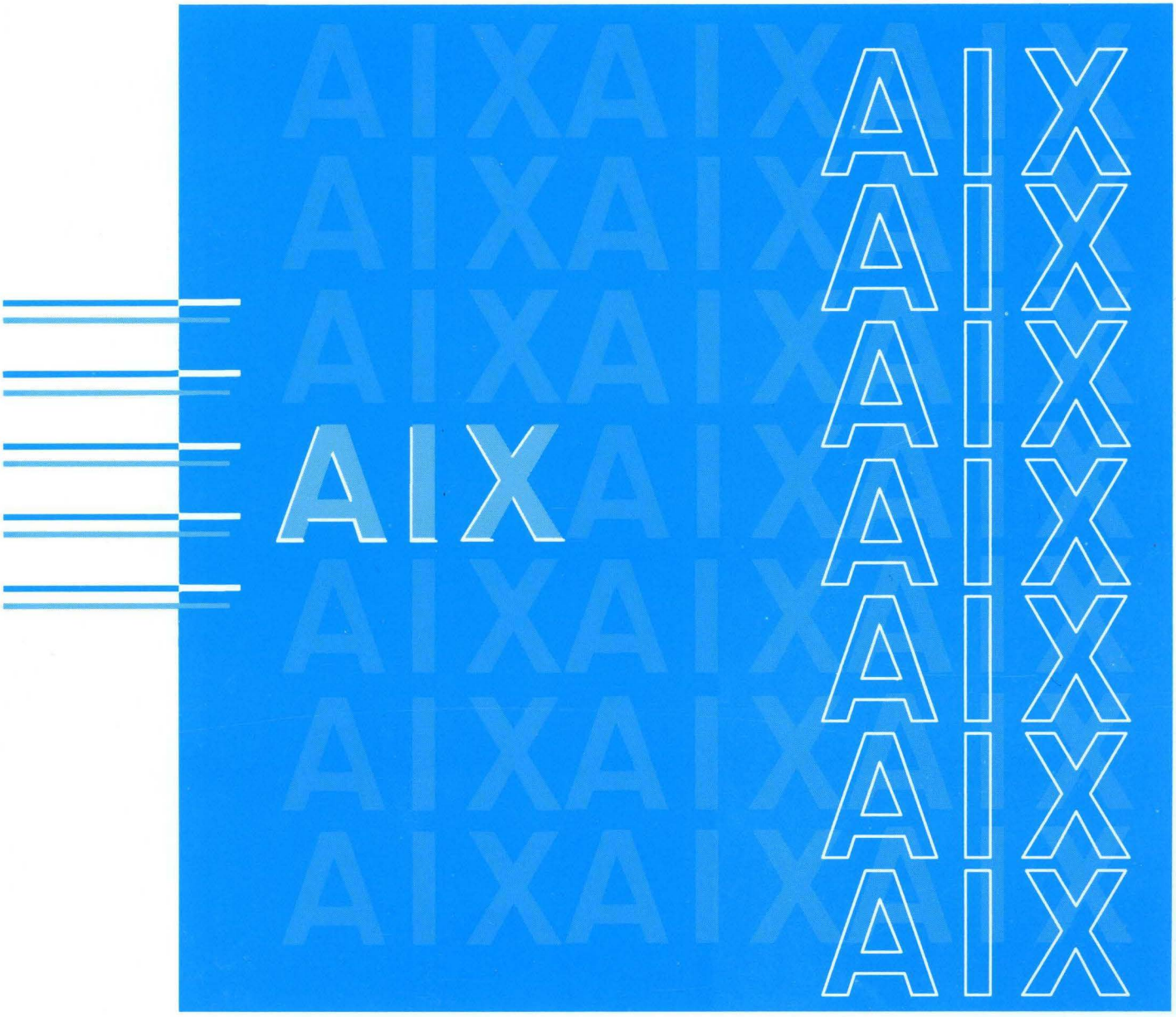
AIX
Operating
System

C Language User's Guide



IBM

AIX Operating System

C Language User's Guide

SC23-2057-01

Second Edition (December 1989)

This edition applies to Version 1.2 of the IBM Advanced Interactive Executive for the System/370 (AIX/370), Program Number 5713-AFL, to Version 2.2.1 of the IBM Advanced Interactive Executive for RT (AIX RT), Program Number 5601-061, and for Version 1.2 of the IBM Advanced Interactive Executive for the Personal System/2, Program Number 5713-AEQ, and to all subsequent releases until otherwise indicated in new editions or technical newsletters. Changes are made periodically to the information herein; these changes will be reported in technical newsletters or in new editions of this publication.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

International Business Machines Corporation provides this manual "as is," without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

Publications are not stocked at the address given below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM dealer or your IBM marketing representative.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to IBM Corporation, Department 35R, 36 Apple Ridge Road, Danbury, CT 06810. IBM may use or distribute, in any way it believes appropriate and without incurring any obligation to the sender, whatever information it receives in this manner.

Portions of the code and documentation were developed at the Electrical Engineering and Computer Sciences Department at the Berkeley Campus of the University of California under the auspices of the Regents of the University of California.

About This Book

This book shows you how to develop, link, and execute programs in C language, using the Advanced Interactive Executive (AIX) Operating System. It describes the operating system dependencies of the language as well as the use of C language, related software utilities, and other program development tools.

Who Should Read This Book

This book is written for programmers who want to write application programs in C language that run on the AIX Operating System.

What You Should Know

You should have an intermediate to advanced understanding of the C programming language. You should also have a general understanding of programming concepts and terminology and some experience in writing programs. To get the most out of this book, you should know how to operate the Personal System/2, the RT PC, or the System/370.

How to Use This Book

This book is intended as a companion reference to the *AIX C Language Reference*. It is organized according to the general classes of elements that are used to construct programs in the C language. To locate specific topics, use the table of contents or the index.

Highlighting

This book uses different type styles to distinguish among certain kinds of information. General information is printed in the standard type style (for example, this sentence).

The following type styles indicate other types of information:

- Commands and keywords appear in **bold** type.
- Examples, words, and characters that must be entered literally appear in `monospace` type.
- Variables appear in *italics*.
- New terms appear in ***bold italic*** type.

Syntax Diagrams

The following typographic conventions are used in the syntax diagrams: If you need information on how to read the syntax diagrams, refer to the *AIX Commands Reference*.

- Syntactic categories appear between angle brackets (< >).
- Alternative syntactic categories appear on separate lines.

AIX is a trademark of International Business Machines Corporation.

PS/2 is a registered trademark of International Business Machines Corporation.

RT PC is a registered trademark of International Business Machines Corporation.

System/370 is a trademark of International Business Machines Corporation.

- Ellipses indicate that a preceding parameter can be repeated, for example:

```
<object>...
```

- Variables that should be replaced by data objects in actual program statements appear in *italics*.
- An optional terminal symbol or non-terminal symbol is indicated by the notation:

```
<object>...
          opt
```

- A syntactic definition is indicated by the name of the object being defined, followed by a colon, followed by the symbols that make up the object. Here is an example of the syntactic definition for a compound-statement:

```
<compound-statement>:
  { <declaration>...    <statement>...   }
                  opt              opt
```

  This specification states that a compound-statement is made up of a left brace, followed by one or more optional declarations and one or more optional statements followed by a right brace. Note that this definition provides for an empty compound statement.

- Brackets [ ] indicate optional items and subscripts of an array.
- Braces { } enclose optional elements that can be repeated more than once.

Related Publications

For additional information, you may want to refer to the following publications:

- *AIX C Language Reference*, SC23-2058, describes the C programming language and contains reference information for writing programs in C language that run on the AIX Operating System.
- *AIX Operating System Commands Reference*, SC23-2292 (Vol. 1) and SC23-2184 (Vol. 2), lists and describes the AIX/370 and AIX PS/2 Operating System commands.
- *AIX Programming Tools and Interfaces*, SC23-2304, describes the programming environment of the AIX Operating System and includes information about operating system tools that are used to develop, compile, and debug programs.
- *SAA Common Programming Interface C Reference*, SC26-4353, describes each component of the common programming interface.
- *AIX Operating System Technical Reference*, SC23-2300 (Vol. 1) and SC23-2301 (Vol. 2), describes the system calls and subroutines a programmer uses 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.
- *Using the AIX Operating System*, SC23-2291, shows the beginning user how to use AIX Operating System commands to do such basic tasks as log in and out of the system, display and print files, and set and change passwords. It includes information for intermediate to advanced users about how to use communication and networking facilities and write shell procedures.

Contents

Chapter 1. Introduction

CONTENTS

About This Chapter

This chapter includes the main features of the AIX C language and a summary of terms and concepts in this manual.

Overview

IBM AIX C Language provides a high-performance optimizing compiler that produces object code for execution under the AIX Operating System.

IBM AIX C offers these functions:

- Automated installation
- Optimized executable code
- Excellent compile-time performance
- Separate module compilation
- Access to command-line options
- Easy inter-language linkages among VS Pascal, VS FORTRAN, and C.

Throughout this manual, whenever possible, common elements between AIX PS/2, RT PC AIX, and AIX/370 have been grouped together.

The main topics covered in this manual are:

- ***The Compiler***

  C-Language source code is compiled on AIX by executing the **cc** command. It produces executable binary code from the C source code.

  C-language source code can also be compiled on the PS/2 by executing the **vs** command. The **vs** command is a script that is user modifiable.

- ***Data Representation***

  This manual describes characteristics of the C data types as defined by the C Compiler in AIX. It also describes how the C Compiler represents data in storage.

- ***Mixing Languages and Linkage Conventions***

  The PS/2 and RT PC language systems permit the mixing of elements from different languages in a single program. The IBM AIX/370 language system permits mixing elements of the C and Assembler languages in a single program. You should be familiar with the languages you want to mix; the elements of the languages are not described here in detail.

  This manual also describes how to organize System/370 specific run-time functions and how the compiler meets register usage conventions.

  In this manual, the FORTRAN language described is IBM VS FORTRAN; the Pascal language is IBM VS Pascal; and the C language is IBM C.

Chapter 2. The Compiler

CONTENTS

About This Chapter

This chapter contains instructions for compiling C programs and describes each of the command-line options. Optimization capabilities are also described in detail.

The Compiler

C-Language source code is compiled on AIX by executing the **cc** command. See the *AIX Operating System Commands Reference*. It produces executable binary code from the C source code.

Note: C-language source code can also be compiled on the PS/2 by executing the **vs** command. The **vs** command is a script that is user modifiable. See the *AIX Operating System Commands Reference*.

Invoking the Compiler

To run the compiler from the command line, enter the following:

```
cc [file| [ option ] ...
```

where *file* is any one of the file types described later under "File Types" and *option* is any one of the options described under "Command-Line Options."

Description

The **cc** command runs the C compiler. It accepts files containing C source code, assembler source code, or object code and changes them into a form that the compute system can run. The **cc** command compiles and assembles source files and then links them with an specified object files, in order listed on the command line. It puts the resulting executable program in a file named **a.out**. Command-line options are used to modify this process.

The Configuration File

The configuration file, **/etc/cc.cfg**, holds definitions for standard path names of compiler components, header files, and libraries; standard preprocessor macro definitions; and default values of other options. These definitions may be overridden by the **-B, -I, -L, -D,** and **-U** flags. An alternative configuration file may be designated by the **-F** flag. See *AIX Operating System Technical Reference* for a discussion of **cc.cfg**.

Standard macro definitions include **_AIX*_family*** plus one of **_AIX370**, **_AIXPS2** or **AIXRT** to identify the AIX platform, plus **_STDC_** if compilation is at the Standard C language level.

File Types

The **cc** command recognizes and accepts the following *file* types.

file.c A C language source file. A **.c** file is preprocessed, compiled, and assembled to produce an object file named **file.o**. When a single file is compiled and linked in one command, the **.o** file is not preserved. The **-c** flag suppresses linking and produces a **.o file**. Compiling multiple files with or without linking always produces **.o** files.

file.i Treated like a **.c** file, except that the preprocessing step is skipped.

file.s An assembly language source file. A **.s** file is treated like a **.c** file, except that preprocessing and compiling steps are skipped.

file.o An object file. A **.o** file is passed to the linker.

file.a A library (or "archive") file. A **.a** file is passed to the linker.

-l*key* An abbreviation for the library file name **lib***key***.a -l***key* is passed to the linker, which searches for **lib***key***.a** in directories designated by **-L** flags, then in standard directories.

other A file name without a single-letter suffix is passed to the linker.

Command-Line Options

Command-line options may or may not be common across platforms. See the individual options for any notes or exceptions.

The following command-line options are also described with the **cc** command in *AIX Operating System Commands Reference*.

The **cc** command recognizes several flags. In addition, flags intended to modify the action of the linkage editor (**ld**), the assembler (**as**), or the preprocessor (**cpp**) may also appear on the **cc** command line. **cc** sends any flags it does not recognize to these commands for processing. The following list includes the most commonly used **cpp** flags (**-D**, **-I**), and **ld** flags (**-l**, **-L** , **-o**).

-a Reserves a register for extended addressing. You should use this flag if a compiled procedure creates a stack greater than 32,767 bytes. Because this flag causes the compiler to reserve a **register** for use by the assembler, it reduces the number of available registers by one.

Note: This option is not supported on the PS/2 or System/370.

-c Does not send the completed object file to the **ld** command. With this flag, the output of **cc** is a **.o** file for each **.c** or **.s** file.

-D*name*[= *def*] Defines *name* as in a **#define** directive. If no equal sign is specified, the default *def* is 1. If an equal sign but no *def* is specified, the *def* is null. This can be used to remove occurrences of *name* from a file.

-E Runs the named C source file through only the preprocessor and writes the result to standard output.

-f Generates code that uses the Floating-Point Accelerator or Advanced Floating-Point Accelerator. Programs compiled with this flag will run correctly only on 032 microprocessors configured with either of the Floating-Point Accelerators.

Note: This option is not supported on the PS/2 or System/370.

-f2 Generates code that uses the Advanced Floating-Point Accelerator. Programs compiled with this flag will run correctly only on AIX processors configured with the Advanced Floating-Point Accelerator and an Advanced Processor Card.

Note: This option is not supported on the PS/2 or System/370.

-g Produces additional information for use with a symbolic debugger (for example, the **dbx** command).

-G Indicates that global variables are volatile. The optimizer makes fewer transformations when you specify this flag. To make a particular variable volatile, add the **.volatile** specification to its declaration.

Note: This option is not supported on the System/370.

-h Treats files with the suffix **.h** in the same way as files with the suffix **.c**.

Note: This option is not supported on the System/370.

-I*dir* Looks first in the directory specified by *dir*, then looks in the directories on the standard list for **#include** files with names that do not begin with / (slash).

-l*key* Searches the specified library file, where *key* selects the file **lib***key*.**a**. **ld** searches for this file in the directory specified by an **-L** flag, then in **/lib** and **/usr/lib**. The **ld** command searches library files in the order in which you list them on the command line.

Note: If you use the **-l** flag, it must be the last entry on the command line, following any file parameters.

-L*dir* Looks in the directory specified by *dir* for files specified by **-l** keys. If it does not find the file in the directory specified by *dir*, **ld** searches the standard directories.

-N[ndpt]*num* Changes the size of the symbol table (**n**), the dimension table (**d**), the constant pool (**p**), or the space for building the parse tree (**t**). Each table must be changed separately. The default size of the symbol table is 1500; the default size of the dimension table is 2000; the default size for the constant pool is 600; the **default** space for the parse tree is 1000.

Note: This option is not supported on the PS/2 or System/370.

-o*name* Assigns the file name specified by *name* rather than **a.out** to the output file.

Note: On the PS/2, if a debug/disassembler file (**.d**) exists and the output file specified is in a directory other than the current directory, the **.d** files are moved to the output file directory.

-O Sends compiler output to the code optimizers.

-p Prepares the program so that the **prof** command can generate an execution profile. The compiler produces code that counts the number of times each routine is called. If programs are sent to **ld**, the compiler replaces the startup routine with one that calls the **monitor** subroutine at the start (see *AIX Operating System Technical Reference* for a description of this subroutine), and writes a **mon.out** file on normal program termination.

-pg Like **-p**, but invokes a run-time recording mechanism that keeps more extensive statistics and produces a **gmon.out** file at normal termination. You can then use **gprof** to generate an execution profile.

-P Runs the named C source file through only the preprocessor and stores the output in a **.i** file.

Note: The **-P** option is not supported on the System/370. See the **-E** option for equivalent results.

-Q! Turns off inlining. The following may be used:

?	Shows the reason for not inlining in the output file.
-name,name...	Does not inline *name*.
+name,name...	Inlines *name*.
|num	Limits the size increase of the function in which inlining occurs to *num* intermediate operations. The **default** *num* is 100.
#num	Limits the expansion of an individual call to *num* intermediate operators. The **default** *num* is 100.
-@file	Reads a list of forbidden functions from *file*.
+@file	Reads a list of requested functions from *file*.

Requesting a function to be inlined overrides size constraints.

Note: The PS/2 C Compiler only accepts the **-Q!** syntax. In this case, all inlining is disabled.

-R Compiles initialized data into the text segment, making the data read-only.

-S Compiles the specified C programs, storing assembly language output in a **.s** file.

-w Prevents printing of warning messages.

Note: The RT PC C Compiler prevents printing of warning messages about functions that cannot be optimized.

-X Produces an assembler listing. This is stored in a file that has the same name as the assembler source file but with the extension **.lst** instead of **.s**.

Note: On the PS/2, the assembler code listing may not match the executable code generated by the C Compiler. This is because the C Compiler directly generates machine code; there is no separate assembler pass. This is not supported on the System/370.

-U*name* Undefines *name* as in an **#undef** directive.

-y[dmnpz] Specifies the rounding mode for floating-point constant folding. These modes are specified as follows:

d	Disables floating-point constant folding.
m	Rounds toward negative infinity.
n	Rounds to nearest whole number. This is the **default** action and applies to constant folding in all applicable passes of the compiler.
p	Rounds toward positive infinity.
z	Rounds toward 0.

Note: This option is not supported on the PS/2 or the System/370.

-z Uses the **libm.a** version, or a version specified by the user, of the following transcendental functions:

acos	asin	atan	atan2	cos	exp
log	log10	sin	sqrt	tan	

If this flag is not used, the compiler generates inline instructions for the 80387 math co-processor for the PS/2, or the Advanced Floating Point Accelerator for the RT PC. (For more information on **libm.a**, see **math.h** in the *AIX Operating System Technical Reference*.)

Note: This is not supported on the System/370.

Debugging

-v Displays the trace as with **-#** and invokes the programs.

-# Displays a trace of the actions to be taken (for example, invoking the preprocessor), without actually invoking any programs.

Extended Functions

-B*prefix* Constructs path names for substitute preprocessor, compiler or optimizer programs. *prefix* defines part of a path name to the new programs. To form the complete path name for each new program, **cc** adds *prefix* to the standard program names (see *AIX Commands Reference* for a list of the standard program names).

Note: The PS/2 C Compiler does not have a separate optimizer program. It also does not invoke the assembler as an intermediate step to compiling a C program.

For example, if you enter the command:

```
cc testfile.c -B/usr/jim/new
```

cc calls the following compiler programs on the PS/2:

/usr/jim/newcpp
/usr/jim/newvsc
/usr/jim/newvspass2
/usr/jim/newvspass3

cc calls the following compiler programs on the RT PC:

/usr/jim/newcpp
/usr/jim/newccom0
/usr/jim/newcccom1

cc calls the following compiler programs on the System/370:

/usr/jim/newcpp
/usr/jim/newhc1com
/usr/jim/newhc2com
/usr/jim/newas
/usr/jim/newld

Similarly, if you enter the command:

```
cc testfile.c -B/usr/jim/new/
```

cc calls the following compiler programs on the PS/2:

/usr/jim/new/cpp
/usr/jim/new/vsc
/usr/jim/new/vspass2
/usr/jim/new/vspass3

cc calls the following compiler programs on the RT PC:

/usr/jim/new/cpp
/usr/jim/new/ccom
/usr/jim/new/ccom1

cc calls the following compiler programs on the System/370:

/usr/jim/new/cpp
/usr/jim/new/hc1com
/usr/jim/new/hc2com
/usr/jim/new/as
/usr/jim/new/ld

The **default** *prefix* is **/lib/o**.

-t[pcgfal] Applies the **-B** flag instructions for constructing file names to only the designated programs.

The parameters accepted on the PS/2 are:

p *preprocessor*
c *compiler first*
g *compiler second*
f *compiler third*
a *assembler*
l *linkage editor*

The parameters accepted on the RT PC are:

p *preprocessor*
c *compiler first*
q *intermediate code optimizer*
g *compiler second*
o *optimizer*
a *assembler*
l *linkage editor*

The parameters accepted on the System/370 are:

c pass 1 and 2 of the compiler
p preprocessor
l loader
1 pass 1 of the compiler
2 pass 2 of the compiler
a assembler

The **-t** flag with no additional parameters designates by **default** the preprocessor and compiler programs.

If you do not specify the **-B** flag when you specify the **-t** flag, the **default** file name *prefix* is **/lib/n**.

Notes:

1. You can specify this *prefix* with the **-B** flag. However, depending on what combination of the **-B** and the **-t** flags you specify, *prefix* can have two possible default values. If you specify **-B** but no accompanying *prefix*, the default *prefix* is **/lib/o**. If you specify the **-t** flag without also specifying the **-B** flag, the default *prefix* is **/lib/n**.
2. The System/370 does not supply default values for the **-t** or **-B** flag.

-W*c*,*flag1*[,*flag2*...]

Gives the listed flags to the compiler program specified by *c*; *c* can be any one of the values described with the **-t** flag. For example, since both **ld** and **as** recognize a **-o** flag, use **-W** to specify the program to which the flag is to be sent. For example, `-Wl,-o` sends it to **ld** and `-Wa,-o` sends it to **as**.

PS/2 Specific Options

The AIX PS/2 C Compiler specific options are described below. They indicate which features are enabled or disabled when the compiler is invoked. The options are described in alphabetical order.

b+ *Floating Point Computation*

Instructs the compiler to promote all floating-point values to double precision before all floating-point computations.

d+ *Disassembler Information*

Produces additional information for use with the **dis** command (the disassembler).

Note: With this option, you can also use the **dbx** command (symbolic debugger). (See the **g+** option below.) However, the **.d** file does not contain symbolic information; therefore, you can only do machine level debugging.

e*filename* *Error File*

Instructs the compiler to place its error output in the file specified by *filename*. If the **e***filename* option is not specified, error messages are written to the standard error device.

g+ *Debugger Information*

Produces additional information for use with the **dbx** command (the symbolic debugger).

Notes:

1. With this option, you can also use the **dis** command (disassembler), see the **d+** option above. However, allocation of variables into registers is turned off.
2. 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.
3. The optimization process is disabled whenever the **g+** option is specified on the command line.

l*filename* *Listing File*

Instructs the compiler to place its listing output in the file specified by *filename*. If the **l***filename* option is not specified, a listing file is not generated.

l+ *List to Standard Output Device*

Instructs the compiler to generate a listing to the standard output device.

o1+ *Optimization Level 1*

Instructs the compiler to use optimization level 1 (see "Optimization of Programs" on page 2-16).

o2+ *Optimization Level 2*

Instructs the compiler to use optimization level 2 (see "Optimization of Programs" on page 2-16).

o3+ *Optimization Level 3*

Instructs the compiler to use optimization level 3 (see "Optimization of Programs" on page 2-16).

o4+ *Optimization Level 4*

Instructs the compiler to use optimization level 4 (see "Optimization of Programs" on page 2-16).

v+ *Compiler Progress Information*

Instructs the compiler to generate information on the progress of the compile.

w− *No Warning Messages*

Instructs the compiler not to generate warning messages.

System/370 Specific Options

The AIX/370 compiler options are described below.

-F *file*[*:stanza*] Uses an alternate *file* and/or *stanza* for the **cc** configuration. See *AIX Operating System Technical Reference* for discussions of the configuration file **/etc/cc.cfg**.

-Hanno Used in conjunction with the **-S** option (explained below). Specifies that the generated **-S** file is to be annotated with lines from the source file.

-Hansi Causes the compiler to accept only programs conforming to the ANSI Standard.

-Hasm Directs the compiler to produce a (pseudo-) assembly listing of the generated code on standard output, by initializing the **Asm** toggle to On. The assembly listing is annotated with lines from the main source file, but not with lines from any included files. These lines appear as comments immediately preceding the corresponding assembly instructions. If the **-S** option (described below) is also specified, the generated .s file is annotated with lines from the source file, and no listing is written on standard output; that is, it has the same effect as **-Hanno**.

-Hcpp Specifies that the outboard C macro preprocessor (**/lib/cpp**) is to be used, rather than the inboard preprocessor.

-Hnocpp Specifies the use of the inboard C macro preprocessor.

-Hfsingle Specifies that single-precision arithmetic is to be used in computations involving only **float** expressions. That is, floating-point operations are not to be performed in **double** precision, which is the **default**. Note that non-prototyped functions declared to **return float** may actually **return double**, depending on the setting of the toggle Double_**return**. Some programs run much faster with this option, but beware of loss of significance due to lower-precision intermediate computations.

-Hlines=n Causes a page eject to occur after every n lines written to standard output. The **default** of 60 is appropriate for most 6-lines-per-inch printers, which allow a maximum of 66 lines per page for 11-inch paper. The setting of **-Hlines** is intended to allow some blank space at page boundaries. For 8-lines-per-inch (88 lines per page) printers, **-Hlines** should be set to 80 or 82. This option is used in conjunction with the **-Hlist** and **-Hasm** options. If n is 0, no page ejects are emitted.

-Hlist Causes the compiler to generate a source listing on standard output. It works by initializing the List toggle to On.

-Hxa Generates an XA370 executable.

-M Specifies that the outboard C macro preprocessor is to be invoked and **makefile** dependencies are to be generated. The output is sent to standard output. No compilation occurs.

-Hon=toggle Turns a toggle On.

-Hoff=toggle Turns a toggle Off.

-Hpcc Specifies that the compiler is to run in PCC mode. In this mode, the compiler relaxes enough of the ANSI extensions to more or less emulate the Portable C Compiler. This permits old C programs that would not ordinarily compile to compile with little (if any) modification.

Specifically, **-Hpcc** does the following:

Turns on the PCC toggle

Unreserves the following keywords: **signed**, **volatile**, and **const**.

Turns on the **Long_enums** toggle so that **enum** types are mapped to full words as is the PCC convention.

-H+w Issues all warnings by turning off the **PCC_msgs** toggle.

-M Generates makefile dependencies for the named C files and writes the result to standard output. Does not compile or link.

Compiler Toggles (AIX/370)

The AIX/370 C Compiler provides a set of toggles, which can be initialized on the command line, with **-Hon** and **-Hoff**. See "Invoking the Compiler" on page 2-2.

The names, **default** values, and meanings of the compiler toggles are described below. Many of the defaults can be altered by making appropriate changes in the **/etc/cc.cfg** file. Toggles with alterable defaults are denoted by the presence of *alterable* in the descriptions below.

Align_members — Default: On

When On, members of structures are aligned. When Off, no such alignment takes place. Not all other implementations of AIX C permit member alignment to be suppressed.

Asm — Default: Off

When On, a (pseudo-)assembly listing is generated, annotated with source code as assembly comments. If the **Asm** toggle is to be turned On and Off over sections of a module, On-Off pairs should surround function definitions; if the pragma is used among executable statements, the point at which the pragma takes place may be obscured by the use of optimizations.

Char_default_unsigned — Default: On

When On, type **char** is unsigned by **default**.

The Standard allows the type **char** by itself, that is, without the adjectives unsigned or signed, to be either signed or unsigned. Of course, the types unsigned **char** and signed char can be used to explicitly control signedness.

Char_is_rep — Default: Off

The ANSI Standard provides three character data types: **char**, unsigned **char**, and signed **char**. Even though **char** is signed or unsigned, depending on the implementation, it is never the same type as **signed char** or **unsigned char**. One of the effects is that **char*** is type compatible with neither **unsigned char*** nor **signed char***. This increases program portability.

When On, **Char_is_rep** specifies that char is identically either signed **char** or unsigned **char**, depending on whether **char** is signed or not. Although this may permit a program to compile, the program might not compile under another C implementation where the signedness of **char** is different. Be sure to turn this toggle On before any declarations or preprocessor statements.

Cross_jump—Default: On

When Off suppresses the cross-jumping (tail-merging) optimization that is otherwise performed when **-o** is specified. Suppressing cross-jumping can make code easier to debug at the instruction level.

Double_math_only — Default: On

When On, floating-point operations are performed in double precision.

When two operands of certain arithmetic operations are both of type **float**, the Standard permits an implementation to do one of two things: perform the operation using **float** arithmetic, in which case the result of the operation is of type **float**, or convert both operands to type **double** and use double arithmetic, in which case the result of the operation is of type **double**. When toggle **Double_math_only** is turned Off, the first option is used. When it is turned On, the second is used instead.

Double_return — Default: Off

When On, any non-prototype function returning type **float** returns type **double** instead. This convention conforms to some Portable C Compiler implementations.

Downshift_file_names — Default: Off

When On, the file name specification of any subsequent Include pragma is interpreted as if it were in all lowercase. This toggle is useful when moving source code from an operating system in which file-name casing is not significant to a system in which it is significant.

Int_function_warnings — Default: Off

When Off, suppresses the warning message normally generated when a function returning **int** has no return **exprn** statement within it, or when a function returning **int** contains a return within it.

This is to remove frequent warnings for old C source that did not use the reserved word **void** to indicate a function returning no result, because such functions return **int** by **default**.

List — Default: Off

When On, the compiler produces a listing on standard output. It is typically given when starting the compilation but may appear in the source file to turn the listing On or Off around a particular section of source.

Long_enums — Default: Off

When On, any variable of an **enum** type is mapped to a full word. Otherwise, such a variable is mapped to the smallest of one, two, or four bytes, such that all values are representable.

Make_externs_global — Default: On

When On, any local declaration of an object with storage class **extern** is made global if there is not already a global declaration of the object. Early C compilers promoted an **extern** declaration within a function to the global scope. This toggle supports programs depending upon that feature.

Parm_warnings — Default: On

When On, the compiler produces warnings whenever arguments to a non-prototype (old-style) function F do not match the types of the declared formal parameters of F. Frequently this inconsistency is a source of disastrous or difficult-to-find bugs. *Example:*

```
double square(x) double x; {return x*x;}
...
printf("%d\n",square(3));
```

In this example, square is passed the integer 3, not double-precision 3.0, and the compiler issues a warning. The C language definition prohibits the compiler from casting 3 to a **double** before passing it.

To eliminate the compiler warnings, turn Off the toggle **Parm_warnings**.

We recommend, however, that the program text be repaired to eliminate the offending function calls rather than eliminating the potentially useful feedback from the compiler.

PCC — Default: On

There are many UNIX C programs that compile under the Portable C Compiler that will not compile under an ANSI compiler such as the C compiler. When the PCC toggle is turned On, many of the ANSI restrictions are relaxed so that such programs are more likely to compile (with appropriate warnings).

The following is a list of the ANSI restrictions that this toggle affects:

Structure member selection is permitted on a variable of any type; it need not be a **struct** or **union**. However, if the name of the member being selected is declared as a member in more than one **struct** or **union** definition, each occurrence must be declared with the same type and be mapped at the same offset.

Any pointer type is considered compatible with any other pointer type.

Note: Turning On the toggle **Pointers_compatible** has this effect.

Pointers and any integer type are considered compatible.

Note: Turning On the toggle **Pointers_compatible_with_ints** has this effect.

These PCC language features are not supported:

- (Cast) variable as left side of assignment
- Old-fashioned assignment operators (= +)
- Old-fashioned initialization (no =)
- Unsignedness-preserving semantics.

See also the **-Hpcc** compiler option in the section "Invoking the Compiler" on page 2-2.

PCC_msgs — Default: Off

When On, the diagnostic capabilities of the compiler are reduced to the PCC ("Portable C Compiler") level, in that the following warnings are not emitted:

```
Function called but not defined.
"=" used where "==" may have been intended.
```

In addition, this toggle disables warnings about passing an **int**, for example, to a non-prototyped function expecting a **long int**, because **int** and **long int** are the same size.

When all warnings are enabled in C, code must be "squeaky clean" to get through the compiler without a warning. Some users have code that was designed with a compiler that is not so demanding, and would prefer fewer prods from the compiler. Hence the **PCC_msgs** toggle is supplied.

Note: The **-H+w** command-line option turns this toggle Off.

Pointers_compatible — Default: Off

When On, the compiler permits a pointer value to be assigned to an incompatible pointer variable, with an appropriate warning.

Pointers_compatible_with_ints — Default: Off

When On, allows pointers of any type to be compatible with ints, with an appropriate warning. Although this is in violation of Standard and C compiler specifications, many old C programs improperly assign pointers to **ints** and vice versa. This toggle allows such programs to be compiled without modification.

ANSI and the C compiler disallow this dangerous practice because pointers are not necessarily the same size as **ints** on all machines.

Print_ppo — Default: Off

When On, the output of the preprocessor is written to standard output. With this toggle, it is possible to print what the compiler proper receives over a local area of source code. This toggle can be used to inspect the expansion of a macro, by turning the toggle On prior to the macro invocation and Off after it.

Note: This toggle is ignored unless **-Hnocpp** is specified or is the default.

Print_protos — Default: Off

When On, the compiler writes to standard output a new, prototype-style function header for each function definition. This toggle aids in the conversion of C programs to the ANSI prototype syntax derived from the C language. For example, for the function definition:

```
int f(x,y,z) int *x,z[]; double (*y)(); {...}
```

the compiler produces the following output:

```
int f(int *x, double (*y)(), int [z]);
```

The old function header can then be replaced with the generated one.

Prototype_conversion_warn — Default: On

When On, the compiler generates a warning message when a function's argument is converted due to a prototype declaration.

When using function prototypes, the compiler may automatically convert a function's argument so that the argument's type matches that of the formal parameter. Wherever such a conversion does not match what would happen in the absence of prototypes, such C code would probably not run correctly on older C compilers that lack prototypes.

Prototype_override_warnings — Default: On

When On, the compiler produces a warning whenever a declaration (not definition) for a function using the new prototype syntax overrides the semantics of an old-style function definition.

Standard C requires that function prototype declarations override old-style function definitions. This means that the simple inclusion of a **.h** header file with prototype declarations of functions will obtain the new prototype semantics for the definitions of those functions. This feature has both disadvantages and advantages.

The advantage is that the new prototype semantics — the Pascal-style assignment-conversion of arguments to the types of the formal parameters — is obtainable by merely including a declaration in a header file. The disadvantage is that a definition can no longer be read out of context; without searching header files one cannot determine whether or not the compiler compiles the function using prototype-style semantics.

For example:

```
    file header.h:
      int func(float f,long l);

    file prog.c:
      #include "header.h"
      int func(f,l) float f; long l; {
...
}
    void sub() {
func(3, 4.4); /* Passes 3.0 and 4L via */
}  /* automatic conversion. */
```

Were `header.h` not included, the call to `func` in `sub` would pass the **int** 3 and the **double** 4.4, and probably `func` would not work properly. With the header file included, the interface for `func` is changed to prototype-style (3 is converted to **float** and 4.4 to **long**). Thus, one can find out how the compiler treats func only by searching all the header files.

To obviate the need for searching, AIX C provides a warning message whenever an old-style definition is overridden by a prototype. The warning message can be disabled by turning Off toggle **Prototype_override_warnings**.

We recommend that function definitions be written with the new prototype syntax for improved readability and reliability.

For example:

```
    file prog.c:
      int func(float f,long l) {
...
}
```

The ANSI committee permitted the override feature for two reasons: first, it would take some work to convert programs to use the new syntax in the definition (although with toggle **Print_protos**, AIX C generates the headers from old-style definitions); second, most compilers do not support prototype-form definitions, and the use of a header that is conditionally included based upon the compiler being used makes code more easily compilable by different compilers.

Read_only_strings — Default: Off

When On, string constants are placed in a read-only data section. Two or more identical string literals in the same module will appear only once in memory. It is an error to modify the storage of a string constant at run time when the toggle is On.

C string literals are not true literals because they are writable data items. This means that they cannot normally be placed in code space. Furthermore, two identical C string literals must normally be duplicated in a program's object code, because one might be modified and the other not. To avoid this, use **Read_only_strings** and **Literals_in_code**. These two toggles cause C string literals to be placed in code.

Recognize_library — Default: On

When On, the compiler may substitute in-line code for any function defined in the ANSI Standard C library. Candidates for substitution currently include the math and string functions. Future releases may encompass more of the library.

Warn — Default: On

When Off, warning messages are suppressed. The **-w** command-line option turns Warn Off initially.

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 AIX C Compiler performs two separate optimization passes—machine-dependent optimizations and machine-independent optimizations.

On the PS/2, the type of optimizations 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
- 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.

On the RT PC, both machine-dependent optimization and machine-independent optimizations are invoked by selecting the **-O** flag.

The machine-dependent optimizations are:

- Elimination of unnecessary branches
- Elimination of redundant loads and stores
- Strength reduction
- Replacement of branches with branch-with-execute instructions
- Instruction scheduling.

The machine-independent optimizations are:

- Constant propagation and folding
- Code motion of loop invariants
- Unused variable elimination
- Dead code elimination
- Idiom recognition
- Common subexpression elimination
- **Register** allocation
- Value recognition
- Copy propagation
- Short circuiting
- Integrated return/assignment.

Optimization Considerations (AIX/370)

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.

The compiler translates each function definition into an internal representation that materializes every computation required by the target machine, down to the loading and storing of registers. The internal representation assumes the existence of an unlimited number of virtual registers. The register allocation phase maps these virtual registers into the set of machine registers.

The internal representation is maintained in the form of a flow graph on which a series of optimizations is performed. Each node of the graph represents a block of code that has no more than one entry point or exit point. The following is a brief description of the optimizations, on the 370, in roughly the order they are performed:

- Dead Code Elimination

  The compiler eliminates any block of code that has no predecessor block (that is, there exists no path to the block). This phase is invoked from other phases when it is potentially valuable to do so. For example, constant propagation may convert a conditional jump into an unconditional jump and thus eliminate a path out of the associated block.

- Constant Propagation

  When the compiler sees a variable being referenced such that the last value V assigned to that variable was a constant, the reference is replaced with V. For example:

```
int debug;
...
debug = 0;
...
if (debug){    /* Constant propagation back-substitutes 0. */
...
}
```

- Constant Expression Folding

  Arithmetic expressions whose operands are constants are evaluated at compile time. Constant propagation often exposes such expressions that are not otherwise apparent.

- Local Common Subexpression Elimination (Local CSE)

  The compiler divides the flow graph into a minimum number of sub-graphs such that each sub-graph is a tree (that is, each node not having more than one predecessor but any number of successors). Common subexpressions are eliminated across nodes of each sub-graph through a technique known as value numbering. It recognizes the commutativity of operations so that "A + B" yields the same value as "B + A." It also keeps track of copy propagation so that, in the following example, the expressions a + b and c + d would be recognized as redundant:

```
int a,b,c,d,e,f;
...
e = a+b;
d = a;
c = b;
f = c+d; /* <== Same value as a+b above. */
```

- Global Common Subexpression Elimination

  In this phase, common subexpressions are eliminated across the entire flow graph. Unlike the local CSE phase, the analysis is restricted to arithmetic identities.

- Invariant Expression Removal from Loops

  Any register computation that is invariant within a loop is moved above the loop.

- Live/Dead Analysis

  Computations whose results are never used are eliminated. Dead stores are also eliminated.

- Strength Reductions

  Multiplication by a constant is converted to a series of additions, subtractions, and shifts. Division and modulo by a integer power of two are converted to a shift or masking operation.

- Global Register Allocation

  Register allocation is performed by building an interference graph and *coloring* it. Each node of the interference graph represents a virtual register. Each edge of the graph designates two registers that are alive at the same time and therefore must be mapped to different machine registers. The coloring process assigns each node to a machine register such that no two adjacent nodes are mapped to the same register.

Some ANSI-Required Specifics (AIX/370)

ANSI document X3J11/88-090 requests that each C implementation provide the following system specifics.

- Identifiers

  There is no imposed limit on the number of significant characters in an identifier. Casing is preserved.

- Characters

  Source and execution character sets are both Standard ASCII. Each character in the source character set maps into the identical character in the execution character set. Without exception, all character constants map into some value in the execution character set.

  A character is stored in a byte, and there are four bytes in an **int**.

  The type specifier **char**, when not accompanied by an adjective, denotes an unsigned character type. However, this can be changed by turning Off the toggle **Char_default_unsigned**.

- Integers

  Integers are represented in two's-complement binary form. The following table illustrates the ranges of values to which the various integer types are restricted:

Type	Range		
signed **char**	-128	to	127
unsigned **char**	0	to	255
short	-32,768	to	32,767
unsigned **short**	0	to	65,535
int	-2,147,483,648	to	2,147,483,647
unsigned **int**	0	to	4,294,967,296
long	-2,147,483,648	to	2,147,483,647
unsigned **long**	0	to	4,294,967,296

  An integer is converted to a shorter signed integer or **int** bit field by bit truncation; that is, when an X-bit value is to be stored into a Y-bit receptacle, where X > Y, the rightmost Y bits of the first value are stored.

An **unsigned integer** U is converted to a **signed integer** I where **sizeof** (U) = **sizeof**(I) by transferring the bits of U into I, whether or not the value of U is representable in I. For example, **(short int)(short unsigned)** 65535 is the **short int** value -1. The **sizeof** operator returns an **int**, unless the value returned is not representable in **int**, in which case unsigned **int** is the type returned.

The result of a bitwise operation on a **signed integer** is the same as if the integer were treated as unsigned.

The sign of the remainder on integer division is the same as the sign of the dividend.

The right shift of a signed integral type is arithmetic; that is, the sign bit is propagated to the right.

- Floating point

  Floating-point data is represented in the floating-point format defined in *IBM System 370 Principles of Operation* manual. The default rounding mode is to "truncate toward zero."

  When a negative floating-point number is truncated to an integral type, the truncation is toward zero. Thus -2.7 is truncated to -2 and -1.2 to -1.

- Pointers

  The difference of two pointers is type **int**.

- Registers

  The compiler attempts to map each **auto-** (or **register-**) class variable to a register provided that its type is appropriate and its address is not required.

- Structures, unions, and bit fields

  Signed and unsigned bit fields are supported. A bit field declared as **int** is treated as **unsigned int**. A bit field declared as **signed int** will be interpreted as signed.

- Preprocessing directives

  A single-character constant in a constant expression controlling conditional inclusion is always non-negative in value, ranging from 0 to 255.

Floating-Point Exceptions for AIX PS/2

The following six operations or types of operations can produce an AIX floating-point exception. They apply to PS/2s configured with or without the Intel 80387 floating-point coprocessor.

- Stack operations—stack overflow, underflow
- Arithmetic operations—includes invalid operations defined in IEEE Standard 754
- Division by zero
- Denormal number
- Overflow
- Underflow.

Floating point exception interrupts are disabled by **default**. You can use the **fp_control**, **fp_exmask**, **fp_exunmask**, **fp_round**, **fp_precision**, **fp_getex**, **fp_getprecision**, **fp_getcw**, **fp_restore**, and **fp_getround** subroutines to manipulate the runtime environ-

ment with respect to floating-point handling and to set the desired rounding and precision modes. (See *AIX Operating System Technical Reference* for more information about these subroutines.)

The following is an example of a program that writes the current control word and then enables the divide by zero interrupt. Note the inclusion of the header file **/usr/include/sys/fpcontrol.h**, which defines the constant FPM_DIVIDE_0. Also, see **fp_getcw(3)** for the library calling option.

```
#include <sys/fpcontrol.h>
main ()
{
     double x,y;
     printf("%d\n",fp_getcw());
     fp_exunmask(FPM_DIVIDE_0);
     x = 0.0;
     y = 1.0;
     y = y/x;
}
```

Any combination of the exception mask constants defined in **fpcontrol.h** can be ORed together to form the desired argument to the appropriate subroutine.

C Programs Under AIX

As illustrated in Figure 2-1 on page 2-22, the four main steps in creating an executable C program under the AIX Operating System are:

1. Create your program using a text editor and store it with a **.c** extension.
2. Compile your source program to generate an object file. The compile process is described in detail below.
3. Link the output with the AIX system linker **ld** to create an executable file.
4. Run the program.

Steps 2 and 3 are executed by the **cc** command, which includes the **libc.a** library by default.

Note: The RT PC C Compiler also includes the **librts.a** library. Optional user libraries may be added to the **cc** command line by using the **-l** option flag.

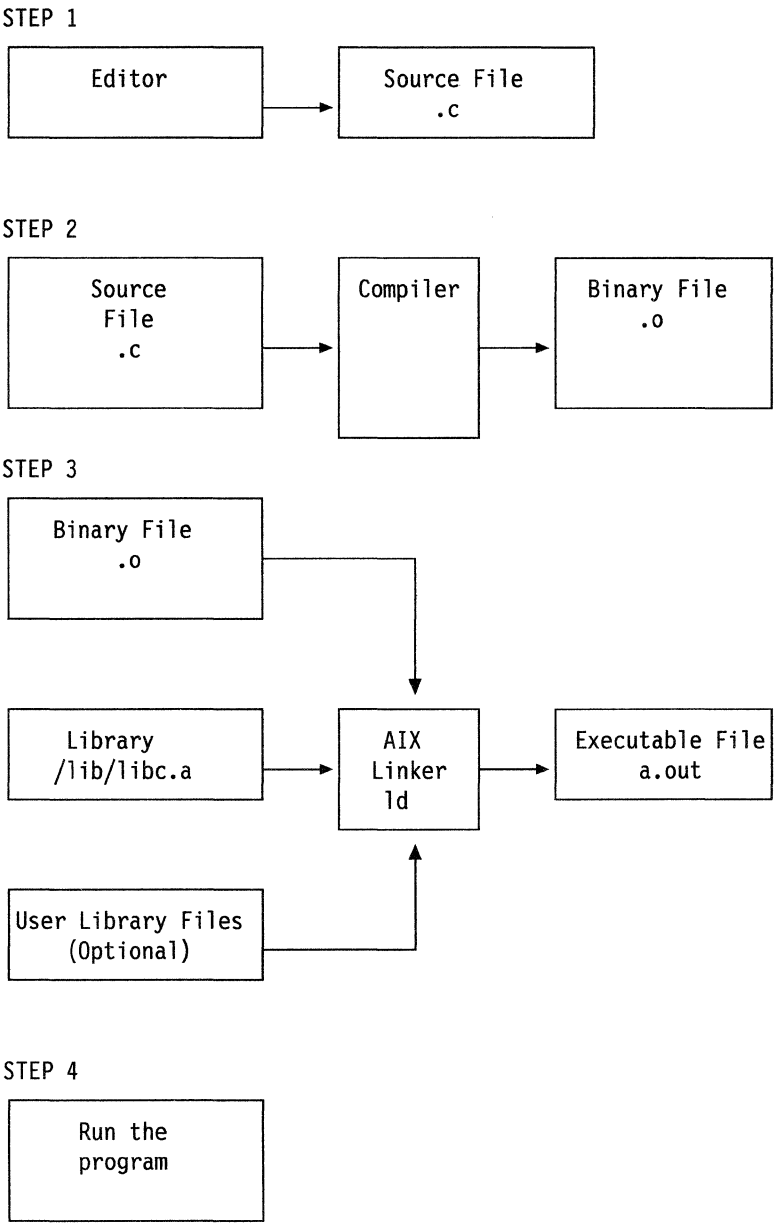


Figure 2-1. Creating a C Program Under AIX

PS/2 and RT PC Compilation Process

As illustrated in Figure 2-2 on page 2-23, the compiler follows these steps when invoked:

1. The source file is passed to the **cpp** command, the C preprocessor, which produces a preprocessed C source file.
2. The preprocessed file is passed to **vsc**, the C compiler front end, which produces intermediate code with an **.i** extension.
3. The **.i** file is passed to **vspass2**, the code generator, which produces intermediate code with a **.j** extension.

4. The **.j** file is passed to **vspass3**, the code formatter, which produces a binary file with an **.o** extension.
5. The **.o** file is passed to the AIX linker (**ld**) which creates an executable file.

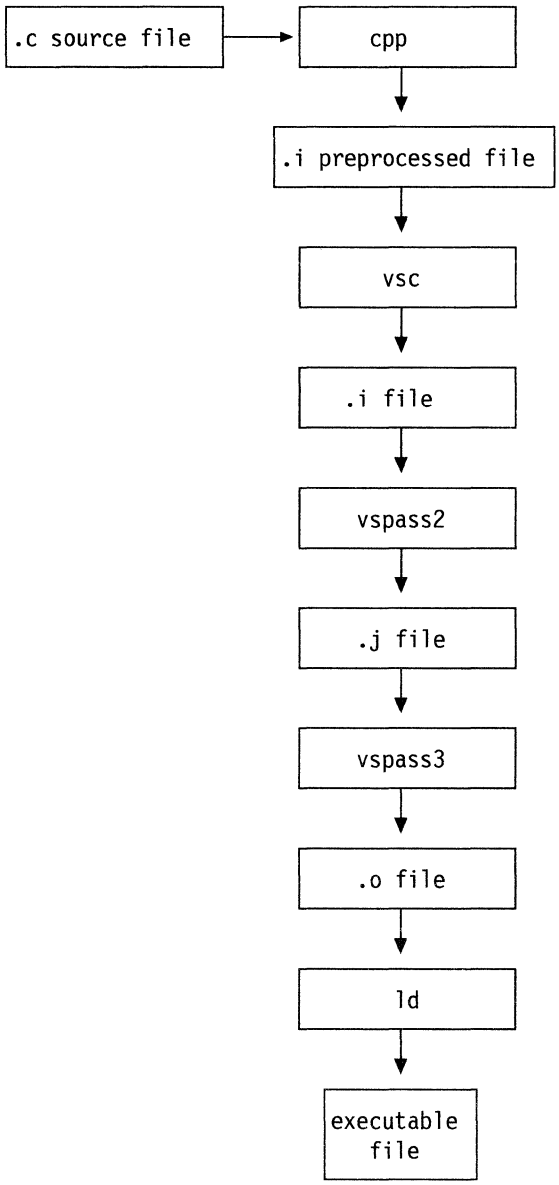


Figure 2-2. PS/2 and RT PC Compilation Process

System/370 Compilation Process

As illustrated in Figure 2-3, the compiler follows these steps when invoked:

1. The source file is passed to the **cpp** command, the C preprocessor, which produces a preprocessed C source file.
2. The preprocessed file is passed to **hc1com** and **hc2com**, which produces assembler code.
3. The **assembler** file is passed to **as**, the assembler, which produces a binary file with an **.o** extension.
4. The **.o** file is passed to the AIX/370 linker (**ld**) which creates an executable file.

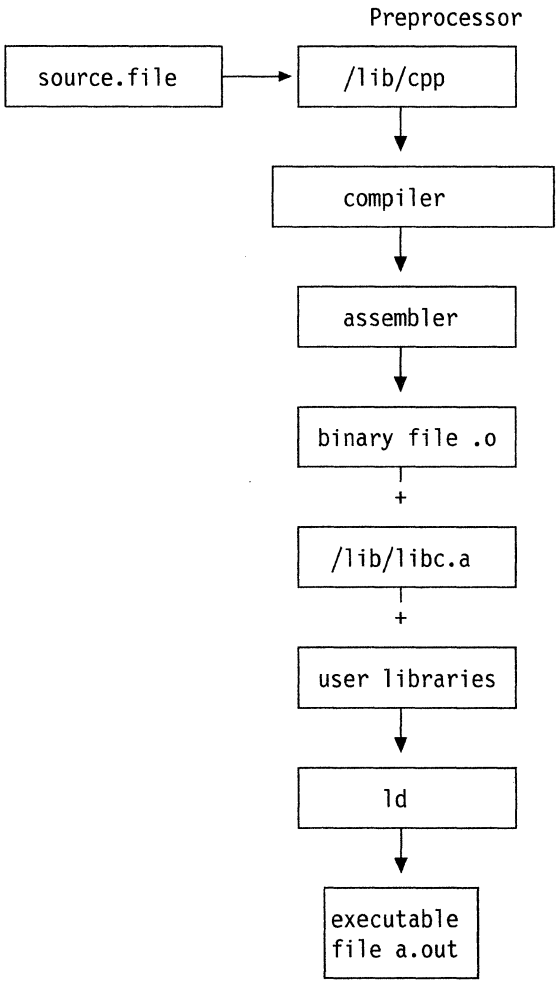


Figure 2-3. 370 Compilation Process

Note: The compiler consists of two passes: **/lib/hc1com** and **/lib/hc2com**.

Chapter 3. Data Representations on PS/2

CONTENTS

About This Chapter

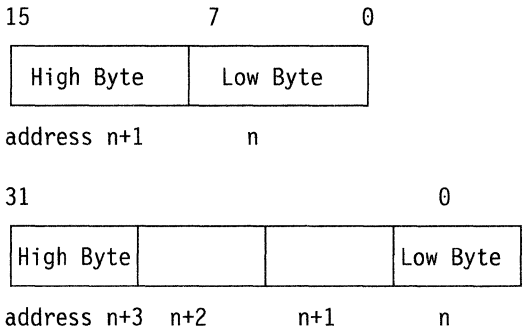
This chapter describes how the AIX PS/2 C Compiler represents data in storage. It also describes characteristics of the C data types as defined by the C Compiler.

Data Representations on PS/2

The following terms are used as defined by the Intel 80386 microprocessor:

- A byte contains 8 bits.
- A word contains 16 bits.
- A doubleword contains 32 bits.

The following example shows the byte order of a word and doubleword stored in memory:

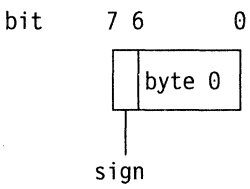


Although the 80386 microprocessor does not absolutely require alignment of data, there is a performance benefit in aligning 2-byte and 4-byte data. For this reason, the AIX PS/2 C Compiler seeks to align 2-byte entities on even addresses and 4-byte entities on addresses which are multiples of 4; 8-byte entities are aligned on addresses which are multiples of 4.

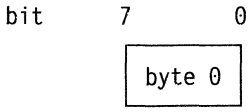
Integral Representation

The following information describes characteristics of the integral data types including the storage size in number of bits, the range of values represented by the data type, and the alignment. Two's complement format is used for all signal integer types.

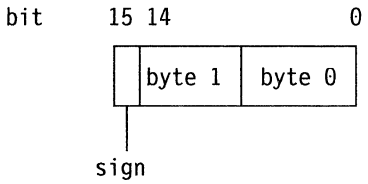
Data Type	Bits	Range	Alignment
signed char	8	-128 to 127	byte



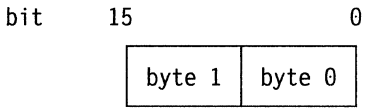
Data Type	Bits	Range	Alignment
char unsigned char	8	0 to 255	byte



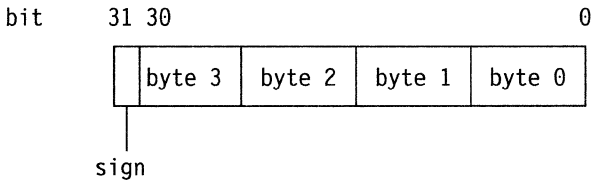
Data Type	Bits	Range	Alignment
short short int signed short int	16	-32768 to 32767	word



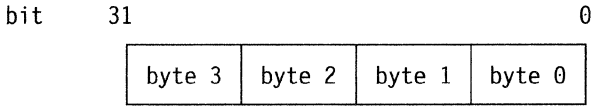
Data Type	Bits	Range	Alignment
unsigned short unsigned short int	16	0 to 65535	word



Data Type	Bits	Range	Alignment
int long long int signed signed int signed long signed long int	32	-2147483648 to 2147483647	doubleword



Data Type	Bits	Range	Alignment
unsigned unsigned int unsigned long unsigned long int	32	0 to 4294967295	doubleword

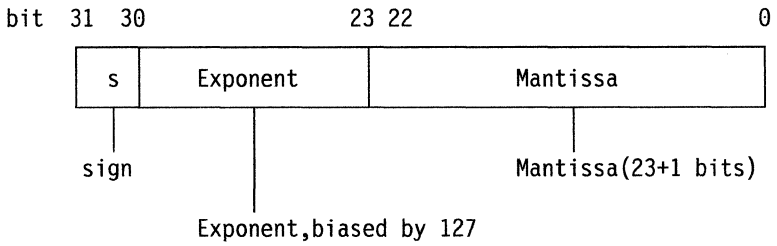


Floating-Point Representation

The following information describes characteristics of the floating-point data types. The AIX C Compiler supports both single and double precision floating-point data, according to the ANSI/IEEE 754-1985 standard.

Single Precision

Data Type	Bits	Range	Alignment
float	32	Approximately -3.37e+38 to 3.37e+38	doubleword

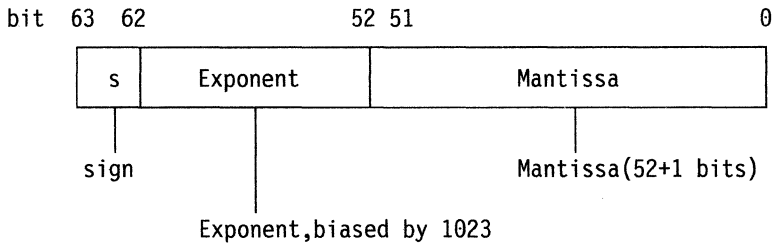


The three fields of a single-precision floating-point number are:

- A sign bit designated by *s* in the diagram. The sign bit is 1 only if the floating-point number is negative.
- An 8-bit biased exponent.
- A 23-bit mantissa with the high-order 1-bit hidden.

Double Precision

Data Type	Bits	Range	Alignment
double long double	64	Approximately -1.67e+308 to 1.67e+308	doubleword



The three fields of a double-precision floating-point number are:

- A sign bit designated by *s* in the diagram. The sign bit is 1 only if the floating-point number is negative.
- An 11-bit biased exponent.
- A 52-bit mantissa with the high-order 1 bit hidden.

A floating-point number is represented by the form:

```
      sign          exponent-bias
(-1)     *      2                *  1.fraction
```

where fraction is the pattern of bits in the mantissa.

Representation of Extreme Numbers

zero (signed) is represented by an exponent of zero and a mantissa of zero.

signed infinity (or affine infinity) is represented by the largest value that the exponent can assume (all ones), and a zero mantissa. When infinity is printed, it appears as +INF or −INF.

Not-a-Number (NaN) is represented by the largest value that the exponent can assume (all ones), and a non-zero mantissa. The sign is usually ignored. When NaN is printed, it appears as QNaN (for *quiet* NaN), or SNaN (for *signalling* NaN).

denormalized numbers are a product of gradual underflow. They are non-zero numbers with an exponent of zero. The form of a denormalized number is:

```
      exponent-bias+1
2                     *  0.fraction
```

where fraction is the number of bits in the mantissa.

Normalized numbers are said to contain a hidden bit, providing for one more bit of precision than would normally be the case. 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.

Arrays

Arrays are stored in row major order, such that the last subscript in a multi-dimensional array varies fastest. For example, an array dimensioned as:

```
x[3][2]
```

is stored in this order:

```
x[0][0], x[0][1],      x[1][0], x[1][1],      x[2][0], x[2][1]
```

Pointers

Pointers occupy four bytes and are unsigned. The NULL value pointer is represented by zero (0). All arithmetic on pointers is computed with four bytes of accuracy.

Structures

Structure members are stored sequentially in the order in which they are declared. The first member is stored at an offset of zero relative to the address of the structure. Subsequent members are given the next available offset consistent with their size. Since **char** and **unsigned char** occupy a single byte, members of these types are given the next free byte. Larger sized members, including those of type **int** and **float** are assigned the next available doubleword offset. The **struct** and **union** variables are aligned according to the maximum alignment of their members. Bytes that are skipped are not reused to store subsequent **char** values.

Bit fields are assigned bits within the next available doubleword beginning at the least significant bit. Subsequent bit fields are kept in adjacent bits, if they will fit in the same doubleword. If not, the upper portion of the current doubleword is left unused, and the bit field is assigned the least significant portion of the next doubleword.

Chapter 4. Data Representations on RT PC

CONTENTS

About This Chapter

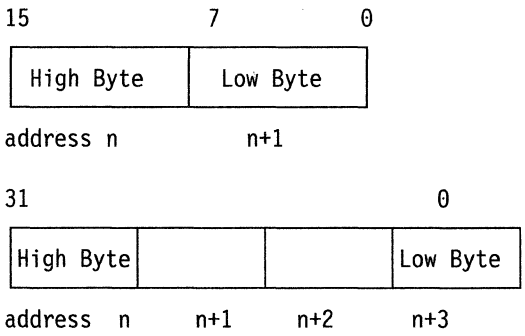
This chapter describes how the AIX RT PC C Compiler represents data in storage. It also describes characteristics of the C data types as defined by the C Compiler.

Data Representations on RT PC

The following terms are used as defined by the IBM processor:

- A byte contains 8 bits.
- A halfword contains 16 bits.
- A word or fullword contains 32 bits.

The following example shows the byte order of a halfword and fullword stored in memory:

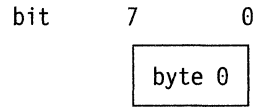


The RT PC microprocessor requires alignment of data. For this reason, the AIX RT PC C Compiler aligns 2-byte entities on even addresses and 4-byte entities on addresses which are multiples of 4; 8-byte entities are aligned on addresses which are multiples of 4.

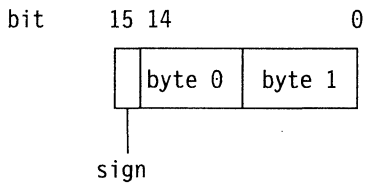
Integral Representation

The following information describes characteristics of the integral data types including the storage size in number of bits, the range of values represented by the data type, and the alignment. Two's complement format is used for all signed integer types.

Data Type	Bits	Range	Alignment
char unsigned char	8	0 to 255	byte

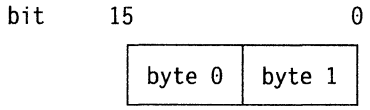


Data Type	Bits	Range	Alignment
short short int	16	-32768 to 32767	halfword



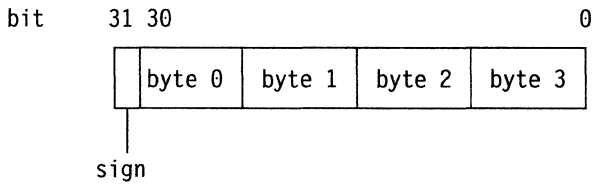
Note: An **external short** data type is fullword aligned.

Data Type	Bits	Range	Alignment
unsigned short unsigned short int	16	0 to 65535	halfword

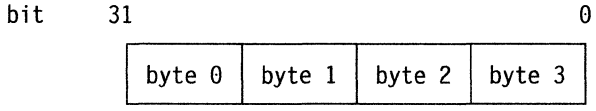


Note: An **external short** data type is fullword aligned.

Data Type	Bits	Range	Alignment
int long long int	32	-2147483648 to 2147483647	fullword



Data Type	Bits	Range	Alignment
unsigned unsigned int unsigned long unsigned long int	32	0 to 4294967295	fullword

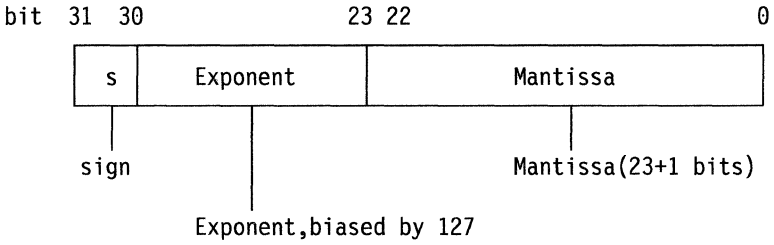


Floating-Point Representation

The following information describes characteristics of the floating-point data types. The AIX C Compiler supports both single and double precision floating-point data, according to the ANSI/IEEE 754-1985 standard.

Single Precision

Data Type	Bits	Range	Alignment
float	32	Approximately -3.37e+38 to 3.37e+38	fullword

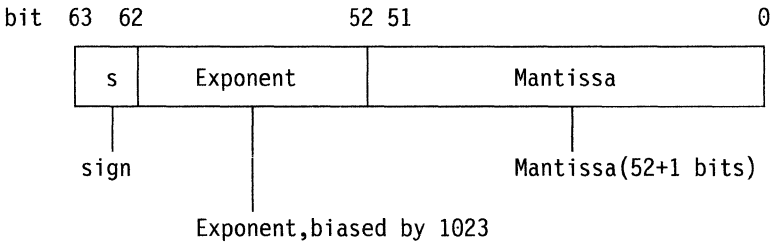


The three fields of a single-precision floating-point number are:

- A sign bit designated by *s* in the diagram. The sign bit is 1 only if the floating-point number is negative.
- An 8-bit biased exponent.
- A 23-bit mantissa with the high-order 1-bit hidden.

Double Precision

Data Type	Bits	Range	Alignment
double long double	64	Approximately -1.67e+308 to 1.67e+308	fullword



The three fields of a double-precision floating-point number are:

- A sign bit designated by *s* in the diagram. The sign bit is 1 only if the floating-point number is negative.
- An 11-bit biased exponent.
- A 52-bit mantissa with the high-order 1 bit hidden.

A floating-point number is represented by the form:

```
          sign          exponent-bias
       (-1)     *      2               *  1.fraction
```

where fraction is the pattern of bits in the mantissa.

Representation of Extreme Numbers

zero (signed) is represented by an exponent of zero, and a mantissa of zero.

signed infinity (or affine infinity) is represented by the largest value that the exponent can assume (all ones), and a zero mantissa. When infinity is printed, it appears as +INF or −INF.

Not-a-Number (NaN) is represented by the largest value that the exponent can assume (all ones), and a non-zero mantissa. The sign is usually ignored. When NaN is printed, it appears as QNaN (for *quiet* NaN), or SNaN (for *signalling* NaN).

denormalized numbers are a product of gradual underflow. They are non-zero numbers with an exponent of zero. The form of a denormalized number is:

```
          exponent-bias+1
       2                      *  0.fraction
```

where fraction is the number of bits in the mantissa.

Normalized numbers are said to contain a hidden bit, providing for one more bit of precision than would normally be the case. 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.

Arrays

Arrays are stored in row major order, such that the last 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[1][2],     x[2][1], x[2][2],     x[3][1], x[3][2]
```

Pointers

Pointers occupy four bytes and are unsigned. The NULL value pointer is represented by zero (0). All arithmetic on pointers is computed with four bytes of accuracy.

Structures

Structure members are stored sequentially in the order in which they are declared. The first member is stored at an offset of zero relative to the address of the structure. Subsequent members are given the next available offset consistent with their size. Since **char** and **unsigned char** occupy a single byte, members of these types are given the next free byte. Larger sized members, including those of type **int** and **float** are assigned the next available fullword offset. The **struct** and **union** variables are aligned according to the maximum alignment of their members. Bytes that are skipped are not reused to store subsequent **char** values.

Bit fields are assigned bits within the next available fullword beginning at the most significant bit. Subsequent bit fields are kept in adjacent bits, if they will fit in the same fullword. If not, the upper portion of the current fullword is left unused, and the bit field is assigned the least significant portion of the next fullword.

Chapter 5. Data Representations on System/370

CONTENTS

About This Chapter

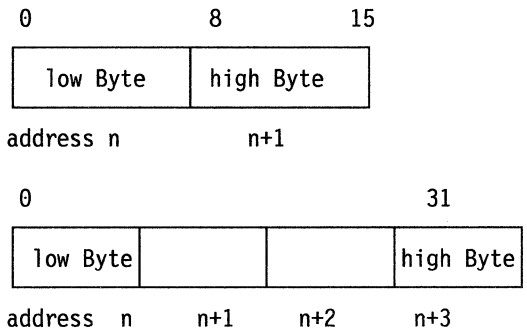
This chapter describes how the AIX System/370 C Compiler represents data in storage. It also describes characteristics of the C data types as defined by the C Compiler.

Data Representations on System/370

The following terms are used as defined by the IBM System/370 architecture:

- A byte contains 8 bits.
- A halfword contains 16 bits.
- A word or fullword contains 32 bits.

The following example shows the byte order of a halfword and fullword stored in memory:

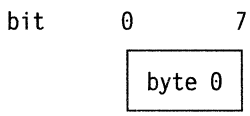


The System/370 architecture requires alignment of data. For this reason, the AIX System/370 C Compiler aligns 2-byte entities on even addresses and 4-byte entities on addresses which are multiples of 4; 8-byte entities are aligned on addresses which are multiples of 4.

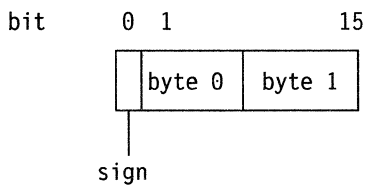
Integral Representation

The following information describes characteristics of the integral data types including the storage size in number of bits, the range of values represented by the data type, and the alignment.

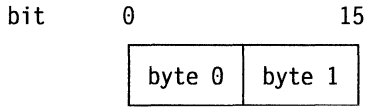
Data Type	Bits	Range	Alignment
char unsigned char	8	0 to 255	byte



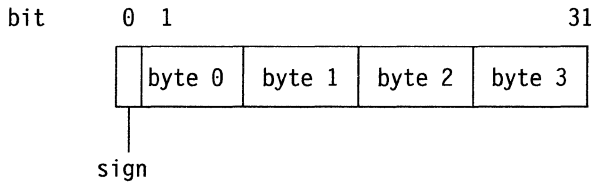
Data Type	Bits	Range	Alignment
short short int	16	-32768 to 32767	halfword



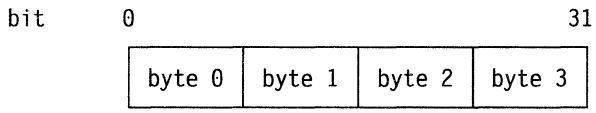
Data Type	Bits	Range	Alignment
unsigned short unsigned short int	16	0 to 65535	halfword



Data Type	Bits	Range	Alignment
int long long int	32	-2147483648 to 2147483647	fullword



Data Type	Bits	Range	Alignment
unsigned unsigned int unsigned long unsigned long int	32	0 to 4294967295	fullword

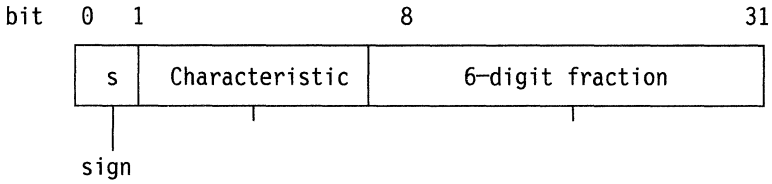


Floating-Point Representation

The following information describes characteristics of the floating-point data types. The AIX C Compiler supports both single and double precision floating-point data, according to System/370 architecture.

Single Precision

Data Type	Bits	Range	Alignment
float	32	Approximately 5.4e-75 to 7.2e+75	fullword

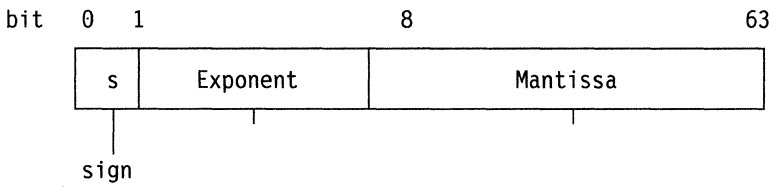


The three fields of a single-precision floating-point number are:

- A sign bit designated by *s* in the diagram. The sign bit is 1 only if the floating-point number is negative.
- A 7-bit characteristic
- A 24-bit 6-digit fraction.

Double Precision

Data Type	Bits	Range	Alignment
double **long double**	64	See "Single Precision"	fullword



The three fields of a **double**-precision floating-point number are:

- A sign bit designated by *s* in the diagram. The sign bit is 1 only if the floating-point number is negative.
- A 7-bit characteristic.
- A 54-bit 14-digit fraction.

Please refer to *IBM System 370 Principles of Operation* for more details.

Arrays

Arrays are stored in row major order, such that the last 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[1][2],      x[2][1], x[2][2],      x[3][1], x[3][2]
```

Pointers

Pointers occupy four bytes and are unsigned. The NULL value pointer is represented by zero (0). All arithmetic on pointers is computed with four bytes of accuracy.

Structures

The size of a **struct** or **union** is dependent upon whether the compiler generates padding to align members. The compiler generates such padding by default if the toggle **Align_members** is on, and does not do so by default if the toggle is off.

The size of an unpadded **union** is the size of the biggest member. The size of a padded union is the size of the biggest member, padded so that its size is evenly divisible by its alignment.

The size of an unpadded **struct** is the sum of the sizes of its members. Non-bit-field members always start on byte boundaries, and there is no padding between members, except in the case of bit members. The size of a padded **struct** is the sum of the sizes of its members, including alignment padding between members. It is padded so that its size is evenly divisible by its alignment.

Notes:

1. A **struct** or **union** is aligned according to the most stringent requirements among its members.
2. The size of **enum** types depends on the status of the **Long_enums** toggle. If the toggle is Off, the type is mapped to the smallest of a byte, half word, or full word, such that all the values can be represented. If the toggle is On, the **enum** maps to a full word (matching the Portable C Compiler convention). See "Compiler Toggles (AIX/370)" on page 2-11.

Storage Classes

Each **static** variable is placed in either the BSS section or the DATA section — the latter if it is initialized.

Each global variable with no **extern** specifier that is not initialized is defined as a common block; if it is initialized, it is mapped into the DATA section and given the global attribute. Each **extern** variable is given the global and undefined attributes. Each local **static** variable is placed in either the BSS section or the DATA section.

Each **auto** variable is assigned either to a machine **register** or to storage in the routine's *stack frame*. The compiler attempts to place each **auto**-classed variable in a **register** provided the variable's type is appropriate and its address is not required. In a function containing calls to **setjmp**, **auto** variables are not mapped to registers, so that their values are not lost across such calls.

Chapter 6. Mixing Languages and Linkage Convention on PS/2

CONTENTS

About This Chapter

This chapter describes the conditions, rules, and conventions that the user or programmer must observe when mixing program elements of two or more AIX PS/2 high-level languages. Detailed information is provided for the VS FORTRAN, VS Pascal, and C Compilers on the PS/2.

Mixing Languages on PS/2

The IBM AIX PS/2 language system permits the mixing of elements from different languages in a single program. This chapter assumes you are familiar with the languages you want to mix; the elements of the languages are not described here in detail. Appendix C, "Program Examples for Mixing Languages" gives sample programs that show how to pass parameters between different languages.

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[1] standard data representations are used as much as possible. Table 6-1 shows some of the correspondence among languages.

Table 6-1. Correspondences of Data Types Among Languages

FORTRAN IBM and R1 Modes	**FORTRAN VX Mode**	**Pascal**	**C**
Logical*1	Logical*1	Boolean	
	Logical*2		
Logical, Logical*4	Logical, Logical*4		
Integer*1	Integer*1 Byte	packed/unpacked CHAR element	signed char
Integer*2	Integer*2		short
Integer*4	Integer*4	Integer	int
Real, Real*4	Real, Real*4	Shortreal	float
Real*8	Real*8	Real	double long double
Complex	Double Precision, Complex		
Complex*8	Complex*8		
Complex*16	Complex*16		
Character	Character	packed array of Char	char
		String	

Note: Table 6-1 shows how the languages represent data internally in the computer's memory rather than how data are passed between program units.

[1] IEEE 754 Floating-Point Standard

As Table 6-1 shows, some data types 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 more 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.

Character Variables

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.

When a FORTRAN character variable is passed as a parameter, the address of the beginning of the character is passed along with a hidden parameter which is the **static** length of the character string. The hidden parameter is added to the end of the declared parameter list.

Pascal's character-variable data types are **String** and **packed array of Char**. The **String** data type has a 4-byte **double** 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. However, **packed array of Char**, like the type **Character** in FORTRAN, 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) A(3,2)
```

Pascal and C matrices are stored in computer memory by row (row major order). For example, an array in Pascal declared as:

```
A : array [1..3,1..2] of Real
```

is stored in this order:

```
A[1,1]      A[1,2]  A[2,1]  A[2,2]      A[3,1] 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 contained in the language run-time libraries. They are bound to your program during the final linking process by the AIX linker **ld**.

When you compile a program using the **cc** or **vs** command, the appropriate libraries are linked in the proper order. The libraries are:

Language	Libraries
C	**libc.a**
VS Pascal	**libvssys.a** **libc.a**
VS FORTRAN	**libvsfor.a** **libvssys.a** **libc.a**

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.

Input/Output Initialization: When mixing languages, special handling is required when calling AIX VS Pascal or FORTRAN subroutines that perform input/output.

An initialization routine must first be called from the main program so that the input/output buffers and floating-point information are properly set up. These routines do not require parameters. The initialization routines and run-time libraries in which they are contained are:

Language of Called Routine	Initialization Routine	Library
VS Pascal	**vspio**	**libvssys.a**
VS FORTRAN	**vsfio**	**libvsfor.a**

Input/Output Termination: If a C function is called from a AIX VS FORTRAN or Pascal main program, the termination routine **vstio** in **libvssys.a** must be called upon to **return** from the C function to properly flush the input/output buffers.

Whenever you mix languages, you need first to become familiar with the requirements of each particular subroutine and each language's argument-passing requirements.

Designation of Entry Points and Other Global Symbols

An entry point is the address of the first instruction of a subroutine or function. If an entry point is not known at compile time, the relative entry point is determined at link time when all the separately compiled routines are brought together with the system library (or libraries) to produce an executable binary file.

An entry-point reference in FORTRAN is designated in a CALL statement or by the invocation of a function in an expression. An entry-point reference in Pascal is designated by a procedure or function call, and in C by a function call. An entry-point reference in Assembler code is designated by a reference to an external symbol.

When compiled programs are linked, the entry points referenced in a calling program cause the linker to add an entry to a table of symbols. The linker then needs to associate a start address with each symbol in the table, which it does when it finds the called routine. The entry point for a called FORTRAN routine is designated by a subroutine or function declaration. The entry point for a called Pascal routine is designated by a procedure or function declaration at the global scope. The entry point for a called C function is designated when the function is defined. The entry point for a called Assembler routine is designated by a global declaration.

When compiled program units of mixed languages are linked together, it is important to note that the compilers may maintain different numbers of significant characters in global names, which include entry points. Also, all local and global names in FORTRAN and Pascal are made lowercase. As a result, for example, a FORTRAN and a Pascal program cannot call a C function whose name is not all lowercase. The AIX PS/2 C compiler creates external symbols which are in the same **case** as the user has declared them. Thus a C routine declared as *Upper* cannot be called from FORTRAN or Pascal since both compilers would produce an external symbol *upper* regardless of the user specified name.

The entry-point name you assign to local Pascal procedures (not visible in the global scope) stays the same.

Argument-Passing Mechanisms

When control is passed from one routine to another by a call, operands for the called routine to act upon are usually also passed. When finished, the called routine usually has results to report and needs to know where control is to be returned in the calling routine. The AIX PS/2 language system provides this information by using one or both of the following:

- Intel 80386 or Intel 80387 registers
- Data structure known as the stack.

The 80386 Registers

The AIX PS/2 Operating System is built around the Intel 80386 microprocessor, which has sixteen 32-bit internal registers. Table 6-2 contains more information on these registers:

Table 6-2. 80386 Registers

Registers	Type	Use
EAX EBP EBX ESP ECX ESI EDX EDI	General purpose registers	Logical and arithmetic operations. Address computations[2]
CS ES DS FS SS GS	Segment registers[3]	Initialized by operating system and should not be changed.
EFLAGS	Status register	
EIP	Instruction pointer register	

The 80387 Registers

AIX PS/2 floating-point performance is optionally provided by the Intel 80387 floating-point numeric processor. The presence of the Intel 80387 chip is determined at the time of the first floating-point instruction for a given process. If the optional 80387 chip is not installed in the system, then the behavior of the 80387 chip is exactly duplicated by software emulation. This emulation is transparent (other than at execution time) to you. The 80387 chip has a floating-point stack architecture which consists of eight 80-bit extended precision registers.

The Stack

Although it is not important to be familiar with the stack when coding in only one language, it is helpful when languages are being interfaced. The stack is a last-in first-out data structure that is located, initialized, and used by the AIX PS/2 language system. Stacks are thought of as having information *pushed* onto them and having information *popped* off them. This is because each time a routine is called, the compiler pushes information about the calling arguments followed by the return address onto the stack. If routines call other routines before a return is made, the

---

[2] The exception is ESP which may be used as an index operand.

[3] A flat address model has been adopted by the AIX PS/2 Operating System which results in a real address space of 4 gigabytes. The segmented features of the Intel 80386 chip architecture are not used.

stack continues to grow; as control returns to the calling routines, the stack becomes smaller. The called routine removes the return address and the local data space. The parameter list of the called routine is popped off the stack by the calling routine.

The physical memory address of the stack is usually not relevant. The stack grows toward smaller absolute addresses, so pushing data onto the stack consists of decrementing the value stored in **ESP** and writing to the address in memory indicated by **ESP**.

The data written to the stack by the calling routine depends both on the language and on the data type of the arguments being passed to the called routine. Usually either the addresses of the arguments (used in the call-by-reference mechanism) or the values of the arguments (used in the call-by-value mechanism) are placed on the stack. The languages also place a 4-byte return address on the stack, which is the last item pushed on to the stack by the calling routine. This return address indicates where execution is to resume after the called routine finishes executing.

Subroutine Linkage Convention

The AIX PS/2 language system adheres to the following linkage convention:

Table 6-3. AIX PS/2 Language Linkage Convention

Registers	Type	Use
EAX ECX EDX	Scratch registers	Need not be preserved
EBX ESI EDI	Register variables	Must be preserved across calls
EBP	Frame pointer	Must be restored at the end of a call
ESP	Stack pointer	Should not be popped by called routine

All parameters are passed in reverse order and the calling routine has the responsibility for cleaning up the stack.

Routines written in FORTRAN, Pascal, and C respect these register usage conventions by preserving the registers listed in Table 6-3 to the states they were in before their call. All routines used with the AIX PS/2 languages must also preserve these registers.

Parameter-Passing Convention

For ease of language interfacing, all AIX PS/2 languages follow the same calling convention. Parameters are passed to the called routine on the stack. They are pushed onto the stack in the reverse order from the way that they were declared. That means the last parameter is pushed first on the stack, and the first parameter is pushed last.

All arguments to FORTRAN subroutines and functions are passed by reference. For every argument except a character argument, a 32-bit pointer to the object is pushed onto the stack. Character parameters are passed as a 32-bit pointer to the Character object. A 32-bit value which is the static length of the Character object is passed at the end of the actual argument list.

In Pascal, if the called procedure is not a procedure or function at the global level, the static link is the last thing pushed onto the stack before the routine is called. Pascal call by reference parameters always have a 4-byte pointer to the variable pushed onto the stack. Pascal call by value parameters are pushed onto the stack. Pascal Reals are pushed onto the stack as 8 bytes. Pascal sets are pushed onto the stack with the least significant element in the least significant word. Thus, the representation of a set on the stack is the same as the representation in memory. If a Pascal value parameter is longer than 4 bytes and not a double or a set, the address of the data is pushed onto the stack, and the called procedure or function copies the data into local storage.

Pascal procedure and function parameters are pushed as follows: the address of the procedure or function is pushed onto the stack. The static link is then pushed onto the stack if the procedure or function is not at the global level. If the procedure or function is at the global (outermost) level, the value 0 (nil) is pushed onto the **stack** instead of the static link.

All arguments to C functions are passed by value. Actual arguments which are expressions are evaluated before the function reference. The result of the expression is then pushed onto the stack. Arrays are passed by the address of the first element of the array as a 4-byte quantity.

All C arguments except **double**, **struct** and **union** arguments are passed as four-byte values; **double** arguments are passed as eight-byte values. All float values are passed as doubles. The **struct** and **union** arguments are passed by value, regardless of the number of bytes required to pass the argument.

When C functions which return **structs** or Pascal functions which return **records** are called, the calling routine allocates sufficient space for the structure and passes a hidden first parameter to the called routine. Remember, the first parameter is the last one pushed on the stack. This parameter is a pointer to the allocated space.

Function Results

The AIX PS/2 languages **return** the results of functions in **register EAX** for non-floating-point values. Floating-point values are returned at the top of the Intel 80387 floating-point stack. C functions which return **structs** or Pascal functions which **return records return** a 32-bit pointer to the **struct** or **record** in **EAX**.

Stack Frame

A typical stack frame for a function call is created by the following sequence: The caller pushes the function arguments in reverse order (right to left) from the function declaration. If the called routine returns a C **struct** or Pascal **record**, then the caller pushes the address of a buffer allocated by the caller for the **struct** or **record**. The **return** address is pushed on the stack by executing the call instruction. The base pointer register **EBP** is pushed and **EBP** is made to point to the stack top, which now contains its old value. The called function allocates all of its local data space. If the allocated registers need to be used in the called routine, then their values may be saved on the stack. The return result is loaded into the **EAX register** for non-floating-point values. C functions returning a **struct** and Pascal functions returning a

record load the address of the buffer into **EAX**. Floating-point values are returned at the top of the 80387 numeric processor stack. If the registers **EBX**, **EDI**, and **ESI** were saved, then they are restored. The called function then reloads the original value of the **EBP** register, removes its local space from the stack, and returns. The caller pops the arguments off the stack.

The following diagram illustrates a typical stack frame after a function call:

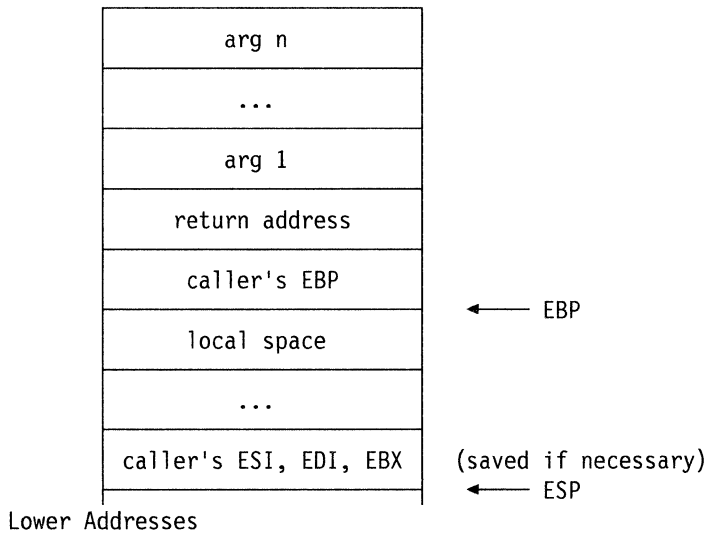


Figure 6-1. Typical Stack Frame

FORTRAN Argument-Passing Conventions

FORTRAN uses the call-by-reference mechanism for passing arguments. This means that a 4-byte address of each argument is pushed onto the stack from the last argument to the first when a call is made. The return address of the calling program unit is pushed onto the stack after the last argument address.

Character data types in FORTRAN are treated differently. When an argument is a character variable, 8 bytes are placed on the stack: a 4-byte address in the location of the argument in the argument list and the 4-byte static character count of the variable at the end of the argument list. This length is the size of the variable made known to the compiler by the character type specification, and is not the number of characters that happen to be stored in the variable.

When an array is passed as an argument in a subroutine call or external function reference, the address of the first byte of the first element is pushed onto the stack. If an element of an array is passed as an argument, the address of that element is pushed.

Actual arguments that are expressions are evaluated before the subroutine call or function reference. The result of the expression is assigned to a temporary storage location and the address of the location is pushed onto the stack. When storing a numeric expression, the temporary storage location is usually 4 bytes (8 bytes for double-precision values).

When an external function is referenced, the result is returned in register **EAX** (or at the top of the floating-point stack if the result is a floating-point number). One exception to this occurs if the external function returns character data. In this case, the 4-byte address of the result is returned in register **EAX**.

When the called program unit finishes its computation, the stack must be cleaned up so that it does not continue to grow indefinitely. When control is returned to the calling function, all arguments are discarded from the stack.

Example A:

```
Character*5 STR
Integer DWORD
A=5.0
CALL XYZ(4.0*A,STR,DWORD)
.
.
END
```

When the call in Example A is made, the 4-byte integer 5 is pushed onto the stack. This value is the static length of **STR**. The address of the 4-byte variable *DWORD* is then pushed onto the stack. Then the address of the beginning of the character string is pushed onto the stack. Then the expression (4.0*A) is evaluated and a temporary storage location is set up for it. The address of this location is the last parameter you pushed onto the stack. Lastly, the return address in the calling program unit is stored in 4 bytes.

Figure 6-2 on page 6-11 illustrates a stack after the call in Example A is made (`msb` and `lsb` indicating the most and least significant bytes, respectively).

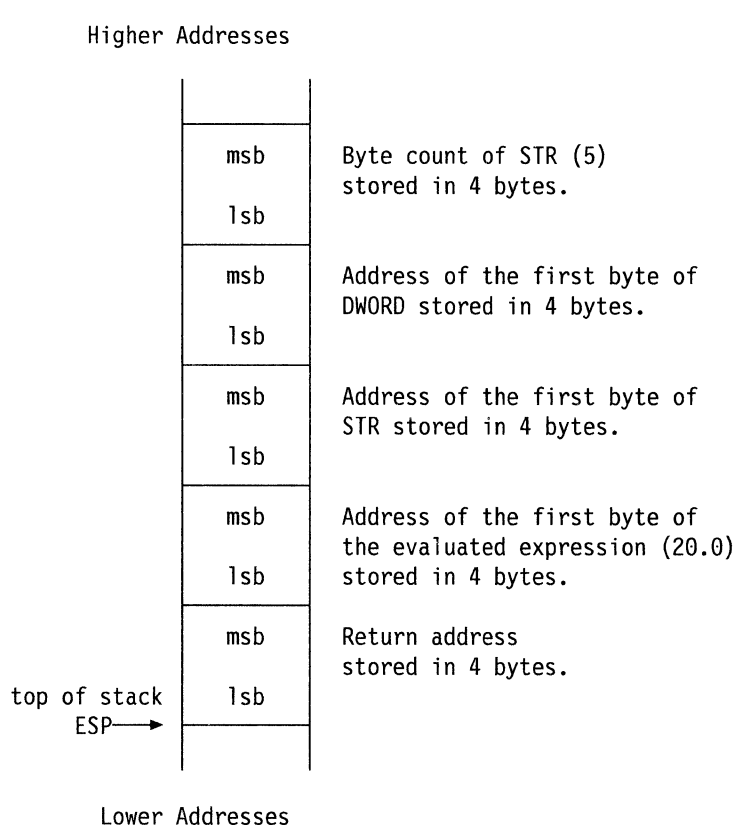


Figure 6-2. Stack After Call in Example A

Upon return, since it is the responsibility of the calling routine to pop the arguments off the stack, the stack pointer in this example should be pointing to the least significant byte of the 4-byte address of the temporary expression. The return address is popped off by the called routine along with the local space used by the routine. The calling routine cleans up the stack by removing the parameters after control is returned.

Although FORTRAN allows the names of subroutines and external functions to be passed as arguments in subroutine calls and function references, you should avoid doing this when interfacing FORTRAN with another language.

Pascal Parameter-Passing Conventions

Pascal uses either the call-by-value or the call-by-reference mechanism for passing parameters. The mechanism used depends on the data type specified for the parameter in the procedure or function being called. If the parameters are pointers to other structured data, addresses of structured data get pushed onto the stack. When this happens, the pointer function (@) combined with the call-by-value mechanism performs the same way as the call-by-reference mechanism.

Pascal defines five parameter types:

- variable
- value
- const
- function
- procedure.

The last two refer to the passing of function and procedure names as parameters in a function or procedure call. They are not further described here since this chapter deals with the calling of routines in languages other than Pascal where the parameter type may not even be defined.

Variable Parameters

Variable parameters are specified by including **var** with the names of the formal parameters in the procedure or function definition statement. The actual parameters must also be declared in a **var** declaration statement by the time the procedure or function is called. The parameters are passed by the call-by-reference mechanism and the 4-byte address of the data structure is placed on the stack.

Value Parameters

Value parameters are specified by the absence of **var** with the names of the formal parameters in the procedure or function definition statement. When the procedure or function is called, the actual parameters can be an expression, a constant, or even a variable that was declared in a **var** declaration statement. A value parameter also falls into one of these three categories: set, double, and other.

For set value parameters, a calling routine always uses the call-by-value mechanism; the parameters are pushed onto the stack regardless of their length. For example, a set that occupies 1 byte of storage is pushed as a byte onto the stack. A set that occupies more than 1 byte of storage is pushed with the least significant element in the least significant word which is at the lowest absolute address. Thus, the representation of a set on the stack is the same as its representation in memory.

For double value parameters, a calling routine always uses the call-by-value mechanism; usually 8 bytes are pushed onto the stack in such a way that the representation of a double value on the stack is the same as in memory, that is, with the sign bit in the highest address byte.

A Pascal value parameter longer than 4 bytes is passed by its address. This last case is actually a call-by-reference, but Pascal requires the called procedure or function to copy the data into its local storage. This is done so that the data associated with the parameters in the calling routine remain unchanged when the data are manipulated by the called routine, as is true with the normal call-by-value mechanism.

Arrays that are referenced in a procedure or function call are considered value parameters longer than 4 bytes. However, whether a value or an address is placed on the stack depends upon the procedure or function declaration statement. If the declaration statement specifies the parameter as an array, the address of the first byte of the array is placed on the stack in 4 bytes, and the called routine has to copy the entire array into its local storage. If the declaration statement does not specify the parameter as an array, the value of the element is placed on the stack.

Example B:

```
program PASS;
    type
       ARAY = array [1. . 20] of Real;
    var
       A : ARAY;

procedure
    APAS(var X : Real); external;
procedure
    BPAS(var X : ARAY); external;
procedure
    CPAS(    X : Real); external;
procedure
    DPAS(    X : ARAY); external;

begin
       A[2] := 2.;
       APAS(A[2]);
       BPAS(A[2]);
       CPAS(A[2]);
       DPAS(A[2]);
end.
```

These procedures have been made external to simplify the example; declaring the procedures external tells the compiler that they exist in a different file.

In Example B, the calls to APAS and BPAS cause the 4-byte address of the second element of the array A to be pushed onto the stack because the single parameter to these procedures is a variable, which is declared in the procedure declaration statement. The call to CPAS places the 4-byte value of A[2] on the stack because the procedure declaration statement specifies a non-variable real number.

The call to DPAS in the example causes the 4-byte address of the second element of the array A to be pushed onto the stack. This occurs because the procedure declaration statement identifies the parameter of DPAS as the first element of an array that is to be passed by the call-by-value mechanism. The address of the start of the array is passed, and the procedure copies the array into its local storage area.

Const Parameters

Const parameters are specified by including **const** with the names of the formal parameters in the procedure or function definition statement. Any expression, variable, or constant may be passed by **const**. No attempt should be made to modify the actual parameter while it is being passed by **const**.

General Pascal Considerations

In addition to the five parameter types, Pascal also defines hidden parameters. They are not user specified; they exist for compiler implementation reasons only. Hidden parameters are defined in the following cases:

- A parameter is inserted after each instance of a conformant string in the formal argument list. It has a value equal to the corresponding actual argument's declared string length.
- An extra parameter appears as the last argument in the parameter list of functions which return structures or strings. It holds the address of the function result.

- A parameter is inserted after each instance of a file type variable. This hidden parameter contains file naming information which is required by the VS Pascal run-time library routines.

As indicated previously, either the values or the addresses are pushed onto the stack by the Pascal compiler in the reverse order in which they are declared in the procedure or function declaration. After the parameter information is pushed onto the stack, the 4-byte **return** address is pushed and the stack pointer points to that address. Then, if the procedure or function is global, the value nil (0) is pushed onto the stack as the final item before the routine is called. If the called routine is not a global procedure or function, a 4-byte address known as the ***static link*** is pushed onto the stack as the final item. The static link is an address for the called routine that indicates the location of the variables in the routine which lexically encloses it.

Static links complicate the mixing of Pascal with other languages and can be avoided by declaring the called routines as external procedures or functions in the Pascal main program. This declaration, which makes the called routine globally defined, must be in the Pascal main program even if the other language routines are referenced only by a separately compiled Pascal procedure.

Function results are returned in register **EAX** or at the top of the Intel 80387 floating-point stack if the value is a floating-point number (single or double precision).

When the called procedure or function finishes its execution, the stack must be cleaned up. All parameters are discarded from the stack by the calling routine after control is returned.

Example C:

```
program SAMPLE;
    type
       STR = packed array [1. . 20] of Char;
    var
       A : Real;
       I : Integer;
       S : STR;
procedure
    FSUB(var ARG1 : Real;
         var STG : STR;
             input : Integer); external;

begin
       I:=5;
       A:=100.5;
       S:=' This is a string. ';
       FSUB(A,S,I);
end.
```

Figure 6-3 illustrates a stack after the call in Example C is made (msb and lsb indicating the most and least significant bytes, respectively):

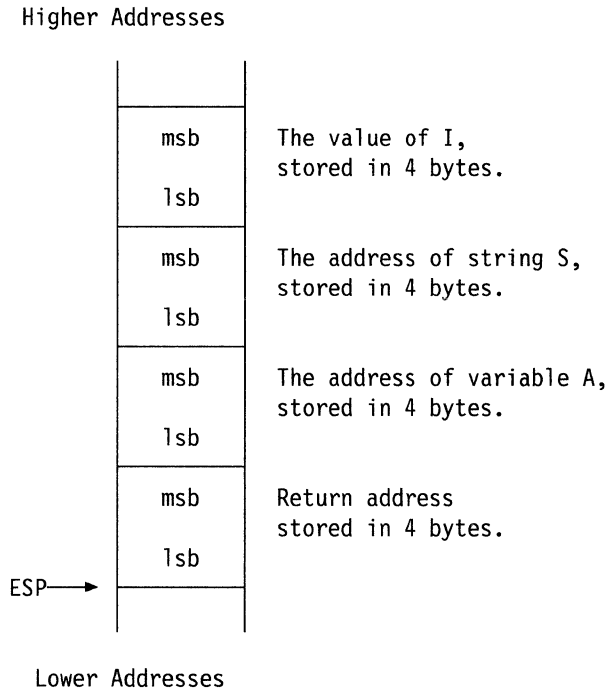


Figure 6-3. Stack After Call in Example C

Upon return, since it is the responsibility of the calling routine to pop the parameters off the stack, the stack pointer in this example should be pointing to the least significant byte of the address of A. The return address is popped off by the called routine. The calling routine cleans up the stack after control is returned.

C Argument-Passing Conventions

The C Language uses the call-by-reference mechanism for passing arguments declared as arrays and uses the call-by-value mechanism for passing other arguments. When you use the address of operator (&), these other arguments may be pointers to other structured data, in which case the addresses of the structured data get pushed onto the stack. This pointer function combined with the call-by-value mechanism performs the same way as the call-by-reference mechanism.

The compiler does not treat as arrays those arguments that lack indices. If an actual argument includes an index, it specifies an element of an array.

Example D:

```
main ()
{
  int x[5];
  x[0] = 100;
  func1(x);
  func2(x[0]);
}
```

In Example D, the call to func1 causes the 4-byte address of the first byte of the first element of array x to be pushed onto the stack. The call to func2, however, causes 4-byte value of the first element of array x (the number 100) to be pushed onto the stack. An actual argument that is an expression is evaluated before the function reference and its value is pushed onto the stack. All arguments except **struct**, **union**, **double**, and **float** arguments are pushed onto the stack as 4-byte values; **double** and **float** arguments are pushed as 8-byte values; actual arguments are pushed onto the stack in the reverse order that they are declared in the function declaration. The last item pushed onto the stack is always a 4-byte return address in the calling routine.

Functions in C Language need not return results; when they do, the non-floating point results are returned in register **EAX**. Floating-point values are returned at the top of the 80387 numeric processor stack. Upon return from a function, the arguments passed to the function are still on the stack, though their values may have been altered by the function. It is the calling routine's responsibility to pop the arguments off the stack.

Example E:

```
main ()
{
  int I;
  short J,K;
  char str[6];
  float x;
  double dx;

  I = 15;
  J = 42;
  K = 59;
  str[0] = 'H'; str[1] = 'e';
  str[2] = 'l';       str[3] = 'l';       str[4] = 'o';
  str[5] = '\0';
  x = 100.;
  dx = 99.e-2;

  xfunc(I,J,&K,str,x,dx);
}
```

Figure 6-4 on page 6-17 illustrates how the stack looks after the call in Example E is made (msb and lsb indicating the most and least significant bytes, respectively).

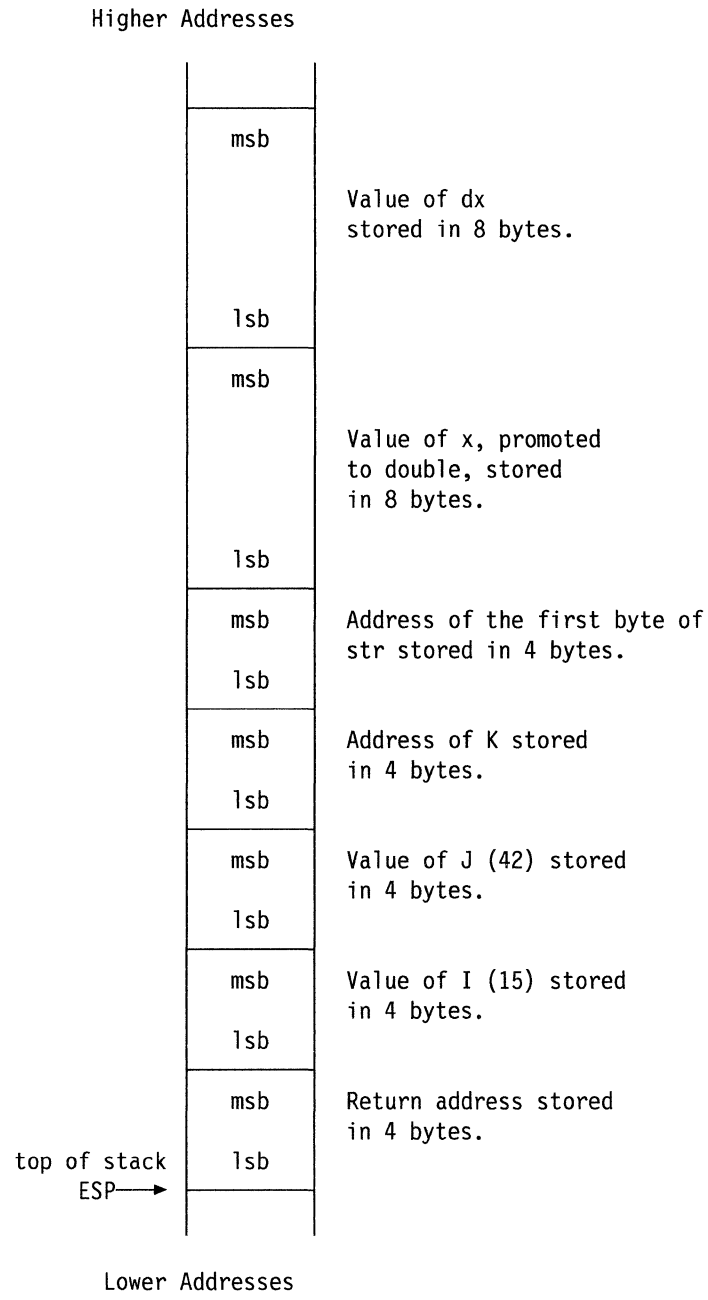


Figure 6-4. Stack After Call in Example E

Upon return, since it is the responsibility of the calling routine to pop the arguments off the stack, the stack pointer in this example should be pointing to the least significant byte of I. The **return** address is popped off by the called routine.

The function **xfunc** may return a value in register **EAX**, but in this example it is ignored. The function may also return a result by modifying the array *str* or in the variable *K*.

Assembler Routines Called By Other AIX PS/2 Languages

When writing 80386 Assembler language routines that are to be called by the other AIX PS/2 languages, do not interfere with the way the high-level languages use the stack, heap, and internal registers of the 80386. See Table 6-3 on page 6-7 for more information on linkage conventions.

In general, an Assembler routine can use the space on the stack below the address indicated by the stack pointer at call time. The stack grows toward smaller absolute addresses, so address space with smaller addresses than that in register **ESP** can be used for working memory.

Using VS Pascal Def/Ref Variables

In addition to passing data through parameter lists, VS Pascal **def** and **ref** variables may be used to communicate between Pascal and FORTRAN or C. The **def** or **ref** variable is associated with a FORTRAN common block name, or a C **extern** variable name.

Note: Only the first eight characters in the **def/ref** name are significant.

For communication between Pascal and FORTRAN, the name of the common block must be exactly the same as the **def/ref** variable name. For communication between Pascal and C, the name of the C **extern** variable must be the same as the **def/ref** variable with an underscore (_) both prepended and appended to it.

For example:

Pascal definition

```
def abc: integer;
```

FORTRAN declaration

```
COMMON /ABC/ A
```

C declaration

```
extern int _abc_;
```

Chapter 7. Mixing Languages and Linkage Convention on RT PC

CONTENTS

About This Chapter

This chapter describes the conditions, rules, and conventions that the user or programmer must observe when mixing program elements of two or more AIX RT PC high-level languages. Detailed information is provided for the VS FORTRAN, VS Pascal and C Compilers on the RT PC.

Mixing Languages on RT PC

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 want to mix; the elements of the languages are not described here in detail. Appendix C, "Program Examples for Mixing Languages" on page C-1 gives sample programs that show how to pass parameters between different languages.

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[1] standard data representations are used as much as possible. Table 7-1 shows some of the correspondence among languages.

Note: Table 7-1 shows how the languages represent data internally in the computer's memory rather than how data are passed between program units.

Table 7-1. Correspondences of Data Types Among Languages

FORTRAN IBM and R1 Modes	**FORTRAN VX Mode**	**Pascal**	**C**
Logical*1	Logical*1	Boolean	
	Logical*2		
Logical, Logical*4	Logical Logical*4		
Integer*1	Integer*1 Byte	packed/unpacked CHAR element	
Integer*2	Integer*2		short
Integer*4	Integer*4	Integer	int
Real, Real*4	Real, Real*4	Shortreal	float
Real*8	Real*8	Real	double
Complex	Double Precision Complex		
Complex*8	Complex*8		
Complex*16	Complex*16		
Character	Character	packed array of Char	char
		STRING	

[1] IEEE 754 Floating-Point Standard.

As Table 7-1 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.

Character Variables

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. The data type **packed array of Char**, however, like the **Character** type in FORTRAN, 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) A(3,2)
```

Pascal and C matrices are stored in computer memory by row (row major order). For example, an array in Pascal declared as:

```
A : array [1..3,1..2] of Real
```

is stored in this order:

```
A[1,1] A[1,2]      A[2,1] A[2,2]      A[3,1] 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.

When the main program is not compiled using the RT PC C compiler, special handling is required when calling RT PC VS Pascal and RT VS FORTRAN subroutines that perform input/output or floating-point arithmetic.

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

Note: When using RT PC FORTRAN languages, a trailing underscore is automatically appended to the routine name; therefore the duplicate routines **vspio_** and **vsfio_** are included in the libraries. When calling VS FORTRAN from a non-RT VS Main program, you should include a call to **vsfio_** as standard practice.

To properly flush input/output buffers before program termination from a main program that has called a FORTRAN subroutine, the routine **vstfio_** must be called immediately before exiting your main program.

Whenever you mix languages, you need first to become familiar with the requirements of each particular subroutine and each language's argument-passing requirements.

Subroutine Linkage Convention on RT PC

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). Table 7-2 lists registers and their functions.

Table 7-2. Register Usage

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.
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.

Stack Frame

When a routine is called, the compiler passes parameter words 5 through *n* on 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.

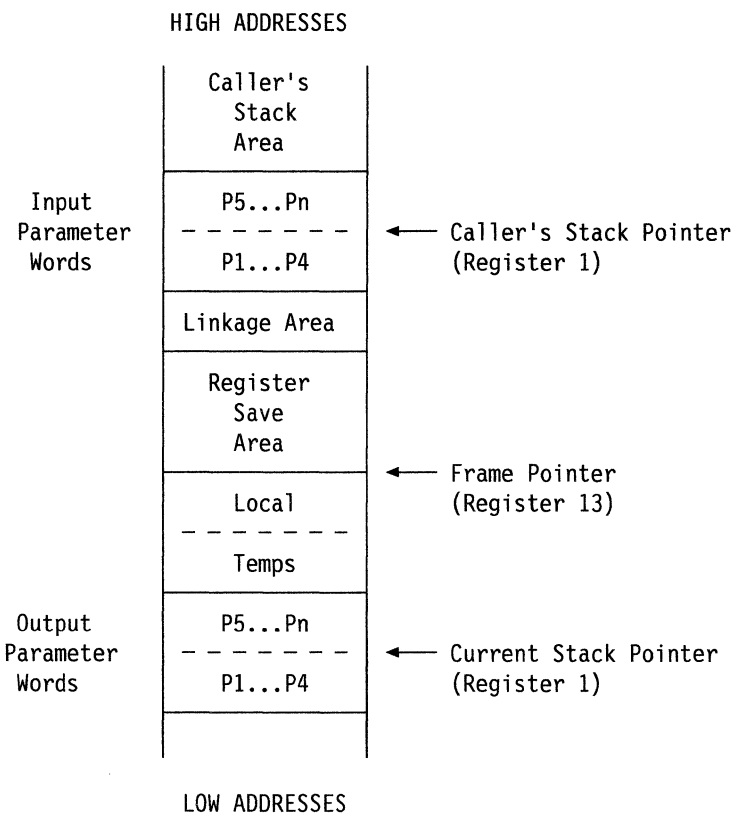


Figure 7-1. Contents of a Stack Frame

Figure 7-1 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, upon entry the stack pointer (register 1) 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.
- In RT PC VS Pascal and FORTRAN, a parameter that is a procedure or function is passed as a pointer to the routine's constant pool. In addition, for RT PC VS Pascal, the routine's environment pointer is passed in the next successive word.
- 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.

Using VS Pascal Def/Ref Variables

In addition to passing data through parameter lists, VS Pascal **def** and **ref** variables may be used to communicate between Pascal and FORTRAN or C. The **def** or **ref** variable is associated with a FORTRAN **common** block name, or a C **extern** variable name.

Note: Only the first eight characters in the **def/ref** name are significant.

For communication between Pascal and FORTRAN, the name of the **def/ref** variable is the same as the **common** block but with an underscore appended to it. For communication between Pascal and C, the name of the **def/ref** variable must have an underscore appended to it. The C **extern** name must be the same as the **def/ref** name, including having the underscore (_) appended to it.

For example:

Pascal definition

```
def abc_: integer
```

FORTRAN declaration

```
COMMON /ABC/ A
```

C declaration

```
extern int abc_;
```

Chapter 8. AIX/370 Linkage Conventions

CONTENTS

About This Chapter

This chapter specifies AIX/370 conventions for inter-procedural calls. Specifically, these include:

- Stack frame layout
- Register usage during call, prolog, and epilogue
- Argument and result passing
- Trace table definition.

Also discussed are object code attributes required to support debuggers and other routines which perform trace-back.

The first section, "Linkage Specifications" on page 8-2, provides the formal definition. Subsequent sections give examples of usage. Debugger considerations (for example, trace back) are discussed in "Trace Back Implications" on page 8-11.

Linkage Specifications

Stack-based

- R13 always identifies the beginning (lowest address) of the current stack frame.
- Negative growing – AIX/370 establishes a write/protected *red-zone* page immediately below the lowest address allocated for stack usage. In the typical linkage,[1] this avoids explicit stack overflow checking in each procedure prolog. Instead, the kernel can automatically expand the stack size when an attempt to store is detected for an address below R13 and above the lowest address currently allocated to the stack.
- 88-byte, doubleword-aligned stack frame header (see "Stack Frame Header" on page 8-3 for a definition).
- 4(R13) is used to store (and hence identify) the address of the calling procedure's stack frame. Frameless routine detection is discussed in "Trace Back Implications" on page 8-11.

Call Linkage

- R14 defines the return address.
- R15 defines the called routine's entry point address.

Proper handling of these registers in procedure prolog/epilogue sequences is critical for trace back operation. See "Trace Back Implications" on page 8-11 for details.

Register Volatility

- R2 – R12 must be saved and restored if modified by a program.
- R0 and the FPRs are scratch unless required for a return value.
- R1 is always scratch.
- Restoration of R13 is implicit in its role as the stack pointer.
- R14 and R15 must be preserved in accordance with trace back requirements in "Trace Back Implications" on page 8-11.

Argument Passing

The argument list occupies successive fullwords in the caller's stack frame immediately following the caller's register save area for R15.

- The first four words are passed in registers R0-R3 and thus an initial STM such as: STM R4, R3, 24 (R13) could be used to save these values, if required, in their respective reserved locations in the argument list.
- An argument of less than four bytes is right-aligned in its word. An argument of more than 4 bytes (for example, a structure) is left-aligned in multiple words.
- Procedures which return a **structure** or **union** are passed a pointer to the return area as a hidden first argument in R0.

[1] Where the stack frame size is ≤ bytes.

Return Value

- Simple values are returned via R0 (integer, character, pointer) or FPR0-6[2] for floating point values.
- **Structure**s and **union**s are moved by the called procedure to the area specified by the hidden first parameter.

Trace Back

Trace back methodology is discussed in "Trace Back Implications" on page 8-11. Each entry point is preceded by a trace table in one of the formats described in "Trace Table" on page 8-4.

Special Considerations

These are special situations which a compiler may choose to be aware of in support of optimization or support of run-time services.

Frameless routines are any routines that do not call another procedure, have a limited requirement for local storage, and do not acquire a stack frame.

Standard Service Routines

Several service routines are required in support of this linkage specification. While the service routines themselves are not a proper part of this specification, certain aspects of their behavior are assumed in order to assure efficient linkages. Additionally, the existence of a **StackCheck** routine is dictated by this specification. See "Service Routines" on page 8-10 for additional information of **StackCheck, alloca()** and **mcount**.

Stack Frame Header

The following structure defines the fields in the stack frame header. Field names are used illustratively and are not intended to be a standard nor is anything implied by conflict between the names here and other names within the AIX/370 system's source files.

```
struct stackframe {
   unsigned long sf_lang_word;          /* A language specific word
                                           reserved for usage such as
                                           the Pascal static link */
   struct stackframe *sf_of_caller;     /* back chain to caller's
                                           stack frame           */
   unsigned long sf_reserved[2];        /*reserved               */
   unsigned long sf_save_2_15[14]       /*R2-R15 save area       */
```

---

2 The number of FPRs (Floating-Point Registers) required to contain the return value will be utilized.

```
        /*   Argument list begins here -- the first two words
             coincide with R0-R3 save areas.
  unsigned long sf_r0_p1;          /* R0 save area / Parameter word 1 */
  unsigned long sf_r1_p2;          /* R1 save area / Parameter word 2 */
  unsigned long sf_r2_p3;          /*   Reserved for Parameter word 3 */
  unsigned long sf_r3_p4;          /*   Reserved for Parameter word 4 */

  unsigned long sf_parm_5;         /* Parameter word 5               */

  ... additional parameters follow here

  }
```

Note: Additional parameters words beyond the first four are passed to the called procedure in memory immediately following *sf_r3_p4*.

Trace Table

Each procedure has a trace table immediately before its entry point. Trace tables are utilized by debuggers to assist with failure analysis. They provide such information about the number of inbound parameters, the maximum number of outbound parameters, assignment of arguments to registers or storage during execution of the procedure, and the size of the code associated with the procedure. Trace table content is extensible in that a format field identifies the particular trace table structure. *C* structures follow describing three currently defined trace table formats.

Format 1: This is the standard trace table format providing the information required by the debuggers for normal routines.

```
struct tracetable_f1  {          /*  Format 1 trace table       */
    unsigned short t_arglength;  /* WORDs of memory required for the
                                    longest outbound parameter list
                                    beyond the minimum four allowed
                                    for in the register save are. */
    unsigned short t_arg1:4,     /*  If a sub-field # 15, then the
                   t_arg2:4,        corresponding parameter is saved
                   t_arg3:4,        in the memory argument list area,
                   t_arg4:4;        otherwise it defines the register
                                    allocated to the argument within
                                    this procedure.   */
    unsigned long  t_codesize;   /* size of procedure code  */
    unsigned short t_flags;      /* See text notes  */
    unsigned char  t_parmlength; /* Number of arguments words declared
                                    by this procedure including the
                                    initial 1-4 passed in registers */
    unsigned char  t_format;     /*  =1 ... a format 1 trace table  */
};
```

Format 2: This trace table format provides the absolute minimum and may only be used for frameless routines.

```
struct tracetable_f2  {          /*  Format 2 trace table       */
    unsigned char  t_parmlength;
    unsigned char  t_format;     /*  =2 ... a format 2 trace table */
  };
```

Format 3: This alternative format provides for procedures which require an undetermined amount of arbitrary data associated with an entry point. The *tagged data* consists of a sequence of arbitrary self-describing entries.

```
struct tracetable_f3  {           /*  Format 3 trace table       */
   struct tagdata *t_tagdata;    /* Address of the tagged data area.*/
    unsigned short t_arglength;
    unsigned short t_arg1:4,
                   t_arg2:4,
                   t_arg3:4,
                   t_arg4:4;
    unsigned long  t_codesize;    /* size of procedure code   */
    unsigned short t_flags;
    unsigned char  t_parmlength;
    unsigned char  t_format;      /*  =3 ... a format 3 trace table */
  };
```

The **tagdata** structure is described under "Tagged Data" on page 8-13.

Format 0: This trace table format is obsolete but may exist in binary files.

```
struct tracetable_f0  {           /*  Format 0 trace table    */
    unsigned short t_arglength;
    unsigned short t_arg1:4,
                   t_arg2:4,
                   t_arg3:4,
                   t_arg4:4;
    unsigned char  t_parmlength;
    unsigned char  t_format;      /*  =0 ... a format 0 trace table */
  };
```

Notes:

1. The t_format field is immediately followed by the entry point.
2. If a trace table requires padding for alignment, the padding should precede the trace table.
3. The *root* (actually the end, memory address wise) section of the trace table is common to all formats.
4. *t_flags* is reserved to store trace table modifiers and/or additional information about the annotated entry point.

Summary of Linkage Characteristics

- Call: 2 instructions (6 bytes plus adcon)
- Prologue: 54 instructions (16 bytes plus constant) typically
- Epilogue: 2 instructions (6 bytes)
- No limitation on size of program, stack frame, or argument list

- Parameter passing: first four words passed in registers
- (Code base register usage is not in the domain of linkage conventions.)
- *Varargs* supported
- **alloca()** supported
- In a typical stack frame of $\leq$ bytes, a single stack pointer can address outgoing arguments, local variables, the register save area, and incoming arguments.
- The stack frame header provides a back chain pointer for locating the previous stack frame as well as a register save area, argument storage, etc.
- A trace table before entry point gives information for debuggers
- Linkage designed for use by C, Pascal, and FORTRAN
- Frameless routines may utilize as scratch storage any word in the basic stack frame header (other than the back chain pointer) that is not required for preserving the caller's registers.

AIX/370 Linkage
Register Conventions

Reg	Life	Use
0	Killed	Parm 1 and simple return value
1	Killed	Parm 2
2	Saved	Parm 3
3	Saved	Parm 4
4	Saved	
5	Saved	
6	Saved	
7	Saved	
8	Saved	
9	Saved	
10	Saved	
11	Saved	
12	Saved	
13	Saved	Stack Pointer
14	Saved	Ret: Return Address
15	Killed	Sbr: Called Procedure Entry Point
f0	Killed	Float or double Return Value
f2	Killed	(return value if complex or extended precision)
f4	Killed	(return value if complex extended precision)
f6	Killed	(return value if complex extended precision)

Explanatory notes

- Simple results (**char, int, pointer**) are returned in R0.
- Floating results are returned in f0 (and f2-f6 if required).
- Program base register allocation[3] is a responsibility of each compiler and is therefore not included in this specification.
- The stack pointer points to the bottom of the stack.

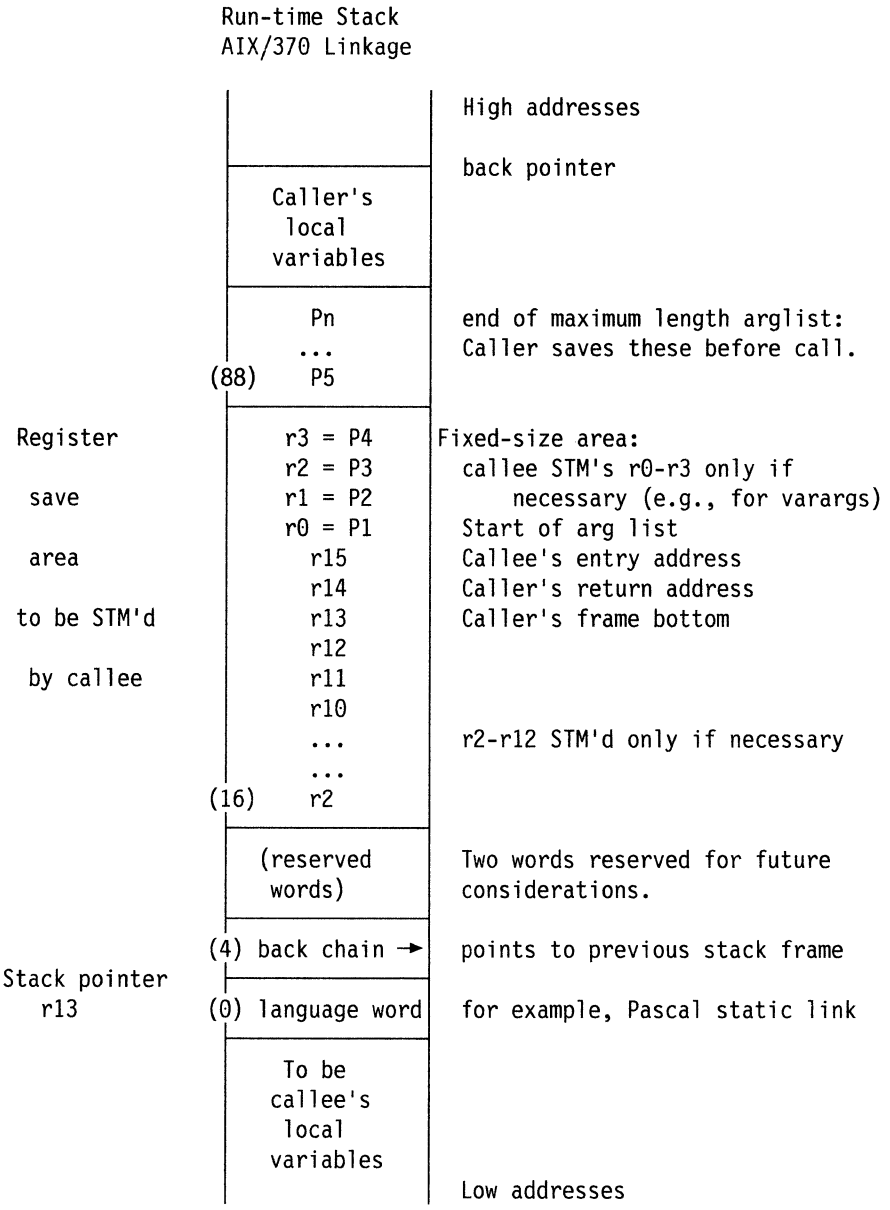
[3] Note that code chunking can be exploited to avoid the need for multiple base registers for large program code.

Linkage Examples

This section is intended to clarify the linkage conventions by illustrating how a compiler or assemble language programmer might code the various epilogue/prolog and call linkage sequences required by the convention.

Registers shown by register number (for example, R13) are specific registers required by the linkage convention. Registers required to accomplish a particular code fragment will be shown as a non-numeric register (for example, Ra, Rb, Rtemp).

This figure shows the stack just before execution of a called procedure's prolog.



Calling Procedure's Call Code

The caller passes the first four argument-list words in R0-R3. For longer argument lists, the caller stores words 5-n in sequential word-aligned stack locations, beginning at 88(R13). A *C* function returning a **struct** or **union** is passed (as a hidden first argument) the address of the area to store the result.

The caller of a nested Pascal procedure stores a static link (the frame pointer to the callee's parent) at 0(R13).

Once the parameters are in place, the caller invokes the callee with the following instructions.

```
L     r15,=A(callee)
BALR  r14,r15
```

Called Procedure's Prologue Code

The entry point is preceded by a trace table holding information for debuggers and execution profiling. The trace table format is described in "Trace Table" on page 8-4.

The first instructions store any registers that must be saved (for example, the return address, the old stack pointer, and any of the R2-R12 that are needed for scratch registers). A local code base is established (if required), the stack pointer is adjusted, and the back chain stored.

Small Size (≤4K) Frame

Callee:

```
STM   Ra,Rz(a-z)*4+16(R13)     The Ra-Rz range utilized will be a
                               compiler optimization issue.  R14
                               and R15 must be saved.
LR    Rtemp,R13                Copy the in-bound stack pointer.
                               Rtemp would typically be the local
                               base chosen by the compiler if
                               required.
LR    Rcode,R15                The code base register is set
SL    R13,=F'framesize'        Decrement the stack pointer
ST    Rtemp,4(R13)             Establish the back chain
```

Note: *framesize* must be a multiple of 8.

The following example illustrates an alternative prolog which makes potentially useful trade-offs in register usage, instruction counts, and memory references.

```
STM   Ra,Rz,(a=2)*4+16(R13)
LR    Rtemp,R13                Copy old stack frame
LA    Rcode,framesize          Trade LA and SLR for SL with
SLR   R13,Rcode                memory reference constant
ST    Rtemp,4(R13)
LR    Rcode,R15
```

Large Size (>4K) Frame

The above code assumes a stack frame smaller than the *red-zone* size. If *framesize* is larger than 4096 bytes, the following prolog might be used. A minimal (88 byte) stack frame is established and then **StackCheck** is called to verify that sufficient memory exists to hold the required stack frame. **StackCheck** returns with R13 properly adjusted to reflect the full size stack frame.

Callee:

```
STM   Ra,Rz,(a=2)*4+16(R13)    (See comments above re. Ra, Rz)
LR    Rtemp,R13                Save the in-bound stack pointer
LA    Rcode,88                 Establish a minimum stack frame
SLR   R13,Rcode                ...
ST    Rtemp,4(R13)             minimum stack frame completed
LR    Rcode,R15                Set code base register
L     R15,=A(stackCheck)       To the checker routine
BALR  R14,R15
DC    AL4(framesize=88)        size of desired frame expansion
```

The choice of registers for addressing locations in the stack frame is a compiler decision. R13 always addresses the start of the stack frame; another register may be needed to address the end of the frame if the frame is large or the value of R13 will be changed by calls to **alloc()**.

Execution Profiling

When profiling is enabled, the following code segment is added to the end of the prolog. *mcountword* is the address of a fullword allocated by the compiler in the data segment. Each procedure profiled has a unique *mcountword.*

```
L     R15,=A(mcount)           profiling routine
BALR  R14,R15
DC    AL4(mcountword)
```

The **mcount** procedure must be written to preserve *all* registers.

vararg Support

If the called procedure takes the address of any argument name, the compiler must ensure that argument list words 1, 2, 3, and 4 reside in memory in their proper positions.[4]

Epilogue Code

An integer result is returned in R0; a floating result in F0(=F6 if required). A multi-word result (for example, a C struct or union) is returned by having the callee move it to the caller's storage. The caller passed the address of this area as a hidden first argument. The epilogue code restores any modified registers in the range R2-R14.

The first example illustrates the typical epilogue sequence applied when the stack frame is $\leq$ 4K bytes in size. A similar optimization is possible if the compiler has allocated a register which would provide basing for the inbound stack frame header.

```
LM    Ra,R14,(a=2)*4+16+framesize(R13)
BR    R14
```

---

[4] The *varargs* macros declare 4 arguments so that all four parameter registers are saved.

The next example illustrates the general case where basing for the caller's stack frame must be established by the epilogue:

```
L     Rtemp,4(R13)              Use of Rtemp avoids an
                                ambiguous state for R13
LM    Ra,R14,(a=2)*4+16(Rtemp)  R15 is restored to insure that
                                frameless routine ambiguity
                                resolution logic will work
BR    R14
```

Service Routines

These routines are not a proper part of a linkage specification, but the functions provided are required. The approaches described here for implementation of these functions serve to illustrate the limitations and requirements for inclusion of similar function. A compiler is likely to desire special knowledge of the existence of these routines or actually require the routine for mandatory services.

These routines all receive a compiler furnished special parameter coded as a full word unaligned constant following the *BALr* instruction.

StackCheck A routine utilized to allocate stack frames which exceed 4K bytes.

Prior to calling **StackCheck**, a standard minimum stack frame (88 bytes) must be allocated.

The special parameter passed following the *BALr* instruction will be the required *increase* in stack frame size.

The requested size is rounded up to an integral number of doublewords and checked to insure that the resulting stack frame pointer will remain within bounds.

The new stack frame header is established and initialized as follows:

1. The 4(r13) value inbound is loaded into **Rtemp**.
2. R13 is set to its new value.
3. ST Rtemp,4(R13) establishes the back-chain, language word and reserved words. **StackCheck** now appears as frameless routine without a stack frame.
4. Control transfers back to the caller.

alloca A special memory allocate interface for expansion of the stack frame.[5] **alloca** allocates space within the stack frame by decreasing R13, checking for stack overflow, and returning a pointer to the area immediately following the outgoing argument list. Processing is similar to **StackCheck** except:

1. The special calling sequence parameter specifies the size of the actual stack frame header and maximum outbound argument list utilized by the calling routine.
2. The size request is passed as parameter 1 (R0).
3. The *allocated* memory address is the return value (in R0).

[5] Programs which utilize this routine may not depend upon R13 for access to local variables.

mcount Used by a compiler to generate linkages to a standard routine for collection of execution profile information. The special parameter will contain the address of a fullword allocated and initialized to zero by the compiler in the *data* section of the program.

Debugger Considerations

Trace Back Implications

Assumptions

1. A frameless procedure does not establish its own stack frame.
2. Each entry point has an associated trace table.
3. An ambiguous state procedure is any procedure which currently does not own a stack frame (for example., during prolog/epilogue processing).
4. R14 and R15 must be saved by non-frameless procedures.
5. R14 reflects the procedure return value for all frameless and ambiguous state procedures.
6. The two instruction sequence required to *push*[6] the stack are adjacent.
7. If a frameless procedure saves R14, it also stores R15.

Methodology

This specification includes the requirements necessary to provide procedure call trace back. This chapter discusses how those requirements might be translated into accurate trace back information, even in those narrow windows between the two instructions required to *push* the stack or *pop* and return from a procedure.

- The stack *push* operation is interpreted as an atomic operation by the debugger. If the next instruction to be executed is a ST Rtemp,4(R13) instruction then the trace back routine will use the actual contents of *Rtemp*[7] rather than the value at 4(R13) as the *back chain pointer* required to establish the address of the calling procedure's stack frame.
- The interval in each routine's prolog between the STM Ra,Rb,xxx(R13) and the ST Rtemp,4(,R13) instruction which establishes the calling procedure's stack frame pointer as well as any frameless routines which do not allocate a stack frame present an ambiguous situation. That ambiguity may be resolved by which ever of the following techniques is applicable:

  1. Object file includes the symbol table:

     The symbol table is used to relate the *PSW* address to the routine in control.

---

[6] R13 must be decremented by the size of the stack frame and the previous stack frame pointer must be stored in the new stack frame header.

[7] Violation of the requirement that the decrement of R13 be immediately followed by establishment of the back chain will almost certainly result in a faulty back trace in the event of an asynchronous interrupt before the back chain is valid.

2. Object file does not include the symbol table:

   a. The *back chain pointer* is used to identify the previous stack frame entry's R15 (and hence entry point of the called procedure associated with that stack frame entry).

   b. The previous trace table is located using the previous R15.

   c. If the current *PSW* is *not* between the previous R15 and the end of that procedure's code as determined from the trace table *and* the current R14 is within that procedure's code space, then it is assumed that a procedure which has not allocated or will not allocate a stack frame is currently in control:

      - Any registers saved by the current routine may be found in the stack frame header identified by R13.
      - An attempt will be made to locate the actual inbound registers associated with a frameless routine as follows:

        1) If the next instruction is an STM from R13 at the proper stack frame header locations for the register being stored and R15 also contains the next instruction address, then a debugger will assume the current register contents are the inbound values.

        2) If the value of R14 matches the appropriate register save area location, then the R15 save area value will be used. If an STM exists at that location, it will be assumed to define the set of registers stored for this procedure.

        3) If the current contents of R15 point to an appropriately structured STM off of R13, then that STM will be assumed to define the inbound register save set.

      If this process is not successful, then the debugger will report the contents of the R13 identified stack frame register save area.

   d. Otherwise, the save area in the current stack frame header will be reported as inactive as it has been determined that the ambiguous situation does not exist and the *back chain pointer* properly identifies the parent of the executing procedure.

All procedures provided with AIX/370 are expected to conform to these specifications. A nonconforming routine may result in an ambiguous or invalid trace back.

Tagged Data

Tagged data provides for inclusion of arbitrary information associated with an entry point which is guaranteed to be memory resident. The self-descriptive format provides for future extensibility.

```
struct tagdata {
     unsigned char  td_typecode; /* type code
                                        0x00 = End marker
                                        0x01 = Entry point name
                                        0x02 = Copyright
                                        0x03 = Program version - includes
                                               the following possible
                                               tagged sub-fields:
                                            0x00 (optional end)
                                            0x01 - Product ID
                                            0x02 - Source date/time
                                            0x03 - Structured Version
                                            0x04 - unstructured version.
                                        0xFF = Escape to alternate format
                                               allowing longer code and
                                               length fields.
     unsigned char  td_length;   /* bytes of data in tagged data */
     unsigned char  td_data[1];  /* data */
};
```

Chapter 9. Program Examples

CONTENTS

About This Chapter

This chapter provides sample programs that can be compiled with the C compiler.

Prime Example

This example determines the prime numbers in the range from 1 to 8190.

```
#define TRUE  1
#define FALSE 0
#define SIZE  8190

char flags[SIZE];

main()
/*
    Count the primes up to 8190 by marking off multiples
    of primes in succession
*/
{
    int i, k, count;

    count =0;

    /* 0 and 1 are not primes */
    for (i=2; i < SIZE; i++) flags[i] = TRUE;

    /* mark multiples of every prime starting with 2 */
    for (i=2; i < SIZE; i++)
    {
        /* is i a prime? */
        if(flags[i])
        {
            /* count i as a prime */
            count++;

            /* mark every multiple as note a prime */
            for(k=i; k<SIZE;k+=i) flags[k] = FALSE;

#ifdef PRINT
            printf("prime = %d\n",i);
#endif
        }
    }
    printf("\n%d primes\n",count);
}
```

AIX System Call Example

This example obtains information about a file. It demonstrates the use of AIX system call **stat**.

```
#include <sys/stat.h>


char *path;
struct stat buf;

main()
{
  printf ( "\n   This program uses the system call: stat\n");
  printf ( "     The call obtains information about a file.\n" );
  printf ( "     ----------------------------------------\n\n" );
  path = "/usr/bin/vi";
  printf ( "     Obtaining information on file: %s\n\n",path );
  if ( stat (path, &buf ) != 0 )
        printf ( "******** Error in stat\n" );
  else {
  printf ( "   ID of the device that contains a directory" );
  printf ( " entry for this file: %d\n",buf.st_dev );
  printf ( "   Inode number: %d\n",buf.st_ino );
  printf ( "   File mode: %d\n",buf.st_mode );
  printf ( "   Number of links: %d\n",buf.st_nlink );
  printf ( "   User ID of the file's owner: %d\n",buf.st_uid );
  printf ( "   Group ID of the file's group: %d\n",buf.st_gid );
  printf ( "   ID of device: %d\n",buf.st_rdev );
  printf ( "   File size in bytes: %d\n",buf.st_size );
  printf ( "   Time of last access: %d\n",buf.st_atime );
  printf ( "   Time of last data modification: " );
  printf ( "%d\n",buf.st_mtime );
  printf ( "   Time of last file status change: " );
  printf ( "%d\n\n",buf.st_ctime );
 }
}
```

Asm Statement Example for PS/2 and RT PC

```
/*
     This is an example of the asm statement on the PS/2 and RT PC.
     It is used to see if an overflow occurs when adding 2 positive
     numbers.
*/

#include <stdlib.h>
#include <stdio.h>

void    test_overflow(unsigned int, unsigned int);
int     check_flag(void);

main(int argc, char *argv[])
{

         unsigned int a,b;

         if (argc != 3)
                 printf("ERROR: 2 arguments expected\n");
         else {
                 a = (unsigned)atoi(argv[1]);
                 b = (unsigned)atoi(argv[2]);
                 test_overflow(a,b);
         }
         exit (0);
}

void test_overflow(unsigned int x, unsigned int y)
{
         int ret;

         x + y;                          /* add the 2 numbers      */
         ret = check_flag();             /* get carry flag         */
         if (ret & 0x100)                /* if carry flag is set   */
              printf("OVERFLOW\n");      /* then overflow occurred  */
         else                            /* else get result        */
              printf("x + y = %u\n", x+y);
}

int check_flag(void)
{
/* lahf instruction: the flags which indicate an overflow are in
                     register EAX, so there is no return statement.  */
        asm(0x9f);
}
```

Asm Statement Example for System/370

```
     _ASM ("     assembler instruction  \n");
```

Appendix A. Messages

The PS/2 C Compiler contains a file of compile-time error messages named **/usr/lib/msg/vscctmsg.inc.**. The compiler generates English error numbers and messages if this file is present on the **default** directory and errors are encountered.

The System/370 C Compiler generates the following error messages:

Error and Warning Messages

This section presents all compiler diagnostic messages (except automatically generated lexical and syntactic messages) in alphabetical order, with explanations where appropriate.

```
')' missing in macro call.

#define directive is malformed.

#else or #elif encountered within a pending #else.

#pragma statement is malformed.

'=' encountered where '==' may have been intended.
```

= was detected as an operator in a Boolean expression, such as `if (x = y) {...}`. Often this is a mistake, as `if (x == y) {...}` was intended.

```
Anonymous structure member is ambiguous.

Applying "offsetof" to a non-struct type.

Argument name is missing in function declaration.
```

For function definitions, argument names must be supplied. For example, `void f( int, float g) {...}` is illegal because the first argument lacks a name.

```
Argument name is not specified in function header.
```

For function definitions, arguments declared must be present in the function header. For example, **void** f(a,b) **int** x;..; {...} is illegal because x is not present in the function header.

```
Argument number n not named.
```

The nth argument in a named call is not specified. For example,

```
int f ( int a, int, int c ) {...}
...
f(a=>1,c=>2);
```

where the second argument is not specified.

```
Argument of type "void" not permitted.

Argument of unknown length being passed.

Argument type not consistent with previous specification.

Array dimension is negative.

Array element type is a yet undefined struct/union.

Array subscript out of range.

Assignment into a "const" variable is attempted.

Assignment into a variable of unknown length is attempted.

Assignment into something of type void is attempted.

Attempting to apply the function "abs" to an unsigned quantity.

Attempting to define a variable in data section name that was previously
defined at Ln/Cn.

Attempting to take the difference of two pointers pointing to objects of
unknown length.

"auto" must appear within a function. "static" assumed.
```

Storage class **auto** cannot be given for declarations that do not appear within a function.

```
Bad line specifier in #line directive.

Bit field exceeds word size.

Bit field is not an integer type.

Bit field is not valid as an argument to "sizeof".
```

Since bit fields need not occupy an integral number of bytes, taking their **sizeof** is prohibited.

```
Bit field value expected but encountered '{'.

"break" statement not within loop or switch.

Call to a function returning an incomplete or zero-length type
is attempted.
```

A function cannot **return** a **struct** of **union** type whose fields have not yet been specified. For example, `struct s; struct s *f() {...}` is legal since f returns a pointer to an incomplete **struct** type, but `struct s; struct s g() {...}` is illegal.

```
Call to non-function is attempted.

Cannot dereference a pointer to void.
```

Type **void*** was introduced as a means of defining a "generic pointer" compatible with other pointers. But there is no such thing as an object of type **void**. Therefore, dereferencing a pointer to **void** is illegal.

```
Cannot dereference pointer; size of object not known.

Cannot take the address of a register variable.

Cannot take the address of a struct bit-field.

"case" encountered outside of a "switch" statement.

Constant expression or static address is expected.

"continue" statement not within loop.

Data class specifier expected (EXPORT,COMMON,IMPORT).

Data section class not consistent with previous specification at Ln/Cn.

"default" encountered outside of a switch statement.

"default" label previously specified in current switch.

Dereferencing of a non-pointer is attempted.

Division by zero is attempted.
```

This was detected in a constant expression at compile time.

```
Duplicate case label (n); other occurrence at Ln/Cn.

Duplicate declaration of identifier.

Duplicate definition of structure member.

End of file encountered within #if construct (unmatched #if-#else-#elif a
t Ln/Cn).

enum tag is not defined.
```

A declaration such as **enum** x; was encountered. Tag x has no definition.

```
enum tag previously declared as struct tag.

enum tag previously declared as union tag.

Expression has no side-effects.
```

An expression used in a statement context has no side effect; therefore the expression is useless. For example, 2 + 3;.

```
Expression of type "type" should be returned.

Floating point constant expected.

Function called but not defined.
```

Any function that was called but not defined is noted as a warning. Although this practice is permissible in C and is especially useful when calling library functions, a common error is to misspell a function name. Without this warning, the error goes undetected until link-time. Further more, errors in parameter linkage can occur when a call is made to an undefined function. We recommend that the library .h header files always be included to get parameter checking, and that function prototypes be used for external function declarations, rather than making use of the "feature" of C for calling undefined functions.

```
Function declaration is inconsistent with previous declaration at Ln/Cn.

Function declaration is inconsistent with the "int"-returning function
declaration imputed at Ln/Cn.
```

A function called before it is declared is assumed to be a function returning **int**, and any subsequent declaration of the function must declare it to be so. For example, `main () { (...) f(3); (...) } void f() {...}` is illegal since f was called before being defined and therefore assumed to **return int**.

```
Function definition semantics overridden by prototype at Ln/Cn.
```

A prototyped function declaration takes precedence over an ordinary function definition. For example,

```
int f ( int a, int b);   /* Prototyped functionality. */
...
int f (a,b)
int a,b; {...}
```

The function declaration in the function definition is overridden by the prototype declaration. This warning may be suppressed by turning off the `Prototype_override_warnings` toggle.

```
Function has no return statement.
```

A function with a non-void return type contains no **return** statement. This typically happens with "old" C programs that did not use **void** to indicate that a function returns nothing.

```
Function name missing.

Function parameter names are allowed only on function definitions, not
declarations.
```

int f(a,b,c); is a function declaration that names the parameters (a,b,c). This is illegal unless function prototype syntax is used, as in `int f( int a, int b, int c);`.

```
Function specified with more arguments than its declaration allows.

Identifier expected.

Identifier is out of context.

Identifier is undeclared.

Identifier missing in declaration.

Identifier missing in struct member declaration.

Illegal construct within #if expression.
```

```
Initialization entry is of wrong type.

Initialization list is longer than aggregate being initialized.

Initialization of a variable within a "COMMON" or "IMPORT" section is not
permitted.

Initializer type does not match type "char".

Initializing a function is attempted.

Insufficient number of arguments in pragmas specification.

Insufficient number of arguments to function.

Insufficient number of arguments to macro.
```

The number of arguments to a macro must agree exactly with the number of parameters in its **#define**.

```
Insufficient number of arguments.

Integer constant exceeds largest unsigned number.

Integer constant expected.

Invalid use of "long" adjective.

Invalid use of "short" adjective.

Invalid use of type adjective.

Label was not defined.

Label was previously defined at Ln/Cn.

Left side of '=' is an array, which is not an lvalue.
```

In this context a so-called **lvalue** is required but was not found. An **lvalue** is something whose address can be taken, and is required on the left side of an assignment expression and as an operand to &, + +, and --.

```
 An array is not an example of an lvalue.

"long char" treated as "short int".

"long float" treated as "double".

Lower bound of range greater than upper bound.
```

This can only happen in C **case** statements where range expressions are allowed as labels (an extension).

```
Macro call has too many arguments.

Macro expansion nested too deeply.

Malformed conditional compilation control sequence.

May not cast a nested function to type "type" .

May not have functions as array elements (but may contain pointers thereto).

May not return a function from another function (but may return a pointer ther

May not return an array from a function (but may return a pointer thereto).

Member not defined in referenced structure.

Mismatched #if-#elif-#else-#endif.

Missing macro argument after or before "##" operator.

Missing macro argument after "#" operator.

Name is not a member in the applicable structure.

Name is undefined; "pragmas Alias" ignored.

Named bit-field of length 0 not permitted.
```

A declaration such as **struct** {**int** i:0, j:2 }; was encountered. It must be omitted. As is, it is possible to refer to the field. Such a reference would be illegal.

```
Nested function may not be assigned into a pointer to a function.

No "pragmas Data" is active.
   pragma Data; was encountered without a preceding, and matching, pragma
 Data(...);.

Not a variable or function; "pragmas Alias" ignored.

"offsetof" operation cannot be applied to bit field.

Operand must be of an integer or float type (instead of type).

Operand of ++ or -- is a pointer to an object of unknown length.

Operand of the function "abs" must be of arithmetic type.

Operand of type "type" is not valid for operator operator.

Operands of type "type1" and/or "type2" are not valid for operator operator.

Packed structs are not supported; "packed" ignored.

Packed structs are not supported; status of ALIGN_MEMBERS toggle ignored.


Parameter not found or specified more than once.
```

In a function call using named parameter association, a parameter was named twice, or a non-existent parameter was referenced.

```
Parameter "name" not supplied.
```

Named parameter not specified. For example,

```
void P ( int a, int b, float c ) {...}
...
P (a=>1, c=> 3);
```

does not initialize the named parameter b.

```
Passing an argument of type "type1" where "type2" is expected.
```

An attempt was made to pass an argument of a wrong type to a function such as passing a **float** for a parameter that is a **struct**.

```
Pointer to an object of unknown length is being indexed.
```

Attempting to index a pointer to an object of unknown length. `struct x *p; *p[0]` is an example of this type of error.

```
"pragma Code" specified within function definition; ignored.
```

```
"pragma Data" active at end of module.
```

A pragma Data (...); was given in a module or function, with no terminating pragma Data;. This is permitted but the programmer may have forgot ten to supply the terminating pragma, thus perhaps including more data declarations in a data segment than intended.

```
Previous "pragma Data" is still active.
```

pragma Data (...); was given in the context of an already active pragma Data (...). Insert pragma Data(); preceding the offending pragma to "turn off" the active pragma.

```
Previous definition of macro superseded.
```

Redefinition of a macro is permitted. The redefined macro takes precedence over existing definition of the macro.

```
Prototype causes non-standard conversion from "type1" to "type2".
```

The prototype syntax, causes the non-standard conversion from `type1 to type2`. This is signaled as a warning to indicate to the user that such a conversion is taking place.

```
Real constant has too many digits.
```

```
Redundant class specification.
```

```
"register" must appear within a function. "static" assumed.
```

Storage class **register** cannot be given for declarations that do not appear within a function definition.

```
Right operand of shift operator exceeds word size (=n).

Right operand of shift operator is negative.

Right operand of "%=" operator is zero.

Section name not specified in "pragma Data"; pragma ignored.

Selection of a member from a non-structure is attempted.

"short char" is not supported.

sizeof being applied to "void" type.

sizeof being applied to a function type.

sizeof being applied to an array of unknown length.

sizeof being applied to incomplete struct/union.
```

The **sizeof** a **struct** or **union** type whose fields have not yet been specified is not known. For example, `struct s; (...) sizeof ( struct s ) (...)` is illegal because the size of the structure is unknown.

```
Specifier "class" ignored; no variable being declared.
```

In a declaration such as `static struct s {int x;};`, the storage class static is useless since no object was declared.

```
Specifier "specifier" is not valid for a formal argument.

Static function is not defined nor referenced.
```

A function of storage class static is neither defined nor called anywhere in the compilation unit.

```
Static function is not defined.

Static function is not referenced.
```

A function of storage class static is not called anywhere in the compilation unit. Since it is not exported, there can be no reference to the function and it is essentially deleted.

```
Storage class "class" is not valid for a struct member.

Storage class "class" cannot be initialized.

Storage class for function should be static or extern.

String constant denoting section name expected.

String exceeds the length of array being initialized.

String exceeds the length of the array being initialized.

String expected.

String truncation required.
```

```
struct tag was previously declared as a union.

struct was previously defined as a union.

struct/union tag was previously declared as an enum.

struct/union tag was previously declared as an enum.

Structure has no contents (is of size zero).

Superfluous "pragma Code" specifiers.

Superfluous "pragma Data" specifiers.

Superfluous type qualifier.

Superfluous type specification.

switch expression cannot possibly have the value n.

switch statement has no cases.

Symbol declaration is inconsistent with a previous declaration at Ln/Cn.

Tag name is being truncated to n characters in length.

The rest of this line is extraneous.

Toggle "name" is unrecognized.

Toggle not recognizable.

Toggle specifier must be an identifier.

Too many arguments in "pragma Alias".

Too many arguments specified.

Type "type" is not appropriate for a conditional expression.

Type "type1" is not assignment compatible with type "type2".
```

(a) In an assignment expression, the right operand of type type1 may not be assigned to the left operand of type type2. (b) In a function call, an argument of the type type1 may not be passed to a function that expects a parameter of type type2.

```
Type "type1" is not compatible with the type "type2".
```

In a comparison, the left operand of type type1 may not be compared with the right operand, of type type2.

```
Type "type" is not valid for a struct member.

Type of entity is a yet undefined struct/union.

Type of entity is an array of unknown length.

typedef name may not be qualified with adjective.

Unable to cast type "type1" to type "type2".

Unable to determine the size of array type.

Unable to initialize array; element type not complete.

Undefined structure member.

Unexpected character "c" follows "\".

Unexpected symbol in expression.  Line ignored.

union tag was previously declared as a struct.

union was previously defined as a struct.

Unreachable statement.

Unrecognizable data section class.

Unrecognizable pragma directive or toggle: name(text)

Unrecognizable pragma directive: name

Unrecognizable pragma.

Unrecognized preprocessor directive: "name".

Unsigned compare with zero always false.

Unsigned compare with zero always true.
```

An expression of type **unsigned int** is never less than zero. Thus, a comparison to check whether the **unsigned int** expression is greater than zero will always be true, and a comparison to check if it is less than zero will always be false.

```
Up-level reference to a register-class variable is not allowed.

Value of escape sequence exceeds maximum value representable in unsigned
char.

Variable "name" is never referenced.

Variable "name" is possibly referenced before set.
```

This warning is issued by the optimizer when it has found an **auto-** or register- class variable that is "live" at its declaration. In other words, the compiler has detected a potential path in which such a variable is referenced prior to being assigned a value.

This condition is not necessarily an error in that the "potential" path may never occur. For example, the variable "x" in the following code fragment would be diagnosed with this warning:

```
static void foo(int i) {
   int x;
   if (i >= 0) x = 1;
   if (i < 0) x = 2;
   printf("x=%d\n",x);
   }
```

```
Variable expected.
```

In this context a so-called "lvalue" is required but was not found. An lvalue is something whose address can be taken, and is required on the left side of an assignment expression and as an operand to &, + +, and --. The rules of C require the automatic conversion of some objects into non-lvalues. For example, the operand of & must be an lvalue, so **int** i = &(a + b) produces the "Variable expected." diagnostic. A common cause of this message is the use of a construct such as:

```
int * p;
c = * (( char*)p)++;
```

which is legal on most PCC compilers, but disallowed by the ANSI Standard. Use instead:

```
int * p ;
c = *(* (char**)&p)++;
```

to circumvent the restriction. This solution works only when p is not a **register** variable; unfortunately we know of no solution for **register** variables.

```
Variable previously initialized at Ln/Cn.
```

```
Variables within a user-specified data section must not be "extern".
```

```
Variables within an imported data section must be "static".
```

```
"void" is not a valid argument type.
```

```
"volatile" qualifier in cast has no effect.
```

The warning is issued to remind the user that simply casting a variable to a **volatile** type does not force the variable to be **volatile**. This is because the variable being cast is actually an rvalue. For example, the cast in the following assignment accomplishes nothing:

```
int a,b;
...
a = (volatile int)b;
```

The effect of forcing b to be **volatile** can be attained by the following sequence:

```
a = *(volatile int *)&b;
```

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.

Table B-1 (Page 1 of 5). ASCII Character Set

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Comment
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
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

Table B-1 (Page 2 of 5). ASCII Character Set

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Comment
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		%	
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	

Table B-1 (Page 3 of 5). ASCII Character Set

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Comment
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	
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	

Table B-1 (Page 4 of 5). ASCII Character Set

Decimal Value	Octal Value	Hex Value	Control Character	ASCII Symbol	Comment
91	133	5B		[	
92	134	5C		\	
93	135	5D		]	
94	136	5E		^	
95	137	5F		_	underscore
96	140	60		`	grave accent
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		|	

Table B-1 (Page 5 of 5). ASCII Character Set					
Decimal Value	**Octal Value**	**Hex Value**	**Control Character**	**ASCII Symbol**	**Comment**
125	175	7D		}	
126	176	7E		~	
127	177	7F		DEL	delete

Appendix C. Program Examples for Mixing Languages

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 C, VS FORTRAN, and VS Pascal.

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 list of commands needed to compile and run the sample programs are included with each sample.

Note: On the RT PC, VS FORTRAN appends the underscore (_) character to external symbols; therefore, routines declared in C or VS Pascal will need to append an underscore to the names of routines called from or to VS FORTRAN. This does not apply to AIX/370.

C Calling FORTRAN and Pascal

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.  \n")
  exit(0);
}

cfunc ()
{
  char chrc[79];
  int count, ic;
  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 \n \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,&ic,&xc,&yc,count);     /* Arrays in C always use */
                                     /* the call-by-reference  */
                                     /* mechanism.             */
  printf("\n \n After FORTRAN call:");
  printf("\n  Text string: %s", chrc);
  printf("\n  ic = %d", ic);
  printf("\n  xc = %f", xc);
  printf("\n  yc = %f", yc);
```

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

Note: On the RT PC, the call to the procedure **fsub** must include the underscore (_) character appended to its name; that is, **fsub_**.

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)
      CHARACTER*80 WORDS
      INTEGER I
      REAL X
      DOUBLE PRECISION Y
C     The string terminator C expects is concatenated.
      WORDS='FORTRAN lives!'//CHAR(0)
      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.p".  }

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 commands needed to compile and run this sample program on the PS/2 are:

```
cc -o cexam cexam.c ccallf.f ccallp.p
cexam
```

The commands needed to compile and run this sample program on the RT are:

```
vs -o cexam cexam.c ccallf.f ccallp.p
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.
```

FORTRAN Calling Pascal and C

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
      INTEGER COUNT
      REAL XFOR
      DOUBLE PRECISION YFOR
      CHARACTER*10 CHRFOR
      CHARACTER*5 CHRFIVE
```

```
C     Some data is initialized.
      EQUIVALENCE (CHRFOR,LETTER)
      CHRFOR='HELLO'
      CHRFIVE='AGAIN'
      IFOR=50
      COUNT=10
      XFOR=10
      YFOR=0.

      WRITE(*,100) CHRFOR,IFOR,XFOR,CHRFIVE,YFOR
 100  FORMAT(/'Before calls:'/' IFOR='I10/' Text string: *'A'*'
     +      ' XFOR='F10.2/' Text string: *'A'*'/' YFOR='F10.2)


C     A Pascal procedure is called.
      CALL PSUB(IFOR,CHRFOR,XFOR,CHRFIVE,YFOR)
      WRITE(*,110) IFOR,CHRFOR,XFOR,CHRFIVE,YFOR
 110  FORMAT(/'After Pascal call:'/' IFOR='I10/' Text string: *'A'*'
     +      ' XFOR='F10.2/' Text string: *'A'*'/' YFOR='F10.2)
C     A C subroutine is called.
      CALL CSUB(IFOR,CHRFOR,XFOR,CHRFIVE,YFOR)
      WRITE(*,120) IFOR,CHRFOR,XFOR,CHRFIVE,YFOR
 120  FORMAT(/'After C call:'/' IFOR='I10/' Text string: *'A'*'
     +      ' XFOR='F10.2/' Text string: *'A'*/' 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.p".  }

segment DUMMYNAME;

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

{ Note that the incoming arguments which were passed
       as FORTRAN characters arrive as pointers in their
       actual locations, with their lengths, in order, at
       the end of the argument list. }

       procedure PSUB (var I     : INTEGER;
                       var WORD1 : TEXT;
                       var X     : SHORTREAL;
                       var WORD2 : TEXT;
                       var Y     : REAL;
                       LEN1      : INTEGER;
                       LEN2      : INTEGER) ;external;
```

```
procedure PSUB ;
  var J : INTEGER;
  begin
   WORD1[0] := 'B'; WORD1[1] := 'Y'; WORD1[2] := 'E';
   FOR J := 3 TO LEN1-1 DO
      WORD1[J] := ' ';
   FOR J := 0 TO LEN2-1 DO
      WORD2[J] := 'P';
   X := X * I;
   I := LEN1;
   Y := 1.0d0;
   end;
```

Note: On the RT PC, the external declaration and definition for procedure **PSUB** must include the underscore (_) character appended to its name; **PSUB_**.

The Called C Function

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


void csub (  i,
             word1,
             x,
             word2,
             y,
            len1,  /* Note the hidden length arguments corresponding */
            len2)  /* to word1 and word2                             */


char word1[79],word2[79];
int *i;
int len1,len2;
float *x;
double *y;

{
      int j;

        word1[0] = 'h';    /* Arrays in C always use the   */
        word1[1] = 'i';    /* call-by-reference mechanism. */
        word1[2] = ' ';
        word1[3] = 'C';
        for (j=4; j<len1-1; j++)
            word1[j] = ' ';

        *i = -3;
        *x = -(*x);
        *y = 2.0;
        word2[0] = 'C';
        word2[1] = 'L';
        word2[2] = 'o';
        word2[3] = 'n';
        word2[4] = 'g';
}
```

Note: On the RT PC, the definition of the function **csub** must include the underscore (_) character appended to its name; **csub_**.

Commands and Output

The commands needed to compile and run this sample program on the PS/2 are:

```
cc -o forexam forexam.f fcallp.p fcallc.c
forexam
```

The commands needed to compile and run this sample program on the RT are:

```
vs -o forexam forexam.f fcallp.p fcallc.c
forexam
```

The output from running this sample program is:

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

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

After C call:
 IFOR=        -3
 Text string: *hi C      * XFOR=   -500.00
 Text string: *CLong*
 YFOR=      2.00

 I've safely returned after doing all that!
```

Pascal Calling FORTRAN and C

The Calling Pascal Program

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

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

   procedure FSUB  (var NAMES : TEXT;
                    var IPAS  : INTEGER;
                    var XPAS  : SHORTREAL;
                    var YPAS  : REAL;
                        LEN1  : INTEGER ); external;

   procedure CSUB  (    NAMES : TEXT;
                    var IPAS  : INTEGER;
                    var XPAS  : SHORTREAL;
                    var 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:2,'  XREAL=',XREAL,'  YDOUB=',YDOUB);
  FSUB (CHRPAS,INTPAS,XREAL,YDOUB,20);

  writeln;
  writeln ('After FORTRAN call:');
  write(' Text: *'); for I := 0 to 20 do write(CHRPAS[I]);
  writeln('*');
  writeln(' INTPAS=',INTPAS:2,'  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:2,'  XREAL=',XREAL,'  YDOUB=',YDOUB);
  writeln;
end;


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

Note: On the RT PC, the external declaration and call to the procedure **FSUB** must include the underscore (_) character appended to its name; **FSUB_**.

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
      DOUBLE PRECISION Y

      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,
          i,
          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);
}
```

Commands and Output

The commands needed to compile and run this sample program on the PS/2 are:

```
cc -o pasexam pasexam.p pcallf.f pcallc.c
pasexam
```

The commands needed to compile and run this sample program on the RT PC are:

```
vs -o pasexam pasexam.p pcallf.f pcallc.c
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= 1.0000000000000E+01  YDOUB= 0.0000000000000E+000
After FORTRAN call:
 Text: *FORTRAN lives!      *
 INTPAS=20  XREAL= 2.0000000000000E+02  YDOUB= 1.0000000000000E+000
After C call:
 Text= *hi C *
 INTPAS=-3  XREAL=-1.0000000000000E+00  YDOUB= 1.0000000000000E+000

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

Appendix D. C Compiler Limits

370	PS/2	RT	PC
PREPROCESSOR STATEMENTS			
– nesting levels for include files	*	*	*
– macro identifiers defined in a file	*	*	*
– arguments in one macro invocation	31[1]	31	*
IDENTIFIERS			
– significant chars in an internal identifier	*	64	64
– significant chars in an external identifier	*	64	64
– external identifiers declared in a file	*	*	*
– block-scope identifiers in one block	*	*	*
DECLARATORS			
– variables in a single declaration	*	*	*
– enumeration constants	*	*	*
– members in a structure or union	*	*	*
– initialization elements	*	*	*
– nesting levels of structure or unions	*	*	*
– characters in a string literal	*	32767[2]	*
STATEMENTS			
– case labels for a switch statement	*	*	2000
FUNCTIONS			
– parameters in one function definition	*	*	*
– arguments in one function call	*	*	*

Note:

[1]This means that in preprocessor statements, arguments in one macro invocation are limited to 31, but not limited in the C inboard cpp.

[2]This includes the null character.

[3]The item is not explicitly limited by the C Compiler.

Index

D

E

F

G

H

I

L

M

Reader's Comment Form

C Language User's Guide **SC23-2057-01**

This publication is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

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

Is there anything you especially like or dislike about the organization, presentation, or writing in this publication? Helpful comments include general usefulness; possible additions, deletions, and clarifications; specific errors and omissions.

Page Number: Comment:

What is your occupation? _____

If you wish a reply, give your name and address: _____

_____

_____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments.)

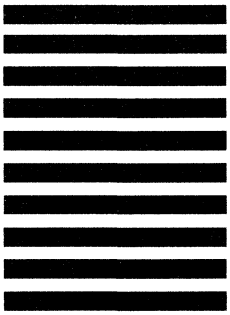
NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department C7D
36 Apple Ridge Road
Danbury, CT 06810



Fold

Fold

Cut or Fold Along Line

Fold and Tape

Please Do Not Staple

Fold and Tape

International Business
Machines Corporation
36 Apple Ridge Road
Danbury, CT 06810

Printed in the
United States of America

SC23-2057-01

IBM®

SC23-2057-01

71F0390