device. This option is available in all modes.
man	AN MODE instructs the compiler to compile in AN mode. Modes are described in Chapter 1, "Introduction."
mr1	R1 MODE instructs the compiler to compile in R1 mode. Modes are described in Chapter 1, "Introduction."
mvx	VX MODE instructs the compiler to compile in VX mode. Modes are described in Chapter 1, "Introduction."
n*xxx*	MAXIMUM CHARACTER LENGTH specifies the maximum length for any character variable, character array element, or character function (where *xxx* is a number from 1 through 32767). Within a program unit, you cannot specify a character length greater than the number specified. The default value of *xxx* is 500. This option is available in IBM mode.
o1+	OPTIMIZATION LEVEL 1 instructs the compiler to use optimization level 1 (see "Optimization of Programs" on page 2-13). This option is available in all modes.

o2+	OPTIMIZATION LEVEL 2 instructs the compiler to use optimization level 2 (see "Optimization of Programs" on page 2-13). This option is available in all modes.
o3+	OPTIMIZATION LEVEL 3 instructs the compiler to use optimization level 3 (see "Optimization of Programs" on page 2-13). This option is available in all modes.
o4+	OPTIMIZATION LEVEL 4 instructs the compiler to use optimization level 4 (see "Optimization of Programs" on page 2-13). This option is available in all modes.
t-	NO CHARACTER TRANSFORMATION instructs the compiler not to perform transformation on any characters. Uppercase and lowercase letters are significant in the program, and keywords are only recognized in lowercase. This option is available in R1 mode.
u-	NO IMPLICIT VARIABLE TYPING instructs the compiler not to implicitly type variables. When this option is specified, all variables must be explicitly declared. This option is available in R1 and VX modes.
v-	NO COMPILER PROGRESS INFORMATION instructs the compiler not to generate information on the progress of the compile. This option is available in all modes.
w-	NO WARNING MESSAGES instructs the compiler not to generate warning messages. This option is available in IBM, R1, and VX modes.

x+ CROSS-REFERENCE LISTING
instructs the compiler to generate a cross-reference listing of the source code file. The cross-reference listing appears in the ".lst" file; therefore, the l*filename* option must also be specified.

This option is available in all modes.

y+ FORTRAN 66 FEATURES
instructs the compiler to accept these FORTRAN 66 features:

- execute DO loops at least once
- allow character and numeric data to be assigned to the same common block
- allow character and numeric data to be "equivalenced"
- allow non-character variables to be initialized with character data statements via the DATA statement
- have INTEGER*2 as the default integer data type
- have LOGICAL*1 as the default logical data type.

For more information on FORTRAN 66 compatibility features, see the *RT PC VS FORTRAN Reference Manual.*

This option is available in all modes.

z*cb1,cb2,...cbn* COMMON BLOCK ALLOCATION
defines the names of common blocks that are to be allocated at execution time, where *cb1,cb2,...cbn* are common block names. This option allows the specification of very large common blocks that can reside in the additional storage space available through the AIX Operating System. No blanks are allowed in the list.

This option is available in IBM mode.

Note: In any instance where a command-line option conflicts with an @PROCESS statement in IBM mode or an OPTIONS statement in VX mode, the @PROCESS or OPTIONS statement prevails. For example, if you specify x+ as an option on the command line yet use @PROCESS NOXREF, no cross-reference information is generated. The @PROCESS and OPTIONS statements are described in "Modifying Compiler Options" on page 2-10.

Option	Function	IBM	R1	VX	AN
a+	Conditional compilation			•	
d+	Disassembler information	•	•	•	•
e*filename*	Error file	•	•	•	•
f+	Floating-point hardware	•	•	•	•
g+	Debugger information	•	•	•	•
h+	Static implementation	•	•	•	•
i*n1,n2,...nn*	Conditional INCLUDE	•			
k-	Free-form format	•			
l*filename*	Listing file	•	•	•	•
l+	List to standard output device	•	•	•	•
man	AN mode				•
mr1	R1 mode		•		
mvx	VX mode			•	
n*xxx*	Maximum character length	•			
o1+	Optimization level 1	•	•	•	•
o2+	Optimization level 2	•	•	•	•
o3+	Optimization level 3	•	•	•	•
o4+	Optimization level 4	•	•	•	•
t-	No character transformation		•		
u-	No implicit variable typing		•	•	

Figure 2-1 (Part 1 of 2). Compiler Command-Line Options

Option	Function	IBM	R1	VX	AN
v-	No compiler progress information	•	•	•	•
w-	No warning messages	•	•	•	
x+	Cross-reference listing	•	•	•	•
y+	FORTRAN 66 features	•	•	•	•
z*cb1,cb2,...cbn*	Common block allocation	•	•	•	•

Figure 2-1 (Part 2 of 2). Compiler Command-Line Options

Compiler Directives

Compiler directives are an extension to ANSI Standard FORTRAN 77 and provide additional controls over the compiler's actions.

For fixed-form input format, the first character of a compiler-directive is entered in column 7 or after. For free-form input format (available in IBM mode), a compiler directive can start in any column. For a description of input formats, see the *RT PC VS FORTRAN Reference Manual.*

EJECT
starts a new page of the source listing.

This directive is available in IBM mode.

INCLUDE (*filename*) [*nnn*] (IBM mode)
INCLUDE '*filename*' (R1 and VX modes)
includes the contents of the file *filename* in the program source code. The *nnn* is a number from 1 through 255 used to decide whether to include the file during compilation. When *nnn* is not specified, the file is always included. When *nnn* is specified, the file is included only if the number is included in the number list of the "i" command-line

option, described in "Command-Line Options" on page 2-3, or is specified in an @PROCESS CI(*nnn*) statement, described in "@PROCESS Statement" on page 2-10.

The contents of the included file appear in the source code as if it had been written there. Included files can be nested to a maximum of five.

This directive is available in IBM, R1, and VX modes.

Directive	**Function**	**IBM**	**R1**	**VX**	**AN**
EJECT	New page	•			
INCLUDE	Include file	•	•	•	

Figure 2-2. Compiler Directives

Modifying Compiler Options

The command-line options specified when the compiler is invoked remain in effect throughout a program's compilation unless they are overridden with the @PROCESS statement (available in IBM mode) or the OPTIONS statement (available in VX mode).

@PROCESS Statement

To modify compiler options in IBM mode, the @PROCESS statement must be the first statement in the program unit that is to be affected. The form of the @PROCESS statement is:

@PROCESS *option* **[,** *option* **] ...**

option
can be any of the following keywords (other keywords are ignored):

FREE	FREE-FORM FORMAT instructs the compiler to read the input source program in free-form format. Formats are described in the *RT PC VS FORTRAN Reference Manual.*
FIXED	FIXED-FORM FORMAT instructs the compiler to read the input source program in fixed-form format. Formats are described in the *RT PC VS FORTRAN Reference Manual.*
XREF	CROSS-REFERENCE LISTING instructs the compiler to generate a cross-reference listing of the source code file.
NOXREF	NO CROSS-REFERENCE LISTING instructs the compiler not to generate a cross-reference listing of the source code file.
FIPS	NON-ANSI STANDARD FLAGS instructs the compiler to flag non-ANSI Standard FORTRAN 77 items as errors.
NOFIPS	NO NON-ANSI STANDARD FLAGS instructs the compiler not to flag non-ANSI Standard FORTRAN 77 items.
CL(*nnn*)	MAXIMUM CHARACTER LENGTH specifies the maximum length for any character variable, character array element, or character function (where *nnn* is a number from 1 through 32767). Within a program unit, you cannot specify a character length greater than the number specified. The default value of *nnn* is 500.
CI(*nnn*)	CONDITIONAL INCLUDE instructs the compiler to selectively activate the INCLUDE statement within the FORTRAN source code during compilation. If the number specified in an INCLUDE statement's optional *nnn* parameter matches the number specified in an @PROCESS CI(*nnn*) state-

ment, the contents of the file specified in the INCLUDE statement are included in the compilation. The *nnn* is a number from 1 through 255, and there is no default value. The INCLUDE statement is described in "Compiler Directives" on page 2-9.

DC(*cbname*) COMMON BLOCK ALLOCATION
defines the name of a common block that is to be allocated at execution time, where *cbname* is a common block name. This option allows the specification of very large common blocks that can reside in the additional storage space available through the AIX Operating System.

The @PROCESS statement must appear in columns 1–8, and the options must appear in columns 9–72 of the statement. Multiple @PROCESS statements can be used in a program unit.

OPTIONS Statement

To modify compiler options in VX mode, the OPTIONS statement must be the first statement in the program unit that is to be affected. The form of the OPTIONS statement is:

OPTIONS *option* **[,** *option* **] ...**

option
can be any of the following keywords (other keywords are ignored and warning messages are issued):

/NOI4 2-BYTE INTEGER
instructs the compiler to allocate 2 bytes for the INTEGER data type.

/I4 4-BYTE INTEGER
instructs the compiler to allocate 4 bytes for the INTEGER data type.

/NOF77 NON-FORTRAN 77 FEATURES
instructs the compiler to accept these FORTRAN 66 features that are not found in FORTRAN 77:

- execute DO loops at least once
- allow character and numeric data to be assigned to the same common block
- allow character and numeric data to be "equivalenced"
- allow non-character variables to be initialized with character data statements via the DATA statement
- have INTEGER*2 as the default integer data type
- have LOGICAL*1 as the default logical data type.

For more information on FORTRAN 66 compatibility features, see the *RT PC VS FORTRAN Reference Manual.*

/F77 FORTRAN 77 FEATURES ONLY
instructs the compiler to accept only FORTRAN 77 features.

Optimization of Programs

"Optimization" refers to the process of improving the execution performance of a given program. It is done at the cost of compile time but results in reduced execution time. The RT PC VS FORTRAN compiler performs two separate optimization passes — machine-dependent optimizations and machine-independent optimizations — which are controlled by selecting compiler command-line options. The command-line options for optimization are:

o1+ MACHINE-DEPENDENT OPTIMIZATION
instructs the compiler to perform a machine-dependent optimizing pass, which takes place after the code-generation phase of the compilation. This pass examines object code at the basic block level and includes:

- eliminating unnecessary branches
- eliminating redundant loads and stores
- exploiting machine idioms
- replacing branches with branch-with-execute instructions
- strength reduction.

o2+ MACHINE-INDEPENDENT OPTIMIZATION
instructs the compiler to perform a machine-independent optimizing pass that includes:

- constant folding
- straightening
- eliminating unreachable code
- copy propagation
- eliminating dead code.

o3+ MACHINE-INDEPENDENT OPTIMIZATION
instructs the compiler to perform a machine-independent optimizing pass that includes:

- eliminating common subexpressions
- subscript optimization
- eliminating induction variables
- loop invariant code motion.

o4+ MACHINE-DEPENDENT AND MACHINE-INDEPENDENT OPTIMIZATION
instructs the compiler to perform machine-dependent and machine-independent optimization.

Optimization Considerations

The choice of algorithm for a given task can have a much greater impact on execution speed than any compiler optimization. It is generally true that most of program execution time is spent on less than 10% of the code. Changes to the algorithm in the critical 10% frequently have dramatic results.

The optimizing feature of the compiler should not be used while developing programs. Some optimizations move statements from one area of a program to another, or change statements or variables in a way that is not obvious.

Since this makes debugging programs more difficult, optimizing before debugging should be avoided. After a program is developed, it can be recompiled with optimization command-line options to improve its performance.

Note: The optimization process is disabled whenever the "g+" option (debugger information) is specified on the command line.

Chapter 3. Opening Files for Input and Output

This chapter describes how to open files for input and output using RT PC input and output facilities under the AIX Operating System.

For a description of the available FORTRAN input/output facilities, see the *RT PC VS FORTRAN Reference Manual.*

Note: *This chapter describes the procedures for opening files in IBM mode. For all other modes, the name specified with the statement that opens the file must be a physical file name. For more information, see the RT PC VS FORTRAN Reference Manual.*

The name of a file to be opened for input or output may be "environment determined" or "program determined". Using an environment-determined file name permits the file name to be specified on the AIX command line at program execution time, which is done using AIX environment variables. Program-determined file names are specified through OPEN statement options.

Opening Files with Environment-Determined Names

The name of an input or output file can be determined at program execution time by using environment-determined file names. The environment variable file names are specified by using a program variable as the name of a file. This permits access to a different file each time the program is executed. The two methods of opening environment-determined files are:

- using environment variables on the command line
- using environment variables in shell scripts.

The following FORTRAN program, "MYPGM", is a sample program used in the descriptions of the input and output procedures throughout this chapter:

```
      PROGRAM MYPGM;
      CHARACTER*1000 BUFFER
      OPEN(UNIT=8,FILE='INFILE')
      OPEN(UNIT=9,FILE='OUTFILE',STATUS='NEW')
123   READ(8,800,END=999)BUFFER
800   FORMAT(A)
      WRITE(9,800)BUFFER
      GOTO 123
999   STOP
      END
```

Note: "MYPGM", "INFILE", and "OUTFILE" are referenced as "mypgm", "infile", and "outfile", respectively, to agree with AIX conventions.

Using Environment Variables on the Command Line

AIX environment variables are used to associate a file name with the chosen program variable (for example, "infile" in the program "mypgm"). The following AIX command formats show how to use environment variables in the Bourne shell, the C shell, and the DOS shell. The AIX command is entered on the command line prior to the invocation of the program, and has the form:

Bourne shell

ENVIRONMENT-NAME=file-name; **export** *ENVIRONMENT-NAME*

C shell

setenv *ENVIRONMENT-NAME file-name*

DOS shell

set *ENVIRONMENT-NAME=file-name*

ENVIRONMENT-NAME
: is the same as the file variable name being used.

  Note: Environment variables under the AIX Operating System are case sensitive.

file-name
: is the actual file name used in the AIX file system.

Note: Unless otherwise specified, the examples in this chapter use the Bourne shell. For detailed descriptions of the C shell and the DOS shell, see the *Using and Managing the AIX Operating System* and *AIX Operating System DOS Services Reference* manuals.

In the following example, the environment variable "INFILE", which is used as a program variable, is associated with the file "file1.in.a1". The program "mypgm" is then executed.

```
INFILE=file1.in.a1; export INFILE
mypgm
```

After execution, the exported filename, "INFILE", remains in the AIX environment for any subsequent executions that use the same variable and AIX file. To run the same program using a different file, you must associate the new file name with the "INFILE" environment variable and export it again.

It is possible to execute one program in the background and to execute the same program in the foreground using different files, as shown in this example:

```
INFILE=file1.in.a1; export INFILE; mypgm&

INFILE=file2.in.a1; export INFILE
mypgm
```

By entering the program invocation on the same line as the environment statements, you associate each line's statements with its own unique AIX process. Environment variables are only known in their current environment. Therefore, "file1" is local to the first invocation of "mypgm", and "file2" is local to the second invocation of "mypgm".

Using Environment Variables in Shell Scripts

The easiest and most efficient method for executing a program containing a variable name involves the use of a shell script. All the necessary commands can be put into the shell script. When the name of the shell script file is invoked, each command in the shell script file is executed sequentially. For a complete description of shell script usage, see *Using and Managing the AIX Operating System.*

The shell script allows the use of the association of environment variable with the same file name each time, or with a different file name each time.

Shell Script Using the Same File Name: The following example illustrates a shell script named "run1" that associates the environment variable named "INFILE" with the file named "file1.in.a1". It also contains the command to execute the "mypgm" program.

The shell script "run1" contains:

```
INFILE=file1.in.a1; export INFILE
mypgm
```

After the shell script is created as a file and is made executable through the command `chmod 755 run1`, it can be executed by entering:

```
run1
```

When "run1" is executed, it is considered an AIX process. Therefore, anything that executes within the script is a child of that process. Since a child process is known only to its parent, the contents of the "INFILE" environment variable are local to the "run1" shell script and unknown to other AIX processes.

Shell Script Using Different Files: To prevent having to edit the shell script whenever a different AIX file is used, the shell script can be created with a variable in place of the file name. Using the same shell script shown in the previous example, a variable "$1" is used in place of the physical file name.

```
INFILE=$1; export INFILE
mypgm
```

To execute the shell script using the "file1.in.a1" file, enter:

```
run1 file1.in.a1
```

When the shell script is executed, "$1" is replaced with "file1.in.a1". This allows "run1" to be executed using different file names. It also allows the execution of "run1" to proceed in the background under one name and in the foreground under a second name. For example:

```
run1 file1.in.a1&
run1 file2.in.a1
```

Opening Files with Program-Determined Names

AIX file names can also be determined from within programs, which is done by using OPEN statement options. This method enables you to control which file is being opened.

Examples:

```
C     Suppose that there is no environment
C     variable set.
C
      OPEN(UNIT=9,FILE='MYFILE',STATUS='NEW',
     +     FORM='FORMATTED',ACCESS='DIRECT')
```

This OPEN statement opens the new file "FILE.MYFILE" as unit 9. The file format is formatted and the file access is direct.

```
C     Suppose that there is no environment
C     variable set.
C
      OPEN(UNIT=8,STATUS='NEW')
```

This OPEN statement opens the new file "FILE.FT08F001" (the default name) as unit 8. The file format is formatted (the default) and the file access is sequential (the default).

For more information on the OPEN statement and its options, see the *RT PC VS FORTRAN Reference Manual.*

Chapter 4. Data Representations

This chapter describes how RT PC VS FORTRAN represents data in storage. Since internal data representation is unspecified by ANSI Standard FORTRAN 77, you should be aware that any code that makes explicit assumptions about the data storage or format may not be portable. At times, however, it is necessary and convenient to write such code.

Programming procedures are usually available to determine the internal data formats used by computers to store data. In RT PC VS FORTRAN, you can output data in Z format or use FORTRAN EQUIVALENCE to provide a window into the internal storage mechanism and number representations used by the compiler.

Storage Allocated For Each Data Type

In general, a word value (a value that occupies 32 bits) is aligned on a word boundary, and data types larger than a word are also aligned on a word boundary. Values that can fit into a single byte are aligned on a byte boundary, which means that any data type larger than a byte has an even-numbered address for its first byte. Data types and arrays that are 2 bytes or larger are aligned on a word boundary.

RT PC VS FORTRAN supports integer, real, double-precision, complex, double-complex, character, and logical data types. Integers are represented internally in twos complement notation. The *1 and *2 specifications are non-ANSI Standard FORTRAN 77 features used to indicate that certain data types are to occupy less than the standard amount of storage.

The standard integer data type (INTEGER) and INTEGER*4 each occupy 1 word (4 bytes, or 32 bits) of storage aligned on a word boundary. INTEGER and INTEGER*4 can assume values from -2,147,483,648 through 2,147,483,647.

The INTEGER*2 data type occupies a halfword (2 bytes, or 16 bits) of storage aligned on a halfword boundary and has an even-numbered address. INTEGER*2 can assume values from -32768 through 32767.

The standard real data type (REAL) and REAL*4 each occupy 1 word (4 bytes, or 32 bits) of storage aligned on a word boundary, and each has an even-numbered address. REAL and REAL*4 can assume values from -3.402824E+38 through -1.175494E-38, 0, and from 1.175494E-38 through 3.402824E+38, with a precision of one part in 2**23 (about seven decimal places). Real data is stored with a sign bit, an 8-bit exponent, and a 23-bit mantissa. The byte with the sign bit has the smallest address of the 4 bytes. RT PC VS FORTRAN real data conforms to the IEEE standard for floating-point data.

REAL*8 is the same as the double-precision data type (DOUBLE PRECISION).

The double-precision data type (DOUBLE PRECISION) occupies 2 words (8 bytes, or 64 bits) of storage aligned on a word boundary and has a 4-byte address. DOUBLE PRECISION can assume values from -1.797693D+308 through -2.225074D-308, 0, and from 2.225074D-308 through 1.797693D+308, with a precision of one part in 2**52 (about 16 decimal places). Double-precision data is stored with a sign bit, an 11-bit exponent, and a 52-bit mantissa. The byte with the sign bit has the smallest address of the 8 bytes. RT PC VS FORTRAN REAL*8 and DOUBLE PRECISION data conforms to the IEEE standard for double-precision floating-point data.

The standard complex data type (COMPLEX) occupies 2 words of storage because it is represented as a pair of single-precision real data values. The first element, which has the smaller address, represents the "real" part of the complex number and the second element represents the "imaginary" part of the number.

COMPLEX*16 occupies 4 words of storage because it is represented as a pair of double-precision real data values. (In R1 and VX modes, COMPLEX*16 may also be specified as DOUBLE COMPLEX.) The first element, which has the smaller address, represents the "real" part of the double-complex number and the second element represents the "imaginary" part of the number.

The character data type (CHARACTER**n*) occupies *n* bytes of storage aligned on a word boundary. RT PC VS FORTRAN uses the ASCII representation for characters and control codes. See Appendix B, "ASCII Character Set."

The standard logical data type (LOGICAL) and LOGICAL*4 each occupy 1 word (4 bytes, or 32 bits) of storage aligned on a word boundary. A value of 0 represents ".FALSE." and a value of 1 represents ".TRUE.". Any other value is undefined.

LOGICAL*1 data occupies 1 byte (8 bits) of storage aligned on a byte boundary. A value of 0 represents ".FALSE." and a value of 1 represents ".TRUE.". Any other value is undefined.

VX mode in RT PC VS FORTRAN allows LOGICAL*2 data, which occupies a halfword (16 bits) of storage aligned on a word boundary. A value of 0 represents ".FALSE." and a value of 1 represents ".TRUE.". Any other value is undefined.

Data Representation For Each Data Type

Whatever the size of the data object in use, the most significant bit is always in the byte with the lowest address of however many bytes are required to represent that object.

Representation of Integer Data

```
INTEGER*2

bit ---> 15                 0
          ___________________
         | byte 0 | byte 1  |
         |________|_________|
INTEGER, INTEGER*4

bit ---> 31                          0
        +--------+--------+--------+--------+
        | byte 0 | byte 1 | byte 2 | byte 3 |
        +--------+--------+--------+--------+
```

Integer data is stored in twos complement notation and the most significant bit is the sign bit. If b(*i*) represents the value of bit *i*, which can be either 1 or 0, and *i*=0 represents the least significant bit and *i*=*m*-1 represents the most significant bit in an *m*-bit number, the formula for the value represented in memory is:

$$\text{value} = -b(m)*(2**m) + \operatorname{sum}_{i=0}^{i=m-1} b(i)*(2**i)$$

A field of all zeros represents 0. The largest negative integer is represented by a 1 in the most significant bit position (b(*m*)=1) and zeros everywhere else. The largest positive integer is represented by a 0 in the most significant bit position (b(*m*)=0) and 1's everywhere else. A field of all 1's represents the value -1.

In twos complement notation, there appear to be more negative integers (-(2***m*)) than positive integers (2***m*)-1. For example, the range of INTEGER*2 variables is -32768 to 32767. This asymmetry is resolved if 0 is counted as a positive integer.

Representation of Floating-Point Data

RT PC VS FORTRAN conforms to the IEEE standard for representation of floating-point numbers. In general, a value is represented in IEEE floating-point format by this formula:

$$\text{value} = [(-1)**s] * [2**(\text{expfld} - \text{bias})] * [1 + \text{mantissa}/(2**\text{manwidth})]$$

The *s* represents the sign bit. The exponent field (*expfld*) and the mantissa are to be considered unsigned binary integers. The bias and the mantissa width (*manwidth*) depend upon whether a single-precision or double-precision floating-point is used. The bias for single-precision is 127 and the bias for double-precision is 1023.

The width of the exponent field determines the dynamic range of the representation, and the width of the mantissa field determines the precision of the representation. The value of the exponent field can be 0 to 255 for single-precision and 0 to 2047 for double-precision. The value of the mantissa field can be 0 to (2***manwidth*)-1.

```
REAL (single-precision floating-point)

             byte 0   byte 1  byte 2  byte 3

bit ----> 31 30    23 22                   0
          +-+--------+----------------------+
          |s| expfld |  mantissa            |
          +-+--------+----------------------+
```

For single-precision floating-point numbers, the exponent field is 8 bits wide and the mantissa field is 23 bits wide. Note that the 8 bits of the exponent are not aligned on a byte boundary. The byte with the sign bit has the smallest address of the 4 bytes.

The standard complex data type (COMPLEX) is represented as a pair of single-precision floating-point data values.

```
DOUBLE PRECISION (double-precision floating-point)

         byte 0  byte 1  byte 2      through      byte 7
bit ---> 63 62        52 51                            0
        +---+----------+-------------------------------+
        | s |  expfld  |     mantissa                  |
        +---+----------+-------------------------------+
```

For double-precision floating-point numbers, the exponent field is 11 bits wide and the mantissa field is 52 bits wide. The byte with the sign bit has the smallest address of the 8 bytes.

COMPLEX*16 (which can also be specified as DOUBLE COMPLEX in R1 and VX modes) is represented as a pair of double-precision floating-point data values.

Representations of Selected Floating-Point Numbers

These figures contain hexadecimal representations of selected single-precision and double-precision floating-point numbers.

Value	Hexadecimal Representation	s,expfld, mantissa	Notes
+0. 00000000	0,00,000000		
-0. 80000000	1,00,000000		
+1. 3F800000	0,7F,000000		
-1. BF800000	1,7F,000000		
+2. 40000000	0,80,000000		
+3. 40400000	0,80,400000		
PI 40490FDA	0,80,490FDA		
-(2**(-129))	80200000	1,00,200000	Denormalized
2**(-149)	00000001	0,00,000001	Smallest denormalized
approximately 3.403E38	7F7FFFFF	0,FE,7FFFFF	Most positive normalized
approximately -3.403E38	FF7FFFFF	1,FE,7FFFFF	Most negative normalized

Figure 4-1 (Part 1 of 2). Selected Single-Precision Floating-Point Numbers

Value	Hexadecimal Representation	s,expfld, mantissa	Notes
2**(-126)	00800000	0,01,000000	Smallest normal-ized
+infinity	7F800000	0,FF,000000	
-infinity	FF800000	1,FF,000000	
NaN	7F803303	0,FF,003303	Sign irrelevant
NaN	FF835F00	1,FF,035F00	Sign irrelevant

Figure 4-1 (Part 2 of 2). Selected Single-Precision Floating-Point Numbers

Value	Hexadecimal Representation	Notes
+0.	00000000,00000000	
+1.	3FF00000,00000000	
-1.	BFF00000,00000000	
+2.	40000000,00000000	
+3.	40080000,00000000	
PI	400921FB,54524550	
-(2**(-1029))	80002000,00000000	Denormalized
2**(-1074)	00000000,00000001	Smallest denormalized
approximately ±2.0D-308	7FEFFFFF,FFFFFFFF	Most positive normalized

Figure 4-2 (Part 1 of 2). Selected Double-Precision Floating-Point Numbers

Value	Hexadecimal Representation	Notes
approximately ±2.0D+308	FFEFFFFF,FFFFFFFF	Most negative normalized
2**(-1022)	00100000,00000000	Smallest normalized
+infinity	7FF00000,FFFFFFFF	
-infinity	FFF00000,00000000	
NaN	7FF00500,00009090	Sign irrelevant — any nonzero mantissa
NaN	FFF03333,78433333	Sign irrelevant — any nonzero mantissa

Figure 4-2 (Part 2 of 2). Selected Double-Precision Floating-Point Numbers

Extreme Values and Denormalized Numbers

Extreme floating-point numbers can be classified as zero, signed infinity, Not-a-Number (NaN), denormalized, or normalized.

Zero is represented by either a 1 (-0) or a 0 (+0) in the sign bit position. Negative zero is treated the same as positive zero.

Signed infinity values are usually generated by arithmetic overflows and are represented by a sign bit of 1 (-infinity) or 0 (+infinity), an exponent field of all ones, and a mantissa field of all zeros. When infinity is printed, all the digits in the field are replaced with plus signs (+) for positive infinity and minus signs (-) for negative infinity.

NaN values are generated when invalid arithmetic is attempted. NaN values are represented by an exponent field of all ones and a nonzero mantissa field. The sign is usually ignored. When a NaN value is printed, all the digits in the field are replaced with question marks (?).

Denormalized numbers represent very small positive and negative numbers that result from gradual underflow. The value represented by a denormalized number is determined by this formula:

value = [(-1)**s] * [2**(-bias+1)] * [mantissa/(2**manwidth)]

The value represented by a normalized number is determined by this formula:

value = [(-1)**s] * [2**(expfld - bias)] * [1 + mantissa/(2**manwidth)]

Normalized numbers are often thought of as containing a hidden bit. This hidden bit is the 1 in the preceding formula that is added to the scaled mantissa to generate the value represented. To understand the reason for this, you need to understand the process of normalization:

> Unnormalized numbers are generated as intermediate results during most floating-point operations, and they must be normalized before they can be processed further. Normalization of an unnormalized number consists of repeatedly shifting the mantissa left or right with the corresponding decrement or increment, respectively, of the exponent field. This process is repeated until the most significant "on" bit of the mantissa is in the most significant bit of the mantissa field. At this point, one more shift left is performed along with a corresponding decrement of the exponent field. The leading "on" bit of the mantissa is lost and therefore not represented explicitly.

Denormalized numbers may be thought of as "unnormalizable" because the exponent field is already so small that the left-shift decrement cannot be performed. Consequently, denormalized numbers do not have a hidden bit.

Arithmetic Operations on Extreme Values

This section describes the results derived from applying the basic arithmetic operations and some of the special functions such as square root, sine, and logarithms to combinations of extreme values and ordinary values. No interceptions or other actions take place when extreme values are generated.

In figures 4-3 through 4-10, the inputs are assumed to be positive. Overflow, underflow, and cancellation are assumed not to happen. The meanings of the abbreviations in the tables are:

Den	Denormalized number
Num	Normalized number
Inf	Infinity (+ or -)
NaN	Not-a-Number
Uno	Unordered

Addition and Subtraction					
Left Operand	Right Operand				
	0	Den	Num	Inf	NaN
0	0	Den	Num	Inf	NaN
Den	Den	Den	Num	Inf	NaN
Num	Num	Num	Num	Inf	NaN
Inf	Inf	Inf	Inf	Note	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Note: Inf + Inf = Inf; Inf - Inf = NaN

Figure 4-3. Addition and Subtraction of Extreme Values

Multiplication					
Left Operand	Right Operand				
	0	Den	Num	Inf	NaN
0	0	0	0	NaN	NaN
Den	0	0	Num	Inf	NaN
Num	0	Num	Num	Inf	NaN
Inf	NaN	Inf	Inf	Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Figure 4-4. Multiplication of Extreme Values

Division					
Left Operand	Right Operand				
	0	Den	Num	Inf	NaN
0	NaN	0	0	0	NaN
Den	Inf	Num	Num	0	NaN
Num	Inf	Num	Num	0	NaN
Inf	Inf	Inf	Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Figure 4-5. Division of Extreme Values

Comparison					
Left Operand	Right Operand				
	0	Den	Num	Inf	NaN
0	=	<	<	<	Uno
Den	>		<	<	Uno
Num	>	>		<	Uno
Inf	>	>	>		Uno
NaN	Uno	Uno	Uno	Uno	Uno

Note: NaN compared with NaN is unordered and also results in inequality. +0 equals -0.

Figure 4-6. Comparison of Extreme Values

Maximum					
Left Operand	Right Operand				
	0	Den	Num	Inf	NaN
0	0	Den	Num	Inf	NaN
Den	Den	Den	Num	Inf	NaN
Num	Num	Num	Num	Inf	NaN
Inf	Inf	Inf	Inf	Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Figure 4-7. Maximum of Extreme Values

Minimum					
Left Operand	Right Operand				
	0	Den	Num	Inf	NaN
0	0	0	0	0	NaN
Den	0	Den	Den	Den	NaN
Num	0	Den	Num	Num	NaN
Inf	0	Den	Num	Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Figure 4-8. Minimum of Extreme Values

Operand	Function				
	ATN	EXP	LN/LOG	SQRT	TRIG
-Inf	-PI/2	0	NaN	NaN	NaN
-Num	Num	*	NaN	NaN	**
0	0	1	-Inf	0	0 or 1
+Num	Num	***	Num	Num	**
+Inf	PI/2	+Inf	+Inf	+Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN

* Result can be 0 or a number less than 1.0.
** Result can be 0, a number, +Inf, -Inf, or (if the magnitude of the number is not less than 65536) NaN.
*** Result can be a number greater than 1.0 or +Inf.

Figure 4-9. Special Functions on Extreme Values

X to I FUNCTION					
Operand	Integer Power				
	Neg/ Odd	Neg/ Even	0	Pos/ Even	Pos/ Odd
-Inf	0	0	NaN	+Inf	-Inf
-Num	*	**	1.0	**	*
0	+Inf	+Inf	NaN	0	0
+Num	**	**	1.0	**	**
+Inf	0	0	NaN	+Inf	+Inf
NaN	NaN	NaN	NaN	NaN	NaN

* Result can be 0, a negative number, or -Inf.
** Result can be 0, a positive number, or +Inf.

Figure 4-10. X to I Function on Extreme Values

Representation of Logical Data

All logical variables assume either a ".TRUE." value or a ".FALSE." value. A ".TRUE." value is represented by zeros in all the bit positions except the least significant bit position. A ".FALSE." value is represented by zeros in all the bit positions. Any other bit pattern in a logical variable represents an undefined value.

Representation of Character Data

RT PC VS FORTRAN uses the ASCII representation for characters and control codes. Appendix B, "ASCII Character Set" shows the correspondence between values stored in a byte and the character or control code that the value represents.

A variable declared as CHARACTER**n* occupies *n* bytes of storage aligned on a word boundary. The first character of a character string is stored in the byte with the lower address, the second character is stored in the next higher address, and so on.

Storage of Arrays

FORTRAN array data is organized in computer memory by column (column major order); therefore, the first subscript in a multi-dimensional array varies fastest. For example, an array dimensioned as `X(3,2)` is stored in this order: `X(1,1)`, `X(2,1)`, `X(3,1)`, `X(1,2)`, `X(2,2)`, and `X(3,2)`.

Alignment

The microprocessor in the RT PC has two types of instructions for accessing memory. Memory access can be done for 1 byte (load and store character instructions) or 1 word (4-byte load and store instructions). Because of memory-access limitations, FORTRAN data types (including arrays and common blocks) that require more than 2 bytes of storage are always aligned on a word (4-byte) boundary.

FORTRAN always passes parameters by reference (address). When calling routines written in other languages, such as Pascal or C, care should be taken to ensure that those routines expect addresses instead of value parameters.

Difficulties can also arise from alignment issues when mixing language procedures. For instance, an addressing exception can occur if the address of a character in C is not on a word boundary and is passed to a FORTRAN routine that expects the address of an integer, which always must be on a word boundary. It is imperative that care be taken to ensure proper alignment when mixing language calls.

Chapter 5. Mixing Languages

The RT PC language system permits the mixing of elements from different languages in a single program. This chapter assumes you are familiar with the languages you wish to mix; the elements of the languages are not described here in detail.

Note: In this chapter, the FORTRAN language described is IBM RT PC VS FORTRAN; the Pascal language is IBM RT PC VS Pascal; the C language is IBM RT PC C.

Correspondence of Data Types

The data types of one language are usually quite different from the data types of another language. Also, the way data is stored is not the same across languages; the internal data representation is left unspecified and usually varies with the implementation.

However, a certain amount of similarity among the data types of the different languages exists since the languages share many system primitives and since IEEE standard data representations are used as much as possible. Figure 5-1 on page 5-2 shows some of the correspondence among languages.

Note: Figure 5-1 shows how the languages represent data internally in the computer's memory rather than how data are passed between program units.

FORTRAN IBM and R1 Modes	**FORTRAN VX Mode**	**Pascal**	**C**
LOGICAL*1	LOGICAL*1	BOOLEAN	
	LOGICAL*2		
LOGICAL*4	LOGICAL*4		
INTEGER*2	INTEGER*2		short
INTEGER*4	INTEGER*4	INTEGER	int
REAL*4	REAL*4	SHORTREAL	float
REAL*8	REAL*8	REAL	double
COMPLEX	COMPLEX		
COMPLEX*8	COMPLEX*8		
COMPLEX*16	COMPLEX*16		
CHARACTER	CHARACTER	packed array of CHAR	char
		STRING	

Figure 5-1. Correspondences of Data Types Among Languages

As Figure 5-1 shows, each language has data types that do not exist in the other languages. When you interface languages, make sure you either avoid mismatching data types or use the mismatches very cautiously. When data types do correspond, the interfacing of the languages is very straightforward.

Most numeric data types have counterparts across the languages. However, character and string data types do not. The most difficult aspect of language interfacing is the passing of character, string, or text variables between languages.

FORTRAN's only character variable type is CHARACTER, which is stored as a set of contiguous bytes, one character per byte. The length of a FORTRAN character variable or character array element is determined at compile time and is therefore static. Character lengths are returned by the FORTRAN intrinsic function LEN.

Pascal's character-variable data types are STRING and packed array of CHAR. The STRING data type has a 4-byte word-aligned string length followed by a set of contiguous bytes, one character per byte. The dynamic length of the string can be determined using the **length** function. Packed array of CHAR, however, like FORTRAN's CHARACTER type, is stored as a set of contiguous bytes, one character per byte.

C character data is typically stored as arrays of type "char". The "char" data type stores one character per byte; therefore, an array of "char" is stored exactly like a FORTRAN CHARACTER variable or a Pascal packed array of CHAR.

Storage of Matrices

FORTRAN matrices are stored in computer memory by column (column major order); therefore, the first subscript in a multi-dimensional array varies fastest. An array dimensioned as `A(3,2)` is stored in this order: `A(1,1)`, `A(2,1)`, `A(3,1)`, `A(1,2)`, `A(2,2)`, and `A(3,2)`.

Pascal and C matrices are stored in computer memory by row (row major order). For example, if an array in Pascal is declared as `A : array [1..3,1..2] of REAL`, it is stored in this order: `A[1,1]`, `A[1,2]`, `A[2,1]`, `A[2,2]`, `A[3,1]`, and `A[3,2]`.

Since the matrix storage convention for Pascal and C differs from that for FORTRAN, be careful when passing references to matrices between FORTRAN and the other languages.

Input/Output Primitives

Primitive input/output routines are usually bound to a user's program during the final linking process when the AIX linker includes the necessary code from the language run-time library ("libvsfor.a" for FORTRAN), and the system run-time library ("libvssys.a" for FORTRAN and Pascal).

When you mix, for example, FORTRAN and Pascal, you must remember to link both "libvsfor.a" and "libvssys.a" in this order. This allows the primitives needed for both the FORTRAN and Pascal parts of the program to be present.

The input/output primitives are different for each language; because of this you are not able, for example, to open a file or device for use by one language and write to it or read from it in a different language. Generally, however, two languages can exist in a program as long as they each have their own files and device for input/output, which includes the console device.

Calling from a Non-RT PC VS Main Program: When the main program is compiled using a non-RT PC VS compiler, special handling is required when calling RT PC VS Pascal and FORTRAN subroutines that perform input/output.

An initialization routine must first be called from the main program so that the input/output buffers and information are properly set up: for RT PC VS Pascal, the routine is "vs_pio" and is in the system run-time library "libvssys.a"; for RT PC VS FORTRAN, the routine is "vs_fio_" and is in the language run-time library "libvsfor.a". These routines do not require parameters.

Note: When using RT PC FORTRAN languages, the trailing underscore is automatically appended to the routine name; therefore the name to be coded is "vs_fio".

When the main program is compiled using RT PC VS Pascal or FORTRAN but the subroutine to be called is not, you need to first become familiar with the requirements of that particular subroutine.

Subroutine Linkage Convention

The "subroutine linkage convention" describes the machine state at subroutine entry and exit. This scheme allows routines that are compiled separately in the same or a different RT PC language to be linked and executed when called.

Load Module Format

The load module format used is AIX GPOFF (General Purpose Output File Format). For the GPOFF, each routine has a "constant pool" in the data segment. A constant pool is a data area created for each routine. The first word of each routine's constant pool contains the address of the routine's entry point. A constant pool also provides the routine with addressability to constants, local data, and any called-routine's constant pool. A constant pool pointer (cpp) is passed in register 0 on a call.

Register Usage

If a register is not saved during the call, its contents may be changed during the call. Conversely, if a register is saved, its contents are not changed, and the register can be used as "scratch" (that is, as a work area). Figure 5-2 lists registers and their functions.

Register	Name	Saved During Call	Use
0	called cpp	no	Constant pool pointer. On call, contains address of called routine's constant pool. Can also be used for scratch between calls.
1	fp	yes	Stack pointer
2	--	no	On call, first word of parameter words to called routines. On return, first word of return value. Between calls, can be used as scratch.
3	--	no	On call, second word of parameter words to called routines. On return, second word of return value (for example, low-order 2 words of a floating-point value). Between calls, can be used as scratch.
4	--	no	On call, third word of parameter words to called routines. Between calls, can be used as scratch.
5	--	no	On call, fourth word of parameter words to called routines. Between calls, can be used as scratch.
6	--	yes	Not involved in call interface. Can contain register variables or can be used as scratch.

Figure 5-2 (Part 1 of 2). Register Usage

Register	Name	Saved During Call	Use
7	--	yes	Not involved in call interface. Can contain register variables or can be used as scratch.
8	--	yes	Not involved in call interface. Can contain register variables or can be used as scratch.
9	--	yes	Not involved in call interface. Can contain register variables or can be used as scratch.
10	--	yes	Not involved in call interface. Can contain register variables or can be used as scratch.
11	--	yes	Not involved in call interface. Can contain register variables or can be used as scratch.
12	--	yes	Not involved in call interface. Can contain register variables or can be used as scratch.
13	--	yes	Frame pointer
14	current cpp	yes	Not involved in call interface. By convention, however, contains address of current routine's constant pool.
15	link	no	On call, contains return address. Can also be used as scratch.

Figure 5-2 (Part 2 of 2). Register Usage

Stack Frame

When a routine is called, the compiler passes parameter words 5 through *n* onto the stack. Space is allocated for parameter words 1 through 4. If the routine uses local or temporary variables, they are allocated space on the stack. The stack grows from higher addresses to lower addresses. A single frame-pointer register (register 13) is used to address local storage, incoming and outgoing parameters, and the save area.

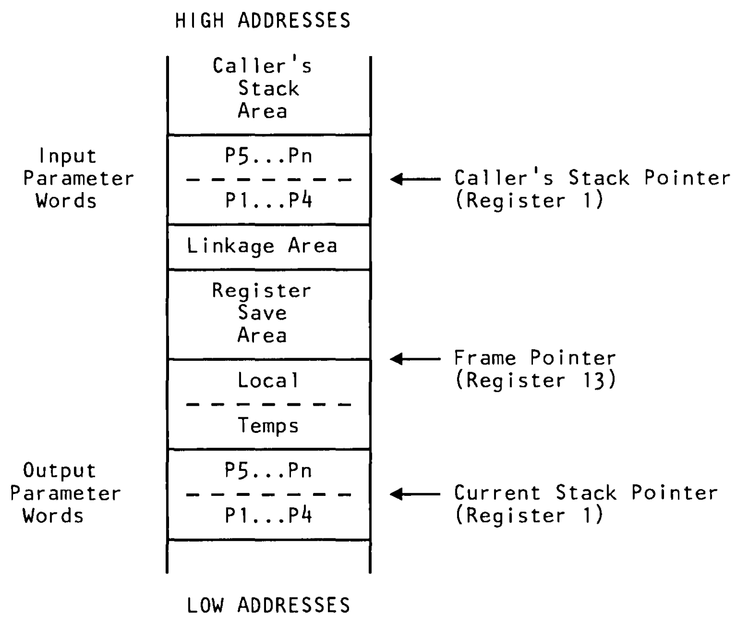

Figure 5-3. Contents of a Stack Frame

Figure 5-3 represents the contents of a stack frame. The areas in the stack are described as follows:

- Input parameter words
- Linkage area
- Register save area
- Local and temporary stack area
- Frame pointer
- Output parameter words
- Total stack frame

Input Parameter Words: If a routine receives more than 4 parameter words, the stack pointer (register 1) upon entry addresses the locations in the stack where parameter words 5 through *n* are stored. Immediately below the stack pointer is a 4-word area in which the first 4 parameter words (passed in registers 2 through 5) can be stored by the compiler. The parameter words are stored only if registers 2 through 5 are to be used as scratch registers or if a parameter address is required. This area is present regardless of the number of parameters being passed.

```
regparsize = 16       # Size of area in which first 4
                      # parameter words can be stored.
```

Linkage Area: The first word of the linkage area is reserved for storing the environment pointer, which is the frame pointer (register 13) of the routine in which the current routine is nested. It is used to gain addressability to the enclosing routine's local variables. Internal calls to nested routines do not use this pointer. It is required to support Pascal parametric procedures or functions since they may be called from a separately compiled routine.

```
envirsize = 4               # Environment pointer size
```

The next 4 words of the linkage area are reserved.

```
resrvsize = 16              # Reserved area size
linksize = envirsize +      # Link area size
     resrvsize
```

Register Save Area: The general-purpose registers (GPRs) and floating-point registers (FPRs) are saved in the register save area. GPR 15 is always saved in the highest word of the register save area. Floating-point registers are saved immediately following the GPRs.

```
Rn                            # First GPR saved (6 <= Rn <= 15)
GPRsize = 4*(16-Rn)           # GPR save area size
save = regargsize +           # Offset of GPR save area
       linksize + GRPsize
FPRsize = 4*163               # FPR save area size
savesize = GRPsize +          # Total register save area
       FPRsize
```

Local and Temporary Stack Area: When a routine needs space for local or temporary variables, the compiler allocates space for them in the local and temporary stack area. The size of this area is known at compile time.

```
set localsize      # Size of local auto's and temp's
```

Frame Pointer: The compiler uses register 13 as the frame pointer to address sections in the stack frame. The register save area, linkage area, and input parameter words are referenced as positive offsets to register 13. The local and temporary variables are referenced as negative offsets to register 13. Output parameter words are referenced using register 1, the current stack pointer.

Output Parameter Words: If a routine makes a call with more than 4 parameter words, the compiler allocates space for the parameter extension list immediately above the stack pointer. This area is large enough to hold the biggest parameter extension list for any call made by the routine.

```
extlistsize      # Size of biggest parameter extension list
```

Total Stack Frame: The entire stack frame can be thought of as including all the space between the caller's stack pointer and the current stack pointer. It is also reasonable to consider the input parameter area as being part of the current stack frame. In a sense, each parameter area belongs to both the caller's stack frame and the current stack frame. In either case, the stack frame size is best defined as the difference between the caller's stack pointer and the current stack pointer.

```
framesize = regargsize +          # stack frame size
  linksize + savesize +
  localsize + extlistsize
```

Parameter Passing

The contents of the parameter words vary among languages. Parameters are understood to occupy an array in the stack, with each parameter aligned on a word boundary. The compiler allocates space in the stack for all the parameter words, but it does not store the first 4 words on the stack. These values are passed in registers 2 through 5. They are only copied to the stack space if a parameter address is required or if registers 2 through 5 are to be used as scratch registers.

Parameter values are passed according to type:

- A type value less than or equal to 4 bytes is passed right-justified in a single word or register, word aligned.
- A procedure or function parameter is passed as a pointer to the routine's constant pool. The routine's environment pointer is also passed.
- A double value is passed in two successive words, which need not be doubleword aligned. One may be in register 5 and the other in the stack frame.

Function Values

Functions return their values according to type:

- A type value less than or equal to 4 bytes is returned right-justified in register 2.
- A double value is returned in registers 2 and 3.

Parameter Addressing

The input parameter words 5 through *n* can be addressed in the stack by:

```
linksize + savesize+4*k-4(r13)   # get k-th parameter word
```

If the compiler stored the first 4 parameter words (registers 2 through 5) in the stack frame, then they can be addressed the same way.

Traceback

The compiler supports the traceback mechanism, which is required by the AIX Operating System Symbolic Debugger in order to unravel the call/return stack. Each module has a traceback table in the text segment at the end of its code. This table contains information about the module including the type of module as well as stack frame and register information.

Entry and Exit Code

The compiler adds entry and exit code around each routine's code, which sets up and removes the routine's stack frame.

The entry code:

- saves modified non-volatile registers
- decreases stack pointer (register 1) by `framesize`
- copies the constant pool pointer from register 0 to register 14
- sets frame pointer (register 13); if the routine is the main program, register 13 points to the global data area.

The exit code:

- restores stack pointer (register 1)
- restores registers.

Calling a Routine

A routine has two symbols associated with it: a constant pool pointer (*_name*) and an entry point (*.name*). When a call is made to a routine, the compiler branches to the *.name* entry point directly and loads the *_name* constant pool pointer into register 0. If the routine entry point is not within a megabyte of the call, the compiler loads the *_name* constant pool pointer, loads the *.name* entry point from the first word of the constant pool, and branches to it.

Sample Programs

The following sample programs show ways to connect program units written in different languages. They also illustrate the mechanisms for passing character, integer, and floating-point variables between Pascal, FORTRAN, and C.

In covering these three variable types, an example of each language calling the other languages is given. The sample programs are included only to illustrate the mixing of the languages and do not show all types of parameter passing.

The lists of AIX commands needed to run the sample programs illustrate that the FORTRAN and Pascal source files must be compiled to ".o" files and then linked together (in these cases, along with C files) to produce an executable file. This can be done by executing each pass of the compiler separately and stopping when the ".o" file is produced, or by interrupting the shell script before the link ("cc") step.

FORTRAN Calling Pascal and C

This example illustrates the passing of FORTRAN CHARACTER, INTEGER, REAL, and DOUBLE PRECISION data to a Pascal procedure and a C function. The source code for each program unit and the AIX commands needed to run the program are shown, as well as a sample run of the program.

Note: The FORTRAN compiler appends an underscore (_) to external symbols. Thus the name of the FORTRAN routine calling Pascal would be "fcallp" but the called Pascal routine must be named "fcallp_" for linkage resolution.

The Calling FORTRAN Program

```
C     This FORTRAN code is in the file named "forexam.f".
C

      PROGRAM EXAMPLE
      CALL MYCHOICE
      WRITE(*,100)
100   FORMAT(' I've safely returned after doing all that!')
      END


      SUBROUTINE MYCHOICE
      INTEGER IFOR
      REAL XFOR
      DOUBLE PRECISION YFOR
      CHARACTER*10 CHRFOR

      EQUIVALENCE (CHRFOR,LETTER)
      CHRFOR='HELLO'
      IFOR=50
      XFOR=10.
C     Some data is initialized.
      YFOR=0.
      WRITE(*,100) CHRFOR,IFOR,XFOR,YFOR
100   FORMAT(/'Before calls:'/' Text string: *'A'*'
     +      /' IFOR='I10/' XFOR='F10.2/' YFOR='F10.2)

C     A Pascal procedure is called.
      CALL PSUB(CHRFOR,IFOR,XFOR,YFOR)
      WRITE(*,110) CHRFOR,IFOL,XFOR,YFOR
110   FORMAT(/'After Pascal call:'/' Text string: *'A'*'
     +      /' IFOR='I10/' XFOR='F10.2/' YFOR='F10.2)

C     A C subroutine is called.
      CALL CSUB(CHRFOR,IFOR,XFOR,YFOR)
      WRITE(*,120) CHRFOR,IFOR,XFOR,YFOR
120   FORMAT(/'After C call:'/' Text string: *'A'*'
     +      /' IFOR='I10/' XFOR='F10.2/' YFOR='F10.2)

      END
```

The Called Pascal Procedure

```
{  This is the Pascal procedure to be called by FORTRAN.
   This code is in the file named "fcallp.pas".  }

segment DUMMYNAME;

       type TEXT = packed array [0..79] of CHAR;

       procedure PSUB_(var WORDS : TEXT;
                           COUNT : INTEGER;
                       var I     : INTEGER;
                       var X     : SHORTREAL;
                       var Y     : REAL);external;

       procedure PSUB_;
         begin
          WORDS[0] := 'B'; WORDS[1] := 'Y'; WORDS [2] := 'E';
          WORDS[3] := ' '; WORDS[4] := ' ';
                 X := X * I;
                 I := COUNT;
                 Y := 1.0d0;
          end;
```

The Called C Function

```
/*   This is the C function to be called by FORTRAN.      */
/*   This code is in the file named "fcallc.c".           */

/*   Note underscore in procedure declaration.            */

int csub_ (word,  /* C subprograms are functions, but the */
           count, /* value returned can be ignored.       */
             i,
             x,
             y)


char word[79];
int count;
int *i;
float *x;
double *y;
```

```
{
    word[0] = 'h';     /* Arrays in C always use the   */
    word[1] = 'i';     /* call-by-reference mechanism. */
    word[2] = ' ';
    word[3] = 'C';
    word[4] = ' ';
    *i = -3;
    *x = -(*x);
    *y = 2.0;
    return (0);        /* A zero is returned which may  */
                       /* be ignored.                   */
}
```

Commands and Output

The AIX commands needed to run this sample program are:

```
vsfort forexam.f
vspass2 forexam.i
vspass3 forexam.obj
vspascal fcallp.pas
vspass2 fcallp.i
vspass3 fcallp.obj
cc -o forexam forexam.o fcallp.o fcallc.c -lm
   /usr/lib/libvsfor.a /usr/lib/libvssys.a
forexam
```

The output from running this sample program is:

```
Before calls:
 Text string: *HELLO     *
 IFOR=        50
 XFOR=     10.00
 YFOR=       .00

After Pascal call:
 Text string: *BYE       *
 IFOR=        10
 XFOR=    500.00
 YFOR=      1.00

After C call:
 Text string: *hi C      *
 IFOR=        -3
 XFOR=   -500.00
 YFOR=      2.00
 I've safely returned after doing all that!
```

Pascal Calling FORTRAN and C

This example illustrates the passing of Pascal CHAR, INTEGER, REAL, and DOUBLE data to a FORTRAN subroutine and a C function. The source code for each program unit and the AIX commands needed to run the program are shown, as well as a sample run of the program.

Note: The FORTRAN compiler appends an underscore (_) to external symbols. Thus the Pascal external declaration must have an underscore appended to the FORTRAN name.

The Calling Pascal Program

```
{  This code is in the file named "pasexam.pas".  }

program MAIN (input,output);
   type TEXT = packed array [0..79] of CHAR;

{ Note underscore in FORTRAN procedure declaration }
   procedure FSUB_ (var NAMES : TEXT;
                    COUNT     : INTEGER;
                    var IPAS  : INTEGER;
                    var XPAS  : SHORTREAL;
                    var YPAS  : REAL    ); external;

   procedure CSUB  (    NAMES : TEXT;
                    var IPAS  : INTEGER;
                    var XPAS  : REAL;
                        YPAS  : REAL    ); external;

   procedure PSUB;
    var
     CHRPAS : TEXT;
     INTPAS : INTEGER;
     XREAL  : SHORTREAL;
     YDOUB  : REAL;
     I      : INTEGER;

   begin
     CHRPAS[0] := 'H'; CHRPAS[1] := 'I'; CHRPAS[2] := ' ';
     CHRPAS[3] := 'W'; CHRPAS[4] := 'I'; CHRPAS[5] := 'R';
     CHRPAS[6] := 'T'; CHRPAS[7] := 'H';
     INTPAS    := 50;
     XREAL     := 10.0;
     YDOUB     := 0.0d0;
```

```
      writeln;
      writeln ('Before calls:');
      write(' Text: *'); for I := 0 to 7 do write(CHRPAS[I]);
      writeln('*');
      writeln(' INTPAS=',INTPAS,'  XREAL=',XREAL,'  YDOUB=',YDOUB);
      FSUB_(CHRPAS,20,INTPAS,XREAL,YDOUB);
      writeln;
      writeln ('After FORTRAN call:');
      write(' Text: *'); for I := 0 to 20 do write(CHRPAS[I]);
      writeln('*');
      writeln(' INTPAS=',INTPAS,'  XREAL=',XREAL,'  YDOUB=',YDOUB);
      CSUB(CHRPAS,INTPAS,XREAL,YDOUB);
      writeln;
      writeln ('After C call:');
      write(' Text: *'); for I := 0 to 4 do write(CHRPAS[I]);
      writeln('*');
      writeln(' INTPAS=',INTPAS,'  XREAL=',XREAL,'  YDOUB=',YDOUB);
    end;

begin
 writeln('This message is printed at the beginning of MAIN.');
 PSUB;
 writeln('This message is printed at the end of MAIN.')
end.
```

The Called FORTRAN Subroutine

```
C     This is the FORTRAN subroutine to be called by Pascal.
C     This code is in the file named "pcallf.f".

      SUBROUTINE FSUB(CHR,I,X,Y)
      CHARACTER*20 CHR
      INTEGER I
      REAL X
      I=LEN(CHR)
      CHR='FORTRAN Lives!'
      X=X*I
      Y=1.0D0
      RETURN
      END
```

The Called C Function

```
/*     This is the C function to be called by Pascal.      */
/*     This code is in the file named "pcallc.c".          */

int CSUB (word,   /* C subprograms are functions, but the  */
            i,    /* value returned can be ignored.        */
            x,
            y  )

char word[79];
int *i;
float *x;
double *y;

    {
        word[0] = 'h';    /* Arrays in C always use the   */
        word[1] = 'i';    /* call-by-reference mechanism. */
        word[2] = ' ';
        word[3] = 'C';
        word[4] = ' ';
        *i = -3;
        *x = -1.0;
        *y =  1.0;
        return (0);   /* A zero is returned which may be   */
                      /* ignored if CSUB is treated as a  */
                      /* procedure, or used if CSUB is    */
    }                 /* treated as a function.           */
```

Commands and Output

The AIX commands needed to run this sample program are:

```
vspascal pasexam.pas
vspass2 pasexam.i
vspass3 pasexam.obj
vsfort pcallf.f
vspass2 pcallf.i
vspass3 pcallf.obj
cc -o pasexam pasexam.o pcallf.o pcallc.c -lm
  /usr/lib/libvsfor.a /usr/lib/libvssys.a
pasexam
```

The output from running this sample program is:

```
This message is printed at the beginning of MAIN.

Before calls:
 Text: *HI WIRTH*
 INTPAS=50  XREAL= 1.00000E+01  YDOUB= 0.00000000000000D+000

After FORTRAN call:
 Text: *FORTRAN lives!      *
 INTPAS=20  XREAL= 2.00000E+02  YDOUB= 1.00000000000000D+000

After C call:
 Text= *hi C *
 INTPAS=-3  XREAL=-1.00000E+00  YDOUB= 1.00000000000000D+000

This message is printed at the end of MAIN.
```

C Calling FORTRAN and Pascal

This example illustrates passing C-language char, int, float, and double data to a FORTRAN subroutine and a Pascal procedure. The source code for each program unit and the AIX commands needed to run the program are shown, as well as a sample run of the program.

Note: The FORTRAN compiler appends an underscore (_) to external symbols. Thus the C external declaration must have an underscore appended to the FORTRAN name.

The Calling C Program

```
/*  This code is in the file named "cexam.c".  */

#include <stdio.h>
 main ()
 {
   printf("\n This message is printed at the start of MAIN.");
   cfunc ();
   printf("\n This message is printed at the end of MAIN.");
 }
```

```
cfunc ()
{
  char chrc[79];
  int ic, count;
  float xc;
  double yc;

  chrc[0] = 'h'; chrc[1] = 'i'; chrc[2] = ' ';
  chrc[3] = 'C'; chrc[4] = '\0';
  ic = 50; xc = 10.; yc = 0.0;
  count=10;

  printf("\n Before calls:");
  printf("\n  Text string: %s", chrc);
  printf("\n  ic = %d", ic);
  printf("\n  xc = %f", xc);
  printf("\n  yc = %f", yc);

  fsub_(chrc,count,&ic,&xc,&yc);     /* Arrays in C always use */
  printf("\n After FORTRAN call:"); /* the call-by-reference  */
  printf("\n  Text string: %s", chrc); /* mechanism.          */
  printf("\n  ic = %d", ic);
  printf("\n  xc = %f", xc);
  printf("\n  yc = %f", yc);

  psub(chrc,&ic,&xc,&yc);
  printf("\n After Pascal call:");
  printf("\n  Text string: %s", chrc);
  printf("\n  ic = %d", ic);
  printf("\n  xc = %f", xc);
  printf("\n  yc = %f", yc);
  }
```

The Called FORTRAN Subroutine

```
C     This is the FORTRAN subroutine to be called by C.
C     This code is in the file named "ccallf.f".

      SUBROUTINE FSUB(WORDS,I,X,Y)
C     FORTRAN uppercases all globals.  Note that the order of
C     the parameters is the order in the calling program unit.
      CHARACTER*80 WORDS
      INTEGER I
      REAL X
      DOUBLE PRECISION Y
      WORDS='FORTRAN lives!'//Char(0)
C     The string terminator C expects is concatenated.
      I=LOG(X)
      X=LOG(X)
      Y=45.D0
      RETURN
      END
```

The Called Pascal Procedure

```
{  This is the Pascal procedure to be called by C.
   This code is in the file named "ccallp.pas".  }

segment DUMMYNAME;

     type TEXT = packed array [0..79] of CHAR;

     procedure PSUB (var WORDS : TEXT;
                     var I     : INTEGER;
                     var X     : SHORTREAL;
                     var Y     : REAL  );external;

     procedure PSUB;
        begin
          WORDS[0] := 'G'; WORDS[1] := 'o'; WORDS[2] := ' ' ;
          WORDS[3] := 'W'; WORDS[4] := 'i'; WORDS[5] := 'r' ;
          WORDS[6] := 't'; WORDS[7] := 'h'; WORDS[8] := '!' ;
          WORDS[9] := chr(0);  { C character string terminator }
                 X := 2*X;
                 I := 2*I;
                 Y := 4.0d0;
        end;
```

Commands and Output

The AIX commands needed to run this sample program are:

```
vspascal ccallp.pas
vspass2 ccallp.i
vspass3 ccallp.obj
vsfort ccallf.f
vspass2 ccallf.i
vspass3 ccallf.obj
cc -o cexam cexam.c ccallp.o ccallf.o -lm
   /usr/lib/libvsfor.a /usr/lib/libvssys.a
cexam
```

The output from running this sample program is:

```
This message is printed at the start of main.


Before calls:
 Text string: hi C
 ic= 50
 xc= 10.000000
 yc= 0.000000

After FORTRAN call:
 Text string: FORTRAN lives!
 ic= 2
 xc= 2.302585
 yc= 45.000000

After Pascal call:
 Text string: Go Wirth!
 ic= 4
 xc= 4.605170
 yc= 4.000000
This message is printed at the end of main.
```

Chapter 6. The Disassembler

The Disassembler produces assembly language listings for Pascal and FORTRAN programs. With the Disassembler, binary code modules created by high-level languages can be translated into assembly language equivalents.

The assembly language output includes:

- absolute address listing
- hex code listing
- variable type listing
- variable location listing
- symbolic references to external entry points
- labels indicating high-level language source-line numbers
- indications of high-level language variable storage locations
- disassembly of certain embedded data constructs used in high-level languages.

The Disassembler is flexible and easy to use, and can be executed in a variety of ways to suit your needs.

Preparation

Before the Disassembler can be used, it is necessary to compile the source program with the "d+" option specified on the command line. This option instructs the RT PC VS Pascal and RT PC VS FORTRAN compilers to place additional tables of symbolic information into the binary code, which is consolidated during the compile into a separate file. This file has the same root name as the source file and is given a ".dbg" extension.

The Disassembler can now be executed using the target program (the compiled program).

Automatic Option Memory File

At the beginning of each disassembly session, the Disassembler searches for a file named "dis.cmd". If the file exists in the current directory, the Disassembler uses the contents of this file to set its options. If the file is not found, the default option profile is used. At the end of the disassembly session, all options in effect are written to the file.

Using the Disassembler

The Disassembler can be executed in these ways:

- from the command line with one or more options specified
- from the command line with only the default settings in effect
- from the menu
- from a command file containing Disassembler options.

From the Command Line — with Options

The Disassembler can be invoked from the command line with one or more options specified. The default settings, which may be changed, are:

- output displayed on screen
- no address listing
- no hex code listing
- no variable type listing
- no variable location listing.

The format for running the Disassembler from the command line with one or more options specified is:

```
disasm +i=filename +m=module option [ option ] ...
```

+i=*filename*
specifies an input file that contains the program or submodule to be disassembled.

+m=*module*
specifies the entry point to be disassembled. The entry point is searched for in the symbol table. If symbolic information is available for this entry point, the information is incorporated in the disassembly.

Note: The **#**, when used for the entry point, causes the entire program to be disassembled.

option
may be any of the following:

+a	ABSOLUTE ADDRESS LISTING instructs the Disassembler to include an absolute address listing in the output. The default is no absolute address listing (**-a**).
+d=*filename*	SYMBOLIC DEBUGGER SYMBOLS specifies a file of Symbolic Debugger symbols to be used in the disassembly. The input file name is used by default. The file name extension defaults to ".dbg".
+o=*filename*	OUTPUT FILE FOR DISASSEMBLY specifies an output file to be used for disassembly. The input file name is used by default. The file name extension defaults to ".dis".
+p=*cmdfile*	OPTION FILE specifies a file from which the Disassembler can read its options. The default name is "dis.cmd".
+r	HEX CODE (RAW DATA) LISTING instructs the Disassembler to include a hex code (raw data) listing in the output. The default is no hex code listing (**-r**).

-s NO OUTPUT DISPLAY ON SCREEN
instructs the Disassembler not to display the output on the screen. The default is output displayed on the screen (+s).

+t VARIABLE TYPE LISTING
instructs the Disassembler to include a variable type listing in the output. The default is no variable type listing (**-t**).

+v VARIABLE LOCATION LISTING
instructs the Disassembler to include a variable location listing in the output. The default is no variable location listing (**-v**).

Option	**Function**
+a	Absolute address listing
+d=*filename*	Symbolic debugger symbols
+o=*filename*	Output file for disassembly
+p=*cmdfile*	Option file
+r	Hex code (raw data) listing
-s	No output display on screen
+t	Variable type listing
+v	Variable location listing

Figure 6-1. Disassembler Command-Line Options

The Disassembler command-line options, procedure names, and module names can be in uppercase or lowercase. However, case is significant in the specified file names. File names can be either uppercase or lowercase, but the case has to correspond to the case used in the operating system. For example:

```
disasm +a +i=INFILE +m=MOD +d=DBGFILE +o=OUTFILE.OUT
```

Note: The same command could be executed with a # substituted for the module name (`+m = #`), resulting in the entire program being disassembled.

Output files may use any extension. However, if an extension is not specified, the default extension ".dis" is used. An input file can be a compiled source program or input file that does not have an extension, or the debug file that has a ".dbg" extension.

If the root of the symbol file name and the input file name are the same, only the root of the name needs to be specified, as in the command:

```
disasm f1 SAMPLE
```

Example:

```
C     For this example, SAMPLE is
C     located in file "f1.f".
C     After the program is compiled,
C     the executable file is "f1".
C
      PROGRAM SAMPLE
      INTEGER X
      X = 1
      X = X + 1
      WRITE (6,10) X
   10 FORMAT (I4)
      STOP
      END
```

The following code creates the assembler equivalent of the SAMPLE program and writes the output to the file named "out.dis". The address, hex code, variable type, and variable location listings are omitted. Provided that the symbol file and the input file are created with the same name (f1), the command form is:

```
disasm +i=f1 +m=SAMPLE +o=out
```

By default, the following output is displayed on the screen:

```
                 XDEF     sample
                 XREF     .r_init
                 XREF     .f_init
                 XREF     .f_ixfw
                 XREF     .f_wrfi
                 XREF     .f_tfwr
                 XREF     .f_stop
                 XREF     .f_rtsf
                 XREF     .r_end
*
*
 sample:         STM      R6,$FFB4(R1)
                 AI       R13,R1,$FF74
                 CAL      R1,$FF6C(R1)
                 LR       R14,R0
                 L        R4,$4(R14)
                 AI       R5,R13,$40
                 BALIX    R15,.r_init
                 L        R0,$8(R14)
                 BALIX    R15,.f_init
                 L        R0,$C(R14)
 USERCODE:       LIS      R12,$1
 LN_2:           AIS      R12,$1
 LN_3:           LIS      R2,$1
                 L        R3,$10(R14)
                 LIS      R4,$5
                 BALIX    R15,.f_ixfw
                 L        R0,$18(R14)
                 LR       R2,R12
                 BALIX    R15,.f_wrfi
                 L        R0,$1C(R14)
                 BALIX    R15,.f_tfwr
                 L        R0,$20(R14)
 LN_4:           LIS      R2,$0
                 LIS      R3,$0
                 BALIX    R15,.f_stop
                 L        R0,$24(R14)
                 BALIX    R15,.f_rtsf
                 L        R0,$28(R14)
                 BALIX    R15,.r_end
                 L        R0,$2C(R14)
                 LM       R6,$48(R1)
                 BRX      R15
                 CAL      R1,$94(R1)
                 END
```

The output file "out.dis" contains:

```
* ROMP Disassembled Instruction Code
* Options in effect:
*    Address listing                                 [N]
*    Hex code listing                                [N]
*    Variable type listing                           [N]
*    Variable location listing                       [N]
*
* Image file: f1
* Debug file: f1.dbg
* Module:     sample
*
* Initial address: 100002D0
* Final   address: 10000340
*
                 XDEF     sample
                 XREF     .r_init
                 XREF     .f_init
                 XREF     .f_ixfw
                 XREF     .f_wrfi
                 XREF     .f_tfwr
                 XREF     .f_stop
                 XREF     .f_rtsf
                 XREF     .r_end
*
*
 sample:         STM      R6,$FFB4(R1)
                 AI       R13,R1,$FF74
                 CAL      R1,$FF6C(R1)
                 LR       R14,R0
                 L        R4,$4(R14)
                 AI       R5,R13,$40
                 BALIX    R15,.r_init
                 L        R0,$8(R14)
                 BALIX    R15,.f_init
                 L        R0,$C(R14)
 USERCODE:       LIS      R12,$1
 LN_2:           AIS      R12,$1
 LN_3:           LIS      R2,$1
                 L        R3,$10(R14)
                 LIS      R4,$5
                 BALIX    R15,.f_ixfw
                 L        R0,$18(R14)
                 LR       R2,R12
```

```
                BALIX    R15,.f_wrfi
                L        R0,$1C(R14)
                BALIX    R15,.f_tfwr
                L        R0,$20(R14)
LN_4:           LIS      R2,$0
                LIS      R3,$0
                BALIX    R15,.f_stop
                L        R0,$24(R14)
                BALIX    R15,.f_rtsf
                L        R0,$28(R14)
                BALIX    R15,.r_end
                L        R0,$2C(R14)
                LM       R6,$48(R1)
                BRX      R15
                CAL      R1,$94(R1)
                END
```

From the Command Line — without Options

The Disassembler can be invoked without options. In this case, output is written to the screen and no address, hex code, variable type, or variable location listings are included.

The format for running the Disassembler from the command line without options is:

disasm *input-file-name* *entry-point*

input-file-name
is the file that contains the program or submodule to be disassembled.

entry-point
is the name of the procedure or function to be disassembled.

Both the *input-file-name* and the *entry-point* parameters are required; specifying only one is a syntax error.

The following example uses the SAMPLE program and the executable file "f1". To disassemble SAMPLE, enter:

```
disasm f1 SAMPLE
```

or

```
disasm f1 #
```

The following is displayed on the screen:

```
                 XDEF     sample
                 XREF     .r_init
                 XREF     .f_init
                 XREF     .f_ixfw
                 XREF     .f_wrfi
                 XREF     .f_tfwr
                 XREF     .f_stop
                 XREF     .f_rtsf
                 XREF     .r_end
*
*
 sample:         STM      R6,$FFB4(R1)
                 AI       R13,R1,$FF74
                 CAL      R1,$FF6C(R1)
                 LR       R14,R0
                 L        R4,$4(R14)
                 AI       R5,R13,$40
                 BALIX    R15,.r_init
                 L        R0,$8(R14)
                 BALIX    R15,.f_init
                 L        R0,$C(R14)
 USERCODE:       LIS      R12,$1
 LN_2:           AIS      R12,$1
 LN_3:           LIS      R2,$1
                 L        R3,$10(R14)
                 LIS      R4,$5
                 BALIX    R15,.f_ixfw
                 L        R0,$18(R14)
                 LR       R2,R12
                 BALIX    R15,.f_wrfi
                 L        R0,$1C(R14)
                 BALIX    R15,.f_tfwr
                 L        R0,$20(R14)
```

```
LN_4:           LIS      R2,$0
                LIS      R3,$0
                BALIX    R15,.f_stop
                L        R0,$24(R14)
                BALIX    R15,.f_rtsf
                L        R0,$28(R14)
                BALIX    R15,.r_end
                L        R0,$2C(R14)
                LM       R6,$48(R1)
                BRX      R15
                CAL      R1,$94(R1)
                END
```

From the Menu System

Options can be selected from a system of menus. To invoke the main menu, enter:

```
disasm
```

The following menu appears on the screen:

```
      **** Main Menu ****
      -------------------

1 ... Select input options
2 ... Produce disassembly
3 ... Select output form options
4 ... Select output designation
5 ... Display options in effect

Enter Selection # (or q to Quit): _
```

Any option may be selected from the menu. Press the Enter key after each selection. For example, to display the options currently in effect, select option 5 from the menu. This menu is illustrated in “Display Options

Selection" on page 6-15. The default profile shows the options initially in effect.

Input Options Selection

If option 1 is selected, the "Input Options Menu" appears:

```
   **** Input Options Menu ****
   ----------------------------

1 ... Specify input file name         [ ]
2 ... Specify program or entry point  [ ]
3 ... Specify debug symbol file       [ ]

Enter Selection # (or <Enter> for Main Menu): _
```

Empty brackets at the end of options 1, 2, and 3 indicate that the input options have not yet been selected.

The SAMPLE program resides in input file "f1". To specify input file "f1", enter option 1. The response is:

```
   **** Input Options Menu ****
   ----------------------------

1 ... Specify input file name         [ ]
2 ... Specify program or entry point  [ ]
3 ... Specify debug symbol file       [ ]

Enter Selection # (or <Enter> for Main Menu): 1

Enter input file name
or <Enter> to cancel:            f1
```

No extension is required for the input file; you need to only enter "f1". The updated screen shows:

```
   **** Input Options Menu ****
   ----------------------------

1 ... Specify input file name          [f1]
2 ... Specify program or entry point   [ ]
3 ... Specify debug symbol file        [f1.dbg]

Enter Selection # (or <Enter> for Main Menu): _
```

If SAMPLE is used as the entry point (option 2), the screen is updated once again to include the entry point information. When option 2 is selected, this screen is displayed:

```
   **** Input Options Menu ****
   ----------------------------

1 ... Specify input file name          [f1]
2 ... Specify program or entry point   [ ]
3 ... Specify debug symbol file        [f1.dbg]

Enter Selection # (or <Enter> for Main Menu): 2

Enter target name
(program, or submodule name)
or <Enter> to cancel:                 SAMPLE
```

A # may be used as the entry point for option 2, in which case the entire program is disassembled.

After the input file name (f1) and the program name (SAMPLE) are entered, it is possible to return to the main menu to disassemble the program, using only the Disassembler's default settings, by choosing the "Produce disassembly" selection.

Produce Disassembly Selection

After the selections for the "Input Options Menu" are complete, the Disassembler can be executed using its default settings. It is not necessary to continue through the menus unless changes are to be made to the default settings or to the output designation.

When the "Produce disassembly" option is selected, the screen is cleared and the disassembled output is displayed. Upon completion, this message is displayed:

```
Do you wish to continue? (y/n)
```

If a "y" is entered, the main menu is displayed once again. You can now change the default settings for the output form by selecting option 3, or change the output designation by selecting option 4. By selecting option 1, another program can be disassembled.

If an "n" is entered, the Disassembler program is terminated.

Output Form Options Selection

If option 3 is selected, the "Output Form Options" menu appears:

```
    **** Output Form Options ****
    -----------------------------

 1 ... Address listing                              [n]
 2 ... Hex code listing                             [n]
 3 ... Variable type listing                        [n]
 4 ... Variable location listing                    [n]

 Enter Selection # (or <Enter> for Main Menu): _
```

To change any of the default settings, enter the appropriate option number. For example, to include an address listing, select option 1. The response is:

```
   **** Output Form Options ****
   -----------------------------

1 ... Address listing                                    [n]
2 ... Hex code listing                                   [n]
3 ... Variable type listing                              [n]
4 ... Variable location listing                          [n]

Enter Selection # (or <Enter> for Main Menu): 1

Include address listing?   (y/n)
```

Enter "y" to include an address listing. The updated screen shows that the change has been made.

Output Designation Selection

If option 4 is selected, the "Output Designation Menu" appears:

```
   **** Output Designation Menu ****
   ---------------------------------

1 ... Write output to file            [ ]
2 ... Display output on screen        [y]

Enter Selection # (or <Enter> for Main Menu): _
```

To write to a file, select option 1. The following screen is displayed:

```
    **** Output Designation Menu ****
    ---------------------------------

1 ... Write output to file            [ ]
2 ... Display output on screen        [y]

Enter Selection # (or <Enter> for Main Menu): 1

Enter output file name
or <Enter> to cancel:
```

It is possible to select both options in this menu; the output is then written to a file and displayed on the screen.

Display Options Selection

If option 5 is selected, the "Options in effect" screen is displayed. This screen contains a list of the options that are currently being used by the Disassembler. These options are selected from the "Output Form Options" menu and the "Output Designation Menu".

```
Options in effect:
— Disassemble high level program or
   submodule
   Input file:  f1
   Module:      SAMPLE
   Debug File:  f1.dbg
— Include variable type definitions
— Include variable location listing
— Display output on screen

Press Enter to Continue ...
```

From a Command File

A command file can be created that contains options readable by the Disassembler. The file is created with an editor, and options are entered one option per line.

The following command file example produces pure assembler code with the variable type and location information in comment form in the output file. This command file contains:

```
+i=f1
+m=SAMPLE
+o=out
-s
+t
+v
```

Note: In this example,"+m=SAMPLE" may be replaced with "+m= #" to disassemble the entire program.

If this command file is named "COMMAND.CMD", the Disassembler command is:

```
disasm +p=COMMAND.CMD
```

The output file "out" uses the default extension ".dis". The output file contains:

```
* ROMP Disassembled Instruction Code
* Options in effect:
*    Address listing                                [N]
*    Hex code listing                               [N]
*    Variable type listing                          [Y]
*    Variable location listing                      [Y]
*
* Image file: f1
* Debug file: f1.dbg
* Module:     sample
*
* Initial address: 100002D0
* Final   address: 10000340
*
```

```
                XDEF     sample
                XREF     .r_init
                XREF     .f_init
                XREF     .f_ixfw
                XREF     .f_wrfi
                XREF     .f_tfwr
                XREF     .f_stop
                XREF     .f_rtsf
                XREF     .r_end
*
*
 sample:        STM      R6,$FFB4(R1)
                AI       R13,R1,$FF74
                CAL      R1,$FF6C(R1)
                LR       R14,R0
                L        R4,$4(R14)
                AI       R5,R13,$40
                BALIX    R15,.r_init
                L        R0,$8(R14)
                BALIX    R15,.f_init
                L        R0,$C(R14)
 USERCODE:      LIS      R12,$1
 LN_2:          AIS      R12,$1
 LN_3:          LIS      R2,$1
                L        R3,$10(R14)
                LIS      R4,$5
                BALIX    R15,.f_ixfw
                L        R0,$18(R14)
                LR       R2,R12
                BALIX    R15,.f_wrfi
                L        R0,$1C(R14)
                BALIX    R15,.f_tfwr
                L        R0,$20(R14)
 LN_4:          LIS      R2,$0
                LIS      R3,$0
                BALIX    R15,.f_stop
                L        R0,$24(R14)
                BALIX    R15,.f_rtsf
                L        R0,$28(R14)
                BALIX    R15,.r_end
                L        R0,$2C(R14)
                LM       R6,$48(R1)
                BRX      R15
                CAL      R1,$94(R1)
*
```

```
* offset definitions
*
* entry sample
* user name sample
*
* entry code begins at $100002D0
* user code begins at  $100002F4
* exit code begins at  $10000326
* addresses for source code by line number:
*    1: $100002F4    2: $100002F6    3: $100002F8    4: $1000031A
*
* variable definitions
*
* sample:
*    x              type  -3  r7
*
* type definitions:
*
*
*  -1 = integer (1 byte )
*  -2 = integer (2 bytes)
*  -3 = integer (4 bytes)
*  -4 = unsigned integer (1 byte )
*  -5 = unsigned integer (2 bytes)
*  -6 = unsigned integer (4 bytes)
*  -7 = character (1 byte )
*  -8 = character (2 bytes)
*  -9 = single precision floating point (4 bytes)
* -10 = double precision floating point (8 bytes)
* -11 = logical (1 byte )
* -12 = logical (2 bytes)
* -13 = logical (4 bytes)
* -14 = file
* -15 = complex floating point (16 bytes)
* -16 = double complex floating point (32 bytes)
                 END
```

Appendix A. Messages

Compile-Time Messages

RT PC VS FORTRAN contains a file of compile-time error messages named "vsfctmsg.inc". The compiler generates error numbers and messages if this file is present in the default directory and errors are encountered.

If the l*filename* command-line option is used, any error messages are written to the listing file. If the e*filename* command-line option is used, any error messages are written to the error file. Otherwise, error messages are displayed on the console.

0	Unknown error
1	Fatal error reading source code block
2	Non-numeric characters in label field
3	Too many continuation lines
4	Fatal end-of-file encountered
5	Labeled continuation line
6	Missing field or syntax error on compiler-directive line
7	Compiler directive allows nonstandard feature
8	Unrecognizable compiler directive
9	Input source code file not a valid text file format
10	Maximum depth of INCLUDE file nesting exceeded
11	Integer constant overflow
12	Error in real constant
13	Too many digits in constant
14	Identifier too long
15	Character constant extends to end of line
16	Character constant is zero length
17	Illegal character in input

18	Integer constant expected
19	Label expected
20	Error in label
21	Type name expected (INTEGER[*n], REAL[*n], DOUBLE PRECISION, COMPLEX, LOGICAL[*n], or CHARACTER[*n])
22	INTEGER constant expected
23	Extra characters at end of statement
24	'(' expected
25	Letter IMPLICITed more than once
26	')' expected
27	Letter expected
28	Identifier expected
29	Dimension(s) required in DIMENSION statement
30	Array dimensioned more than once
31	Maximum number of array dimensions exceeded
32	Incompatible arguments to EQUIVALENCE
33	Variable appears more than once in a type specification statement
34	This identifier has already been declared
35	This intrinsic function cannot be passed as an argument
36	Identifier must be a variable
37	Identifier must be a variable or the current FUNCTION name
38	'/' expected
39	Named COMMON block already saved
40	Variable already appears in a COMMON block
41	Variables in different COMMON blocks cannot be equivalenced
42	Number of subscripts in EQUIVALENCE statement does not agree with variable declaration
43	EQUIVALENCE subscript out of range
44	Two distinct cells equivalenced to the same location in a COMMON block
45	EQUIVALENCE statement extends a COMMON block in a negative direction
46	EQUIVALENCE statement forces a variable to two distinct locations, not in a COMMON block
47	Statement number expected
48	Mixed CHARACTER and numeric items not allowed in same COMMON block
49	CHARACTER items cannot be equivalenced to non-character items
50	Illegal symbols in an expression

51 Cannot use subroutine or namelist name in an expression
52 Type of argument must be INTEGER or REAL
53 Type of argument must be INTEGER, REAL, or CHARACTER
54 Types of comparisons must be compatible
55 Type of expression must be LOGICAL
56 Too many subscripts
57 Too few subscripts
58 Variable expected
59 '=' expected
60 Size of equivalenced CHARACTER items must be the same

61 Illegal assignment — types do not match
62 Can only call subroutines
63 Dummy arguments cannot appear in COMMON statements
64 Dummy arguments cannot appear in EQUIVALENCE statements
65 Assumed-size array declarations can only be used for dummy arrays
66 Adjustable-size array declarations can only be used for dummy arrays
67 Assumed-size array dimension specifier, "*", must be the upper bound of the last dimension
68 Adjustable bound must be either a dummy argument or in COMMON prior to appearance
69 Adjustable bound must be simple integer expression containing only constants, COMMON variables, or PARAMETER constant names
70 Cannot have more that one main program

71 The size of a named COMMON block must be the same in all subprograms
72 Dummy arguments cannot appear in DATA statements
73 Variables in blank COMMON cannot appear in DATA statements
74 Names of subroutines, functions, intrinsic functions, and namelists cannot appear in DATA statements
75 Subscripts out of range in DATA statement
76 Repeat count must be integer value greater than zero
77 Constant expected
78 Type conflict in DATA statement
79 Number of variables does not match the number of values in DATA statement list
80 Statement cannot have a label

81 No such intrinsic function
82 Type declaration for intrinsic function does not match actual type of intrinsic function

83	Letter expected
84	Type of function does not agree with previous usage
85	This subprogram has already appeared in this compilation
87	Error in type of argument to intrinsic function
88	Subroutine/function previously used as a function/subroutine
89	Unrecognizable statement
90	Expression not allowed
91	Missing END statement
93	Fewer actual arguments than formal arguments in a function or subroutine reference
94	More actual arguments than formal arguments in a function or subroutine reference
95	Type of actual argument does not agree with formal argument
96	The following procedures were called but not defined
98	Size of type CHARACTER must be consistent with the number in 'n' option or @PROCESS CL(nnn) statement
99	INTEGER*4 variable required
100	Statement out of order
101	Unrecognizable statement
102	Illegal jump into block
103	Label already used for FORMAT
104	Label already defined
105	Jump to FORMAT label
106	DO statement forbidden in this context
107	DO label must follow a DO statement
108	ENDIF forbidden in this context
109	No matching IF for this ENDIF
110	Improperly nested DO block in IF block
111	ELSEIF forbidden in this context
112	No matching IF for ELSEIF
113	Improperly nested DO or ELSE block
114	'(' expected
115	')' expected
116	THEN expected
117	Logical expression expected
118	ELSE statement forbidden in this context
119	No matching IF for ELSE
120	Unconditional GOTO forbidden in this context

121	Assigned GOTO forbidden in this context
122	Block IF statement forbidden in this context
123	Logical IF statement forbidden in this context
124	Arithmetic IF statement forbidden in this context
125	',' expected
126	Expression of wrong type
127	RETURN forbidden in this context
128	STOP forbidden in this context
129	END forbidden in this context
131	Label referenced but not defined
132	DO or IF block not terminated
133	FORMAT statement not permitted in this context
134	FORMAT label already referenced
135	FORMAT must be labeled
136	Identifier expected
137	Integer variable expected
138	'TO' expected
139	Integer expression expected
140	Assigned GOTO but no ASSIGN statements
141	Unrecognizable character constant as option
142	Character constant expected as option
143	Integer expression expected for unit designation
144	STATUS option expected after ',' in CLOSE statement
145	Character expression as file name in OPEN statement
146	FILE= option must be present in OPEN statement
147	RECL= option specified twice in OPEN statement
148	Integer expression expected for RECL= option in OPEN statement
149	Unrecognizable option in OPEN statement
150	Direct-access files must specify RECL= in OPEN statement
151	Adjustable arrays not allowed as input/output list elements
152	End of statement encountered in implied DO, expressions beginning with '(' not allowed as input/output list elements
153	Variable required as control for implied DO
154	Expressions not allowed as reading input/output list elements
155	REC= option appears twice in statement
156	REC= option expects integer expression
157	END= option only allowed in READ statement
158	END= option appears twice in statement
159	Unrecognizable input/output unit

160	Unrecognizable format in input/output statement
161	Options expected after ',' in input/output statement
162	Unrecognizable input/output list element
163	Label used as format but not defined in FORMAT statement
164	Integer variable used as assigned format but no ASSIGN statement
165	Label of an executable statement used as format
166	Integer variable expected for assigned format
167	Label defined more than once as format
169	Function references require '()'
170	Integer expression expected for array dimension bound
171	Lower-dimension bound must be less than or equal to upper-dimension bound
172	DATA statement cannot initialize arrays of unknown size
200	Variable name of named COMMON block expected
201	This variable already saved or declared as STATIC
202	Cannot SAVE dummy arguments
203	COMMON variables may not be saved or declared as STATIC
204	INTEGER and LOGICAL *1, *2, or *4 only
205	No *n allowed for DOUBLE PRECISION
206	Only REAL*4 or REAL*8 allowed
207	No *n allowed for DOUBLE COMPLEX
208	Size expression only allowed for CHARACTER
209	INTEGER constant expression expected
210	INTEGER constant or INTEGER constant expression expected
211	CHARACTER substring expression out of range
212	CHARACTER substring expression must be of type INTEGER
213	Error in CHARACTER substring expression
214	CHARACTER expression expected
215	LOGICAL expression expected
216	CHARACTER*(*) only allowed for dummy arguments
217	Undeclared PARAMETER constant
218	Constant expression not allowed
219	Arithmetic operators only apply to numeric values
220	Malformed COMPLEX constant
221	Maximum of seven levels of implied-DO allowed
222	Error in DATA statement variable list
223	Error in implied-DO list in DATA statement

224 Variables in named COMMON can only appear in a DATA statement that is in a block data subprogram
225 Integer subscript expected
226 Subscript error
227 This identifier is already in use as an implied-DO control variable
228 Integer constant expression or implied-DO control variable expected
229 Integer expression required
230 Division by zero

231 Error in COMPLEX primary
232 Numeric expression or CHARACTER expression expected
233 COMPLEX can only compare for equality
234 COMPLEX is not compatible with DOUBLE PRECISION
235 Constant expression expected
236 ENTRY statements must appear in subroutine or function subprograms
237 ENTRY statements cannot be within a block IF or a DO statement range
238 Concatenation only applies to CHARACTER values
239 ':' expected
240 Substring operations only apply to CHARACTER variables or CHARACTER array elements

241 Error in implied DO expression in a DATA statement
242 Implied DO iteration count is zero in a DATA statement
243 Error in formal argument list
244 Alternate return is not allowed in a function subprogram
245 Substring error in EQUIVALENCE statement
246 EQUIVALENCE statement must not require *2, *4, or *8 variables to be allocated on odd-byte addresses
247 EQUIVALENCE statement must not require a COMMON block to be allocated on odd-byte addresses
248 CHARACTER arguments cannot contain concatenation of values that are of size *(*)
249 Numeric expression expected
250 Subroutine or function name has already been used as a COMMON name

251 Recursive calls are not allowed
252 Statement functions require variable or value arguments
253 Alternate ENTRY in character function must be of type CHARACTER and must be the same size as the function

254	This intrinsic function cannot be passed as an argument
255	Executable statements cannot appear in block data subprograms
256	An argument to an ENTRY statement has already appeared as a local variable
270	Assigned GOTO variable must be INTEGER or INTEGER*4
271	INTEGER, REAL, or DOUBLE PRECISION variable expected
272	INTEGER, REAL, or DOUBLE PRECISION expression expected
273	Unrecognizable element in option list
274	Option appears more than once in an option list
275	Incorrect type for variable
276	Variable must be *4 in size
277	CHARACTER variable or CHARACTER array element required
278	CHARACTER expression expected
279	Cannot have FILE and UNIT specifier in same INQUIRE statement
280	Must have a FILE or UNIT specifier in INQUIRE statement
281	Must have UNIT specifier
282	PRINT statement requires no option list — use WRITE
283	WRITE statement must have an option list
284	READ statement must not have both REC= and END= options
285	Must not specify REC= option with * format specifier
286	Cannot do internal input/output with * format specifier
287	Cannot use REC= specifier with internal input/output
288	Malformed implied DO loop
289	Implied DO loop must have simple variable for loop control
290	Wrong number of arguments to intrinsic function
291	Unit set more than once in input/output statement
292	No unit specified in input/output statement
293	Error in FORMAT statement
294	Hexadecimal constant expected
295	Too many characters in statement
296	Cannot find INCLUDE file
299	Improper use of Hollerith constant
300	Non-ANSI standard feature used
400	Code file write error
403	Procedure too large (code buffer too small)
405	Blank lines are not allowed with free-form input
406	A comment line cannot follow a continuation line in free-form input

407	A label can have only 1 to 5 decimal digits
420	Reserved words must be in lowercase
500	''' (Single quote) expected
501	Binary constant expected
502	Octal constant expected
503	Declared size too small for binary constant in DATA statement — All digits besides 16 rightmost truncated
504	Declared size too small for binary constant in DATA statement — All digits besides 8 rightmost truncated
505	DO WHILE statement forbidden in this context
506	END DO expected
507	Invalid binary constant digit(s)
508	Invalid octal constant digit(s)
509	Declared size too small for octal constant — All digits besides 6 rightmost truncated
510	Declared size too small for octal constant — All digits besides 3 rightmost truncated
511	Invalid hex constant digit(s)
512	END DO forbidden in this constant
513	DO, IF, or DO WHILE block not terminated
514	Declared size too small for hexadecimal constant — All digits besides 4 rightmost truncated
515	Declared size too small for hexadecimal constant — All digits besides 2 rightmost truncated
516	Hex constant expected
517	Declared size too small for hexadecimal constant — All digits besides 8 rightmost truncated
518	Declared size too small for octal constant — All digits besides 11 rightmost truncated
519	Declared size too small for binary constant in DATA statement — All digits besides 32 rightmost truncated
520	A namelist group name must be declared only once
521	Namelist group name has been declared as variable previously
522	Dummy arguments may not appear in a NAMELIST statement
523	Namelist READ or WRITE should not specify iolist
524	Only COMPLEX*8 or COMPLEX*16 allowed
525	This VS or VAX feature is not supported
526	This identifier name has been declared as namelist name previously

532	Declared size too small for hexadecimal constant — All digits besides 16 rightmost truncated
552	Cannot declare function as AUTOMATIC or STATIC
553	Same identifier declared as both AUTOMATIC and STATIC (or saved)
554	Cannot declare dummy arguments as AUTOMATIC or STATIC
555	Variables declared as AUTOMATIC may not appear in COMMON statement
556	Variable declared as AUTOMATIC may not be equivalenced with static variable
557	Variables declared as AUTOMATIC may not appear in DATA statement
559	A variable has been declared as AUTOMATIC more than once
560	Single subscript reference for multi-dimensional array element in EQUIVALENCE statement
562	In IMPLICIT statement, the dollar sign ($) follows the letter Z
581	Not a VS FORTRAN feature or syntax
582	Not an RT PC FORTRAN 77 Version 1.1 feature or syntax
583	Not a VAX FORTRAN feature or syntax
584	Dynamic COMMON is not allowed to initialize data at compile time
585	The following qualifiers of OPTIONS statement have no effect in AIX System (/G_FLOATING, /CHECK)
595	The data type of a dummy argument of a function is undefined
596	The data type of a dummy argument of a statement function is undefined
597	The data type of a function is undefined
598	The data type of a statement function is undefined
599	Variable type undefined due to 'u-' option, or IMPLICIT UNDEFINED statement or IMPLICIT NONE statement specified
600	Non-blank characters truncated in string constant
610	Dummy arguments cannot appear in type initialization statements
620	Variables in blank COMMON cannot appear in type initialization statements
622	Names of subroutines, functions, intrinsic functions, statement functions, and namelists cannot appear in type initialization statements

900	OPTIMIZER ERROR PHASE 0 — degrade optimization level
901	OPTIMIZER ERROR PHASE 1 — degrade optimization level
902	OPTIMIZER ERROR PHASE 2 — degrade optimization level
903	OPTIMIZER ERROR PHASE 3 — degrade optimization level
904	OPTIMIZER ERROR PHASE 4 — degrade optimization level
905	OPTIMIZER ERROR PHASE 5 — degrade optimization level
906	OPTIMIZER ERROR PHASE 6 — degrade optimization level
907	OPTIMIZER ERROR PHASE 7 — degrade optimization level
908	OPTIMIZER ERROR PHASE 8 — degrade optimization level
909	OPTIMIZER ERROR PHASE 9 — degrade optimization level

1000	Could not do block write on outfile
1001	Could not do block read on outfile
1002	Could not do block read on infile
1003	Could not seek to block requested in infile
1004	No more memory
1005	Code not implemented yet
1006	FATAL CODE GENERATION ERROR
1007	Unable to open input file *.i
1008	Unable to open output file *.obj
1009	Input file is not a .i file
1010	Input file is not correct version

Run-Time Messages

RT PC VS FORTRAN contains a file of run-time error messages named "vsfrtmsg.inc". The compiler generates error numbers and messages if this file is present in the default directory and errors are encountered.

600	FORMAT statement missing final ')'
601	Sign not expected in input
602	Sign not followed by digit in input
603	Digit expected in input
604	Missing N or Z after B in format
605	Unexpected character in format
606	Zero repetition factor in format not allowed
607	Integer expected for w field in format
608	Positive integer required for w field in format

609	'.' expected in format
610	Integer expected for d field in format
611	Integer expected for e field in format
612	Positive integer required for e field in format
613	Positive integer required for w field in A format
614	Hollerith field in format must not appear for reading
615	Hollerith field in format requires repetition factor
616	X field in format requires repetition factor
617	P field in format requires repetition factor
618	Integer appears before '+' or '-' in format
619	Integer expected after '+' or '-' in format
620	P format expected after signed repetition factor in format
621	Maximum nesting level (10 levels) for formats exceeded
622	')' has repetition factor in format
623	Integer followed by ',' invalid in format
624	'.' is invalid format-control character
625	Character constant must not appear in format for reading
626	Character constant in format must not be repeated
627	'/' in format must not be repeated
628	'\', '$', ':', 'S', 'SP', and 'SS' in format must not be repeated
629	BN or BZ format control must not be repeated
630	Attempt to perform input/output on unknown unit number
631	Formatted or list-directed input/output attempted on file opened as unformatted
632	Format fails to begin with '('
633	I format expected for integer read
634	F, D, G, or E format expected for real read
635	Two '.' characters in formatted real read
636	Digit expected in formatted real read
637	L format expected for logical read
639	T or F expected in logical read
640	A format expected for character read
641	I format expected for integer write
642	w field in F format not greater than d field + 1
643	Scale factor out of range of d field in E format
644	E, D, G, or F format expected for real write
645	L format expected for logical write
646	A format expected for character write

647	Attempt to do unformatted input/output to a file opened as formatted
648	Unable to write blocked output — possibly no room on output device
649	Unable to read blocked input
650	Error in formatted text file — no carriage return in last 512 bytes
651	Integer overflow on input
652	T, TL, or TR in format must not be repeated
653	Positive integer expected for c field in T, TL, or TR format
654	Attempt to open direct-access unit on unblocked device
655	Attempt to do external input/output on a unit beyond end-of-file record
656	Attempt to position a unit for direct access on a non-positive record number
657	Attempt to do direct access on a unit opened as sequential
658	Attempt to position direct-access unit on an unblocked device
659	Attempt to position direct-access unit beyond end-of-file for reading
660	Attempt to backspace unit connected to unblock device or unformatted file
661	Attempt to backspace sequential unformatted unit
662	Argument to ASIN or ACOS out of bounds — ABS(x) > 1.0
663	Argument to SIN or COS too large
664	Attempt to do unformatted input/output to internal unit
665	Attempt to put more than one record into an internal unit
666	Attempt to write more characters to an internal unit than its length
667	EOF called on unknown unit
668	Direct-access formatted input files must not use DLE
669	Error in opening file
670	Error in closing file
671	Cannot specify KEEP in CLOSE if file opened SCRATCH
672	Unrecognizable option specified as character value in input/output statement
673	File name required unless status is SCRATCH
674	Must not name file if status is SCRATCH
675	Record length not allowed for sequential files
676	Record length must be positive
677	Record length must be specified for direct-access files
678	BLANK option only for formatted files
679	Rewind only allowed on sequential files
680	Endfile only allowed on sequential files

681	Backspace only allowed on sequential files
682	Formatted records must be less than or equal to 512 characters
683	More characters written to internal file record than record length
684	Incorrect number of characters read in formatted record of direct-access file
685	Attempt to write too many characters into formatted record of direct-access file
686	No repeatable edit-descriptor found and format exhausted
687	Digit expected in input field exponent
688	Too many digits in input real number
689	Numeric field expected in input
690	Unexpected character encountered in list-directed or namelist-directed input
691	Repeat factor in list-directed input must be positive
692	',' between reals for complex expected in list-directed input
693	')' expected to terminate complex in list-directed input
694	Attempt to do list-directed or namelist-directed input/output to direct-access file
697	Integer variable not currently assigned a FORMAT label
698	End-of-file encountered on a read with no END= option
699	Integer variable not assigned a label used in assigned GOTO statement
701	Integer input item expected for list-directed input
702	Numeric input item expected for list-directed input
703	Logical input item expected for list-directed input
704	Complex input item expected for list-directed input
705	Character input item expected for list-directed input
706	Incorrect number of bytes read or written to direct-access unformatted file
707	Substring index range error
708	Invalid character in hex read
709	Invalid character in octal read
720	Read or write beyond the end of internal file
751	Subscript error in namelist input record
752	Item name in namelist input record is not defined in namelist item list
753	No input data for specified namelist group name
754	No item name precedes '=' or '(' in namelist input record
755	'=' expected after item name in namelist input record

756	List-directed input/output to internal file is valid only under IBM mode
760	Positive infinity floating-point exception — Maximum positive number substituted
761	Negative infinity floating-point exception — Maximum negative number substituted
762	NaN floating-point exception — Maximum positive (or negative) number substituted
763	Q format code is valid as character count edit-descriptor only under VX mode

Appendix B. ASCII Character Set

This appendix lists the standard ASCII characters in numerical order with the corresponding decimal, octal, and hexadecimal values. The control characters are indicated by a "Ctrl-" notation. For example, the horizontal tab (HT) is indicated by "Ctrl-I", which is keyed by simultaneously pressing the Ctrl key and I key.

Note that this character set was originally developed for teletype communications. Consequently, most of the original control characters (decimal 0 through 31) are undefined in other types of communication. However, two important control characters have retained their original function: LF (decimal 10), which generates a line feed (causing subsequent output on a display or printer to appear on the next line), and CR (decimal 13), which generates a carriage return.

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
0	000	00	Ctrl-@	NUL	null
1	001	01	Ctrl-A	SOH	start of heading
2	002	02	Ctrl-B	STX	start of text
3	003	03	Ctrl-C	ETX	end of text
4	004	04	Ctrl-D	EOT	end of transmission
5	005	05	Ctrl-E	ENQ	inquiry
6	006	06	Ctrl-F	ACK	acknowledge
7	007	07	Ctrl-G	BEL	bell
8	010	08	Ctrl-H	BS	backspace

Figure B-1 (Part 1 of 6). ASCII Character Set

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
9	011	09	Ctrl-I	HT	horizontal tab
10	012	0A	Ctrl-J	LF	line feed
11	013	0B	Ctrl-K	VT	vertical tab
12	014	0C	Ctrl-L	FF	form feed
13	015	0D	Ctrl-M	CR	carriage return
14	016	0E	Ctrl-N	S0	shift out
15	017	0F	Ctrl-O	SI	shift in
16	020	10	Ctrl-P	DLE	data link escape
17	021	11	Ctrl-Q	DC1	device control 1
18	022	12	Ctrl-R	DC2	device control 2
19	023	13	Ctrl-S	DC3	device control 3
20	024	14	Ctrl-T	DC4	device control 4
21	025	15	Ctrl-U	NAK	negative acknowledge
22	026	16	Ctrl-V	SYN	synchronous idle
23	027	17	Ctrl-W	ETB	end of transmission block
24	030	18	Ctrl-X	CAN	cancel
25	031	19	Ctrl-Y	EM	end of medium
26	032	1A	Ctrl-Z	SUB	substitute
27	033	1B	Ctrl-[	ESC	escape
28	034	1C	Ctrl-\	FS	file separator
29	035	1D	Ctrl-]	GS	group separator
30	036	1E	Ctrl-^	RS	record separator
31	037	1F	Ctrl-_	US	unit separator
32	040	20		SP	space
33	041	21		!	
34	042	22		"	
35	043	23		#	
36	044	24		$	
37	045	25		%	

Figure B-1 (Part 2 of 6). ASCII Character Set

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
38	046	26		&	
39	047	27		'	apostrophe
40	050	28		(	
41	051	29		)	
42	052	2A		*	
43	053	2B		+	
44	054	2C		,	comma
45	055	2D		-	minus
46	056	2E		.	period
47	057	2F		/	
48	060	30		0	
49	061	31		1	
50	062	32		2	
51	063	33		3	
52	064	34		4	
53	065	35		5	
54	066	36		6	
55	067	37		7	
56	070	38		8	
57	071	39		9	
58	072	3A		:	
59	073	3B		;	
60	074	3C		<	
61	075	3D		=	
62	076	3E		>	
63	077	3F		?	
64	100	40		@	
65	101	41		A	
66	102	42		B	

Figure B-1 (Part 3 of 6). ASCII Character Set

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
67	103	43		C	
68	104	44		D	
69	105	45		E	
70	106	46		F	
71	107	47		G	
72	110	48		H	
73	111	49		I	
74	112	4A		J	
75	113	4B		K	
76	114	4C		L	
77	115	4D		M	
78	116	4E		N	
79	117	4F		O	
80	120	50		P	
81	121	51		Q	
82	122	52		R	
83	123	53		S	
84	124	54		T	
85	125	55		U	
86	126	56		V	
87	127	57		W	
88	130	58		X	
89	131	59		Y	
90	132	5A		Z	
91	133	5B		[	
92	134	5C		\	
93	135	5D		]	
94	136	5E		^	
95	137	5F		_	underscore

Figure B-1 (Part 4 of 6). ASCII Character Set

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
96	140	60		`	grave
97	141	61		a	
98	142	62		b	
99	143	63		c	
100	144	64		d	
101	145	65		e	
102	146	66		f	
103	147	67		g	
104	150	68		h	
105	151	69		i	
106	152	6A		j	
107	153	6B		k	
108	154	6C		l	
109	155	6D		m	
110	156	6E		n	
111	157	6F		o	
112	160	70		p	
113	161	71		q	
114	162	72		r	
115	163	73		s	
116	164	74		t	
117	165	75		u	
118	166	76		v	
119	167	77		w	
120	170	78		x	
121	171	79		y	
122	172	7A		z	
123	173	7B		{	
124	174	7C		|	

Figure B-1 (Part 5 of 6). ASCII Character Set

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Meaning
125	175	7D		}	
126	176	7E		~	
127	177	7F		DEL	delete

Figure B-1 (Part 6 of 6). ASCII Character Set

Appendix C. Migrating Programs

IBM RT PC VS FORTRAN is source-language compatible with IBM VS FORTRAN Version 2, IBM RT PC FORTRAN 77 Version 1.1, and VAX FORTRAN Version 3, except for the minor limitations described in this appendix. Unless noted here, the statements, data types, and compiler directives in each of these variations of FORTRAN are supported.

Most VS FORTRAN Version 2, RT PC FORTRAN 77 Version 1.1, and VAX FORTRAN Version 3 programs may be compiled on the RT PC and executed without modification, although some compiler directives are accepted syntactically but not functionally. Also, because of differences in hardware architectures, operating systems, and compiler implementations, some of these programs may produce unintended results.

The purpose of this appendix is to describe the areas of the compiler that are known to cause problems so that you can determine the extent to which your programs might be affected, and can implement the necessary changes to achieve the intended results.

From VS FORTRAN Version 2

Limitations

The following VS FORTRAN Version 2 features are not supported in RT PC VS FORTRAN:

- quadruple precision
- FORTRAN 66 language mode
- asynchronous input/output
- indexed file support

- floating-point exceptions.

Programs that use these features must be recoded so that these features are not used.

Uncertainties

Differences between VS FORTRAN Version 2 and RT PC VS FORTRAN may cause unintended results in the following areas.

Floating-Point Representation

Precision of Results: The VS FORTRAN Version 2 and RT PC VS FORTRAN floating-point representations differ in the number of bits used to represent the mantissa and exponent of a number, and therefore in the precision of the number. Additionally, there may be differences in the algorithms used to compute mathematical functions, which could lead to different results near the limits of precision.

Exception Handling: RT PC VS FORTRAN calls for floating-point exceptions (overflow, underflow, undefined) to be reported by returning a particular bit pattern (+infinity, -infinity, Not-a-Number) rather than by raising an actual exception condition.

Representation Dependence: Unintended results may be produced by programs that map floating-point variables onto other data types and depend on the bitwise floating-point representation.

Output Format: Since ANSI Standard FORTRAN 77 does not precisely specify the output format for floating-point numbers, the output format may differ in some instances.

Character Representation

Environments: VS FORTRAN Version 2 operates in EBCDIC environments, while RT PC VS FORTRAN operates in ASCII.

Character Data Values: Unintended results may be produced by programs that depend on a particular data value for a character or a particular relationship among character data values.

Collating Sequence: Unintended results may be produced by programs that depend on the order of character values to sort or otherwise work with character data.

Binary Files with Embedded Character Data: Characters remain in EBCDIC when you port a FORTRAN-created binary data file in which character and numeric data is mixed from an IBM System 370 to an RT PC.

Run-Time Errors

Error Numbers, Contexts, Message Texts: Error numbers, contexts, and message texts are different.

IOSTAT Tests: Unintended results may be produced by programs that test IOSTAT for particular values to indicate run-time error conditions.

Data Storage

Uninitialized Data: Unintended results may be produced by programs that depend on the value of uninitialized storage or on the value of previously used storage uninitialized in a particular subroutine.

Static Treatment of Local Variables: Unintended results may be produced by programs that depend on local variables having the SAVE attribute even when SAVE is not specified.

Logical Representation: Unintended results may be produced by programs that depend on the internal representation of LOGICAL data values.

Storage Mapping: Unintended results may be produced by programs that index out of one array and into another. In general, unintended results may be produced by programs that depend on the storage layout or alignment of data.

Files

Binary Files Not Pure: If a FORTRAN-created binary data file is ported from an IBM System 370 to an RT PC, the internal format of the data file may be different.

Character and Floating-Point Files: Data files containing characters or floating-point numbers must be mapped by a translate utility if they are to be ported.

File Names: Case is significant in the RT PC AIX environment, but not significant in the IBM System 370 environments.

Function Calls

Parameter Persistence: Programs can call a FORTRAN subroutine with a parameter list and subsequently enter the same subroutine through an ENTRY statement with a shorter parameter list. However, the program should not depend on the parameter values of the extra parameters persisting from the first call to the second call. Such programs may produce unintended results.

Function Results When No Assignment is Made: Unintended results may be produced by programs that depend on a particular function result (such as 0) when no assignment to the function has been made.

Order of Evaluation of Parameter Expressions: Unintended results (because of side effects) may be produced by programs that depend on parameter expressions being evaluated in a particular order.

Mismatched Parameter Types: Unintended results may be produced by programs that intentionally pass character parameters to subroutines with non-character dummy parameters which then pass them on to other subroutines.

Compiler Behavior

Using Debugger as Part of FORTRAN Language: In VS FORTRAN Version 2, the debugger may be treated as part of the language. For example, the DISPLAY statement might be used instead of FORTRAN PRINT. This is not supported in RT PC VS FORTRAN.

From RT PC FORTRAN 77 Version 1.1

Uncertainties

Differences between RT PC FORTRAN 77 Version 1.1 and RT PC VS FORTRAN may cause unintended results in the following areas.

Run-Time Errors

Error Numbers, Contexts, Message Texts: Error numbers, contexts, and message texts are different.

IOSTAT Tests: Unintended results may be produced by programs that test IOSTAT for particular values to indicate run-time error conditions.

Data Storage

Uninitialized Data: Unintended results may be produced by programs that depend on the value of uninitialized storage or on the value of previously used storage uninitialized in a particular subroutine.

Logical Representation: Unintended results may be produced by programs that depend on the internal representation of LOGICAL data values.

Storage Mapping: Unintended results may be produced by programs that index out of one array and into another. In general, unintended results may be produced by programs that depend on the storage layout or alignment of data.

Function Calls

Function Results When No Assignment is Made: Unintended results may be produced by programs that depend on a particular function result (such as 0) when no assignment to the function has been made.

Order of Evaluation of Parameter Expressions: Unintended results (because of side effects) may be produced by programs that depend on parameter expressions being evaluated in a particular order.

Mismatched Parameter Types: Unintended results may be produced by programs that intentionally pass character parameters to subroutines with non-character dummy parameters which then pass them on to other subroutines.

From VAX FORTRAN Version 3

Limitations

The following VAX FORTRAN Version 3 features are not supported in RT PC VS FORTRAN:

- quadruple precision
- text libraries
- indexed file support
- expressions in FORMAT
- run-time range checking
- argument list built-in functions
- %LOC function
- non-ANSI keywords in input/output statements
- ENCODE and DECODE statements
- Alternative PARAMETER syntax
- octal notation for integer constants
- DEFINE FILE statement
- FIND statement
- /NOF77 interpretation of external statement
- RADIX-50 constants and character set
- ERRSNS subroutine.

Programs that use these features must be recoded so that these features are not used.

Uncertainties

Differences between VAX FORTRAN Version 3 and RT PC VS FORTRAN may cause unintended results in the following areas.

Floating-Point Representation

Precision of Results: The VAX FORTRAN Version 3 and RT PC VS FORTRAN floating-point representations differ in the number of bits used to represent the mantissa and exponent of a number, and therefore in the precision of the number. Additionally, there may be differences in the algorithms used to compute mathematical functions, which could lead to different results near the limits of precision.

Exception Handling: RT PC VS FORTRAN calls for floating-point exceptions (overflow, underflow, undefined) to be reported by returning a particular bit pattern as a result (+infinity, -infinity, Not-a-Number) rather than by raising an actual exception condition.

Representation Dependence: Unintended results may be produced by programs that map floating-point variables onto other data types and depend on the bitwise floating-point representation.

Output Format: Since ANSI Standard FORTRAN 77 does not precisely specify the output format for floating-point numbers, the output format may differ in some instances.

Run-Time Errors

Error Numbers, Contexts, Message Texts: Error numbers, contexts, and message texts are different.

IOSTAT Tests: Unintended results may be produced by programs that test IOSTAT for particular values to indicate run-time error conditions.

Data Storage

Uninitialized Data: Unintended results may be produced by programs that depend on the value of uninitialized storage or on the value of previously used storage uninitialized in a particular subroutine.

Integer Representation: Unintended results may be produced by programs that equivalence longer and shorter forms of INTEGER data and that depend on the internal order of significant bytes.

Logical Representation: Unintended results may be produced by programs that depend on the internal representation of LOGICAL data values.

Storage Mapping: Unintended results may be produced by programs that index out of one array and into another. In general, unintended results may be produced by programs that depend on the storage layout or alignment of data.

Files

Binary Files Not Pure: If a FORTRAN-created binary data file is ported from a VAX to an RT PC, the internal format of the data file may be different.

Character and Floating-Point Files: Data files containing characters or floating-point numbers must be mapped by a translate utility if they are to be ported.

File Names: Case is significant in the RT PC AIX environment, but not significant in the VAX environments.

Function Calls

Function Results When No Assignment is Made: Unintended results may be produced by programs that depend on a particular function result (such as 0) when no assignment to the function has been made.

Order of Evaluation of Parameter Expressions: Unintended results (because of side effects) may be produced by programs that depend on parameter expressions being evaluated in a particular order.

Mismatched Parameter Types: Unintended results may be produced by programs that intentionally pass character parameters to subroutines with non-character dummy parameters which then pass them on to other subroutines.

Index

IBM RT PC

Reader's Comment Form

IBM RT PC VS FORTRAN
User's Guide

SH23-0129-0

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the authorized IBM RT PC dealer in your area.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 79L, Building 4
Commerce Park & Eagle Road
Danbury, Connecticut 06810

Fold and tape Fold and tape

Cut or Fold Along Line

Tape Please Do Not Staple Tape

International Business Machines Corporation
Department 79L, Building 4
Commerce Park and Eagle Road
Danbury, CT 06810

Printed in the
United States of America

SH23-0129

SH23-0129-00