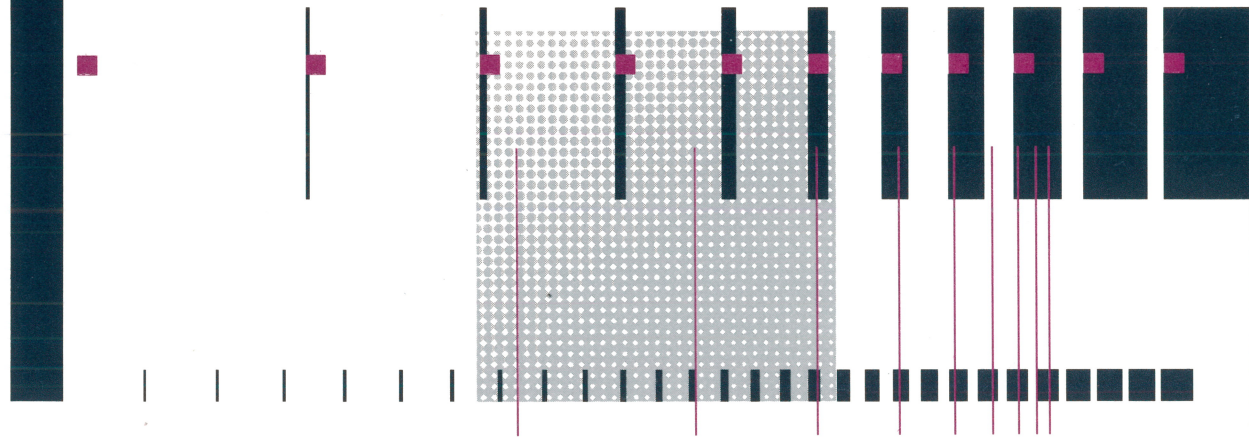


**RISCompiler Languages
Programmer's Guide**

Order Number 3200DOC



The power of RISC is in the system.

***RISCompiler Languages
Programmer's Guide***

Order Number 3200DOC

December 1988

Your comments on our products and publications are welcome. A postage-paid form is provided for this purpose on the last page of this manual.

© 1988 MIPS Computer Systems, Inc. All Rights Reserved.

RISCompiler and RISC/os are Trademarks of MIPS Computer Systems, Inc.
UNIX is a Trademark of AT&T.
Ethernet is a Trademark of XEROX.

IBM is a registered trademark of International Business Machines Corporation.

MIPS Computer Systems, Inc.
930 Arques Ave.8
Sunnyvale, CA 94086

Customer Service Telephone Numbers:

California:	(800)	992-MIPS
All other states:	(800)	443-MIPS
International:	(415)	330-7966

December 1988 Edition

The following summarizes the changes made to the former (February 1987) edition of this manual that appear in this edition:

New Compiler Options. The `-cord` and `-feedback` driver options were added to the summary of driver options in the table on p. 1–8. The *Reducing Cache Conflicts* section in **Chapter 4** has been added to show how use of these options can create significant improvements in program performance.

New Link Editor Options: The `-jmopt`, and `-nojmpopt` link editor options are described in Table 1.1 in **Chapter 1**. The *Filling Jump Delay Slots* section in **Chapter 4** describes when to use these options.

Pascal: the text in **Chapter 2** (pp. 2–9 – 2–9) concerning the mapping of Pascal objects has been greatly expanded with additional rules and examples. Additional information has also been provided in Chapter 3 (p. 3–2) on the interface between programs written in Pascal and those written in C.

Index. Approximately 200 entries have been added to the Index, enhancing the ability to retrieve information from this manual more efficiently.

General. Numerous minor technical and editorial corrections have been made throughout the manual.

This book provides information on the compilers and high-level languages that comprise the MIPS RISCompiler System.

The RISCompiler system provides a consistent programming environment for all currently supported languages. This book describes the components and programming tools that comprise the compiler system.

Scope

Although the programming environment includes all standard UNIX driver commands and system tools, this book does not describe those tools in detail. For details, you may need to refer to the *User's Reference Manual* and other associated publications. This book contains implementation details on the supported languages, but does not contain detailed reference information giving the syntax and definition of each language.

Audience

This book assumes that you are fluent in the programming language you're using and that you are comfortable using the tools of the UNIX system (System V or BSD) to write your programs. It also assumes that you are using a MIPS RISComputer to compile your programs.

If you need to compile, to debug, to profile, or to optimize, code, you need to read this book.

Topics Covered

This book has these chapters:

Chapter 1: The Compiler System. Gives an overview of components of the compiler system and provides reference and guide information in using the various options provided by the compiler drivers.

Chapter 2: Storage Mapping. Describes storage mapping for variables in C and Pascal.

Chapter 3: Language Interfaces. Provides reference and guide information in writing programs in C and Pascal that can communicate with each other.

Chapter 4: Improving Program Performance. Describes the profiling and optimization facilities available to increase the efficiency of your programs, and how to use them.

Chapter 5: Debugging Your Code. Shows you how to use the features of the source level debugger.

Appendix A: C Implementation. Describes extensions and modifications supported by the C compiler that differ from other C implementations.

Appendix B: Pascal Implementation. Describes language extensions and modifications supported by the Pascal compiler that differ from other Pascal implementations.

Appendix C: Byte Ordering. Describes how the big endian and little endian affect the mapping of data in storage.

Index. Contains index entries for this publication.

Publications Index. Contains index entries to other MIPS publications.

For More Information

You may need to refer to the following as you use this manual:

MIPS Assembly Language Programmer's Guide 3201DOC

RISC/os User's Reference Manual 3204DOC

ar(1)

dbx(1)

prof(1)

cc(1)

f77(1)

pc(1)

ld(1)

dump(1)

file(1)

nm(1)

About This Book

Scope	iii
Audience	iii
Topics Covered	iii
For More Information	iv

1

The Compiler System

Overview	1-1
The Drivers	1-1
Languages Supported	1-2
Driver Commands	1-2
Files	1-3
Operational Overview	1-3
Default Options	1-5
Compiling Multi-Language Programs	1-6
Linking Objects	1-6
Compilation Options	1-7
General Options	1-8
Byte Ordering Options	1-10
Debugging Options	1-11
Profiling Option	1-12
Optimizer Options	1-12
Compiler Development Options	1-12
Including Common Files (Definition Files)	1-12
Link Editor	1-14
Running the Link Editor	1-14
Specifying Libraries	1-14
Link Editor Options	1-15
Object File Tools	1-20
Dumping Selected Parts of Files (odump)	1-20
Listing Symbol Table Information (nm)	1-27
Determining a File's Type (file)	1-31
Determining a File's Section Sizes (size)	1-31
Archiver	1-32
Examples	1-33
Archiver Options	1-34

2

Storage Mapping

C Language	2-1
Alignment, Size, and Value Ranges	2-1
C Arrays, Structures, and Unions	2-2
Storage Classes	2-6
Pascal	2-8
Alignment, Size, and Value Ranges	2-8
Pascal Arrays, Records and Variant Records	2-11
Rules for Set Sizes	2-17

3

Language Interfaces

Pascal/C Interface	3-1
General Considerations	3-1
Calling Pascal from C	3-4
Calling C from Pascal	3-7
.....	3-10

4

Improving Program Performance

Introduction	4-1
Profiling	4-1
Overview	4-1
How Basic Block Counting Works	4-8
Averaging Prof Results	4-10
How PC-Sampling Works	4-12
Creating Multiple Profile Data Files	4-13
Running the Profiler (prof)	4-13
Optimization	4-16
Optimization Options	4-19
Full Optimization (-O3)	4-20
Optimizing Large Programs	4-21
Optimizing Frequently Used Modules	4-22
Building a Ucode Object Library	4-24
Using Ucode Object Libraries	4-24
Improving Global Optimization	4-24
Improving Other Optimization	4-29
Limiting the Size of Global Data Area	4-31
Purpose of Global Data	4-31
Controlling the Size of Global Data Area	4-31
Obtaining Optimal Global Data Size	4-32
Examples (Excluding Libraries)	4-32
Example (Including Libraries)	4-33
Reducing Cache Conflicts	4-33
Filling Jump Delay Slots	4-34

Introduction	5-2
Why Use a Source-Level Debugger?	5-2
What Are Activation Levels?	5-3
Isolating Program Failures	5-3
Incorrect Output Results	5-4
Avoiding Some Pitfalls	5-4
Running DBX	5-5
Compiling Your Program for Debugging	5-5
Building a Command File	5-5
Invoking DBX (dbx)	5-6
Ending DBX (quit)	5-7
Using DBX Commands	5-7
DBX Command Syntax	5-7
Qualifying Variable Names	5-9
DBX Expressions and Precedence	5-9
DBX Data Types and Constants	5-10
Basic DBX Commands	5-12
Working with the DBX Monitor	5-12
Using History (history & ! commands)	5-13
Editing on the DBX Command Line	5-13
Typing Multiple Commands	5-14
Completing Program Symbol Names	5-15
Controlling DBX	5-16
Setting DBX Variables (set)	5-16
Removing Variables (unset)	5-17
Predefined DBX Variables	5-18
Creating Command Aliases (alias)	5-22
Removing Command Aliases (unalias)	5-22
Predefined DBX Aliases	5-23
Recording Input (record input)	5-25
Recording Output (record output)	5-26
Playing Back the Input (source or playback input)	5-27
Playing Back the Output (playback output)	5-28
Invoking a Shell from DBX (sh)	5-29
Checking the Status (status)	5-29
Deleting Status Items (delete)	5-30
Examining Source Programs	5-31
Specifying Source Directories (use)	5-31
Moving to a Specified Procedure (func)	5-31
Specifying Source Files (file)	5-32
Listing Your Source Code (list)	5-33
Searching Through the Code (/ and ?)	5-34
Calling an Editor from DBX (edit)	5-34
Printing Symbolic Names (which and whereis)	5-35
Printing Type Declarations (whatis)	5-35

Controlling Your Program	5-36
Running Your Program (run and rerun)	5-36
Executing Single Lines of Your Code (step and next)	5-37
Returning from a Procedure Call (return)	5-38
Starting at a Specified Line (goto)	5-39
Continuing after a Breakpoint (cont)	5-39
Assigning Values to Program Variables (assign)	5-40
Setting Breakpoints	5-41
Overview	5-41
Setting Breakpoints at Lines (stop at)	5-41
Setting Breakpoints in Procedures (stop in)	5-42
Setting Conditional Breakpoints (stop if)	5-43
Tracing Variables (trace)	5-43
Writing Conditional Code in DBX (when)	5-44
Stopping at Signals (catch and ignore)	5-45
Examining Program State	5-46
Doing Stack Traces (where)	5-46
Moving Up and Down the Stack (up, down)	5-47
Printing (print and printf)	5-48
Printing Register Values (printregs)	5-49
Printing Information about Activation Levels (dump)	5-50
Debugging at the Machine Level	5-51
Setting Breakpoints in Machine Code (stopi)	5-52
Continuing after Breakpoints in Machine Code (conti)	5-53
Executing Single Lines of Machine Code (stepi and nexti)	5-53
Tracing Variables in Machine Code (tracei)	5-54
Printing the Contents of Memory	5-55
Debugger Command Summary	5-57
Sample Program	5-64

Appendix A
C Implementation

Vararg.h Macros	A-1
Deviations	A-2
Extensions	A-3
Translation Limits	A-3

Appendix B
Pascal Implementation

Names	B-1
Use of Underscores	B-1
Lowercase in Public Names	B-1
Alphabetic Labels	B-2
Constants	B-2
Non-Decimal Number Constants	B-2
String Padding	B-2
Non-Graphic Characters	B-3
Constant Expressions	B-3

Statement Extensions	B-5
Otherwise Clause in Case Statement	B-5
Return Statement	B-5
Continue Statement	B-6
Break Statement	B-6
Declaration Extensions	B-6
Separate Compilation	B-6
Shared Variables	B-8
Initialization Clauses	B-9
Relax Declaration Ordering	B-10
Predefined Procedures	B-10
Assert	B-10
Argv	B-10
Date	B-10
Time	B-10
Predefined Functions	B-10
Type Functions	B-10
Min	B-11
Max	B-11
Lbound	B-11
Hbound	B-11
First	B-12
Last	B-12
Argc	B-12
Clock	B-12
Bitand	B-12
Bitor	B-12
Bitxor	B-12
Bitnot	B-12
Lshift	B-13
Rshift	B-13
I/O Extensions	B-13
Specifying Radix in the Write Statement	B-13
Filename on Rewrite and Reset	B-13
Reading Character Strings	B-13
Reading and Writing Enumeration Types	B-14
Lazy I/O	B-15
Standard Error	B-15
Predefined Data Type Extensions	B-15
Double	B-15
Cardinal	B-15
Pointer	B-15
Compiler Notes	B-15
Macro Preprocessor	B-15
Short Circuiting	B-16
Translation Limits	B-16

Appendix C
Byte Ordering

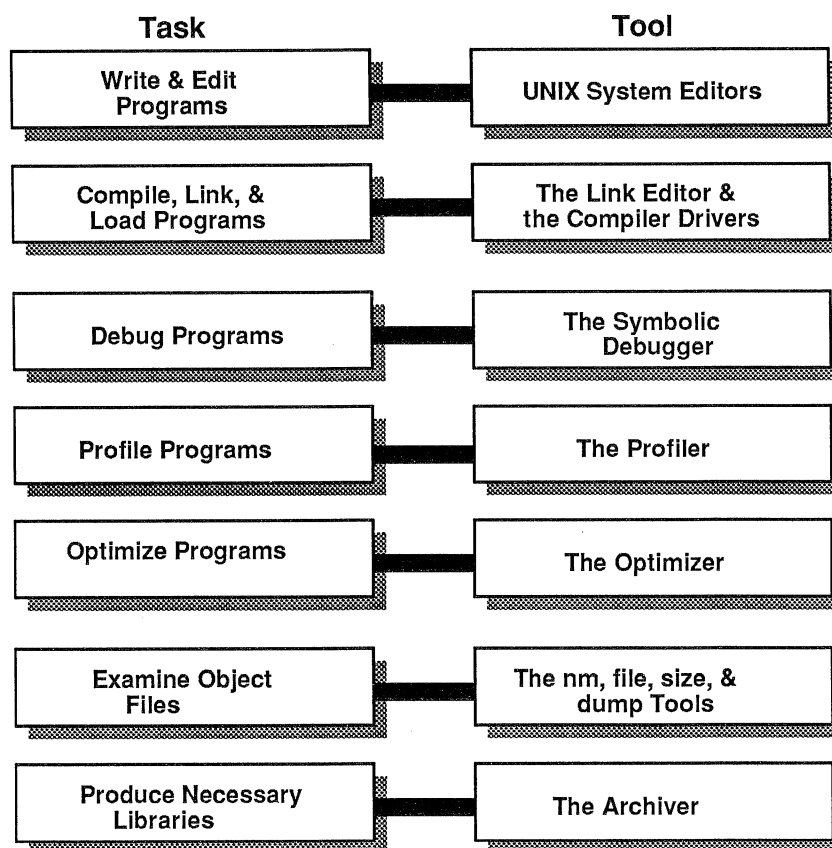
What Is Byte Ordering? C-1
Big-Endian Byte Ordering C-1
Little-Endian Byte Ordering C-2

Index

Chapter 1 describes the components of the compiler system and how to use them.

Overview

The components that comprise the compiler system and the task each performs are summarized in the following figure:



The Drivers

Intelligent programs called *drivers* actually invoke the following major components of the compiler system: the macro preprocessor (cpp), the compilers (C, FORTRAN 77, COBOL, PL/I or Pascal), the assembler, and the link editor. A separate driver exists for each language. This section gives an overview of driver operations and commands.

Languages Supported

The table below shows the languages supported by the compiler system and the driver name that invokes the respective drivers:

Language	Driver Name	Operands
C	cc	[compiler options] [link editor options] [source name list]
Pascal	pc	[compiler options] [link editor options] [source name list]
FORTRAN 77	f77	[compiler options] [link editor options] [source name list]
COBOL	cobol	[compiler options] [link editor options] [source name list]
PL/I	pl1	[compiler options] [link editor options] [source name list]
MIPS Assembly	as	[compiler options] [source name list]

NOTE: The languages supported by any one system is an optional choice made at purchase. Thus, the configuration of your particular system may not support all of the above languages.

Driver Commands

The commands **cc(1)**, **pc(1)**, **cobol(1)**, **pl1(1)** and **f77(1)**, and **as(1)** run the drivers that cause your programs to be compiled (if in a high-level language), optimized, assembled, and link edited.

Each command knows the appropriate libraries associated with the main program and passes only those libraries to the link editor.

Files

The driver recognizes the contents of an input file by the suffix assigned to the filename, as shown below.

File Suffixes	
Suffix	Description
.p	Pascal source code
.u	ucode object file
.a	object library
.b	ucode object library
.c	C source code
.cob	COBOL source code
.e	efl source
.f	Fortran 77 source
.i	The driver assumes that the source code was processed by the C preprocessor (cpp) and that source code is that of the processing driver. For example: <pre>pc -c source.i</pre> <i>source.i</i> is assumed to contain Pascal source statements.
.o	object file
pl1 or pli	PL/1 source code
.r	ratfor source code
.s	assembly source code

NOTE: The assembly driver **as** assumes that any file, regardless of the suffix, contains assembly language statements; **as** accepts only one input source file.

Operational Overview

Figure 1.1 on the next page show the relationship between the major components of the compiler system and their primary inputs and outputs.

Note that FORTRAN uses preprocessors (see Figure 1.2) that the other languages do not use. For more information, see the **efl(1)**, **ratfor(1)**, and **m4(1)** manual pages in the *User's Reference Manual*.

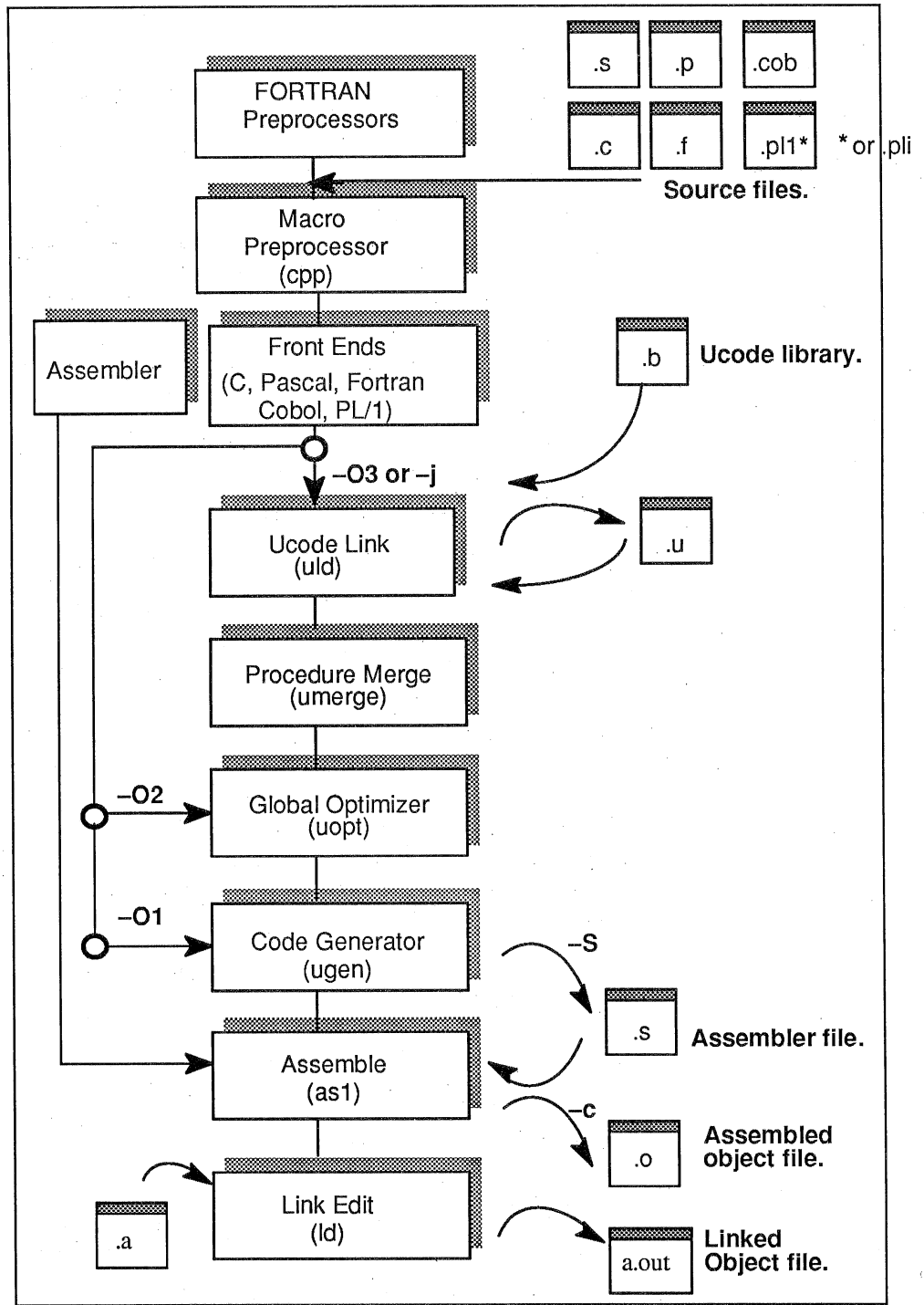


Figure 1.1. The Compiler System Driver.

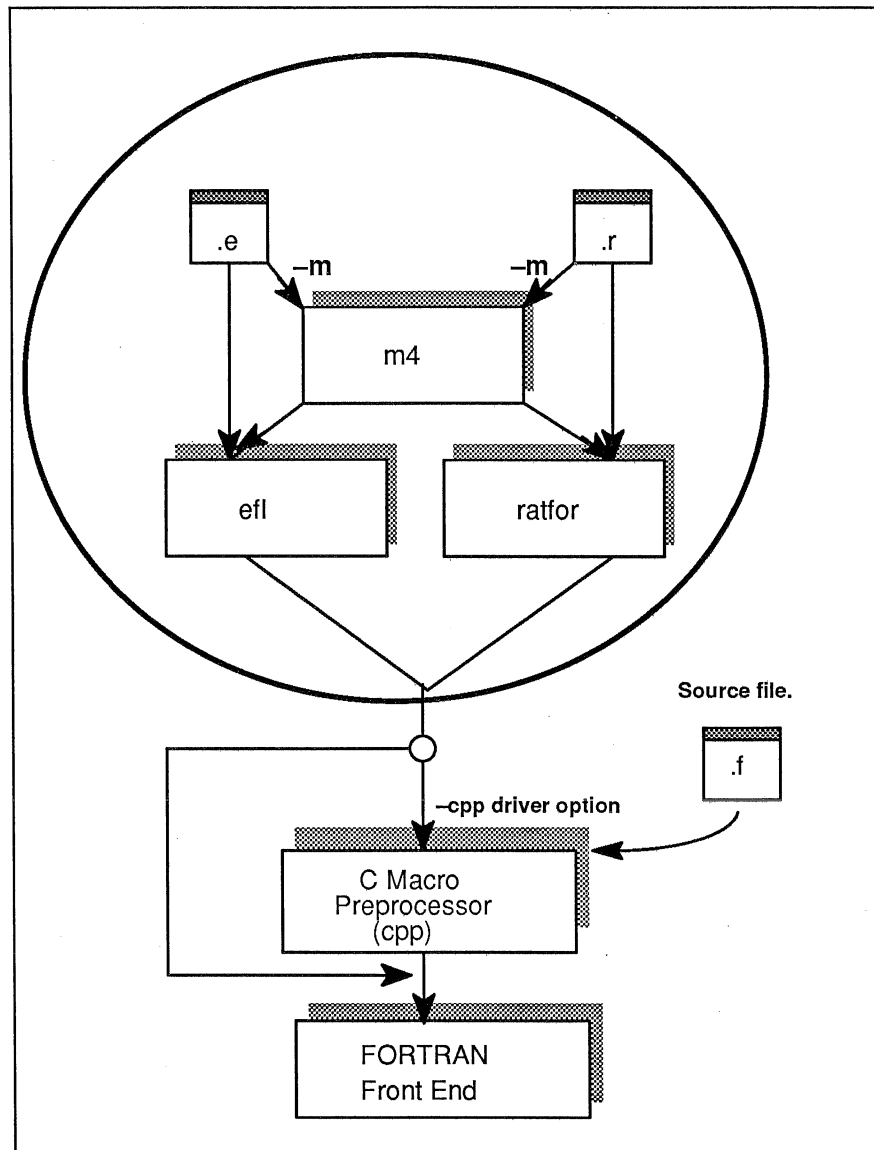


Figure 1.2 The FORTRAN Preprocessors. See Figure 1.1.

Default Options

At compilation, you can select one or more options that affect a variety of program development functions, including debugging, optimization, and profiling facilities, and the names assigned to output files.

Some options have defaults, which apply even if you don't specify them. For example, the default names for output files are *filename.o* for object files, where *filename* is the base name of the source file; the default name for executable

program objects is a.out. The following example uses the defaults in compiling source files foo.c and bar.c:

```
% cc foo.c bar.c
```

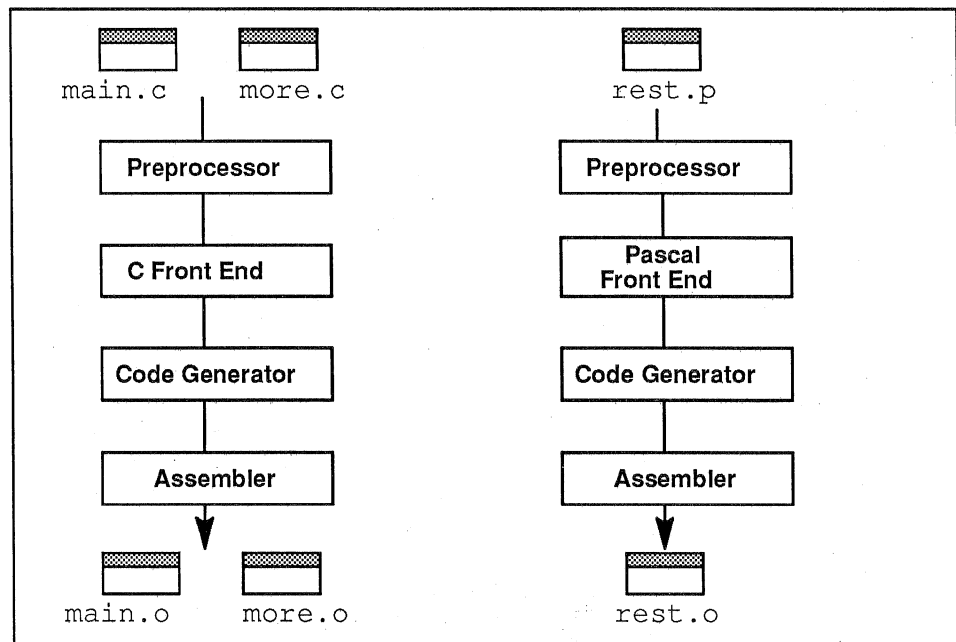
← runs the C compiler, creates object modules foo.o and bar.o, and the executable program a.out.

Compiling Multi-Language Programs

When the source language of the main program differs from that of a subprogram, you should compile each program module separately with the appropriate driver and then link them in a separate step. You can create objects suitable for link editing by specifying the `-c` option, which stops the driver immediately after the assembler phase. For example:

```
% cc -c main.c more.c
% pc -c rest.p
```

The figure below shows the compilation control flow for these two commands.



Linking Objects

You can also use a driver command to link edit separate objects into one executable program. The driver recognizes the `.o` suffix as the name of a file containing object code suitable for link editing and immediately invokes the link editor. You could link edit the object created in the last example using Pascal driver `pc`, as shown below:

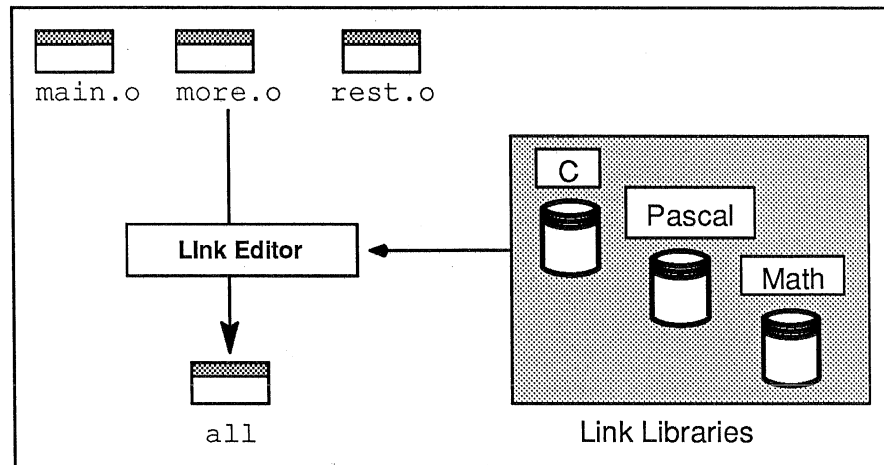
```
% pc -o all main.o more.o rest.o
```

This statement produces the executable program object *all*. You could achieve the same results using the C driver *cc*, as shown below:

```
% cc -o all main.o more.o rest.o -lp -lm
```

The *cc* driver needs two additional options, which *pc* uses by default and which are specified using the link editor *-l* option: *-lp* (which specifies the Pascal link library) and *-lm* (which specifies the math link library). Both *pc* and *cc* use the C link library by default.

The figure below shows the flow of control for both the *pc* and *cc* commands listed above.



For more information on the link editor and on specifying link libraries, see the **Link Editor** section of this chapter. For a detailed listing of the default libraries used by each driver, see the *cc(1)*, *f77(1)*, *pc(1)*, *cobol(1)*, or *pl1(1)* manual page, as applicable, in the *User's Reference Manual*.

Compilation Options

The tables on the following pages summarize the options you can specify for the compilation phases, which include the preprocessing phase through the assembly phase (Figure 1.1 shows these phases); the options summaries are divided into the following major groups:

- General Options
- Byte Ordering Options
- Debugging Options
- Profiling Option
- Optimizer Options
- Compiler Development Options

NOTE: The tables list only the most frequently used options; they don't list all available options. See the `cc(1)`, `f77(1)`, `pc(1)`, `cobol(1)`, or `pl1(1)` manual page, as applicable, in the *User's Reference Manual* for a complete list of options available.

General Options

The general options are listed in alphabetical order in the tables that follow.

General Compiler Options	
Option Name	Purpose
<code>-5</code>	Use the System V compatible include files and libraries instead of the 4.3 BSD default include files and libraries.
<code>-c</code>	Prevents the link editor from linking your program after compilation. This option forces the compiler to produce a <code>.o</code> file when you compile only one program.
<code>-cord</code>	Re-arranges the procedures in the link-edit object file to reduce cache conflicts in the executable object (<code>a.out</code>). At least one <code>-feedback</code> file must be specified. See <i>cache conflicts</i> and the <code>-cord</code> option <i>profiling</i> for more information.
<code>-cpp</code>	Run the C macro preprocessor on the source code before compiling. The default varies from driver to driver. Refer to the appropriate man page (<code>cc</code> , <code>pc</code> , <code>as</code> , etc) in the <i>User's Reference Manual</i> for more information.
<code>-C</code>	C and assembler drivers only. Used with the <code>-P</code> and <code>-E</code> options. Prevents the macro processor from stripping comments. Use this option when you suspect the preprocessor is not emitting the intended code and you wish to examine the code with its contents.
<code>-C</code>	Pascal and FORTRAN drivers only. Generates code that causes range checking for subscripts during program execution.
<code>-feedback file</code>	Produces (together with the <code>-cord</code> option) an object with procedures rearranged so as to reduce cache conflicts; <i>file</i> is the output produce when running <code>-prof</code> with the <code>-prof -feedback</code> option specified. See <i>cache conflicts</i> and the <code>-cord</code> option for details.
<code>-E</code>	Runs only the C macro preprocessor and sends results to the standard output. Specify also <code>-C</code> to retain comments. Use <code>-E</code> when you suspect the preprocessor isn't emitting the intended code.
<code>-D name</code> or <code>-D name=def</code>	Defines a macro name if you specified a <code>#define</code> in your program. Unless you specify a definition <code>name=def</code> , the compiler defines the name to be "1".

General Compiler Options	
Option Name	Purpose
-G <i>num</i>	<i>num</i> is a decimal number that specifies the maximum size in bytes of an item to be placed in the global pointer area. The default is 8 bytes. You can raise or lower <i>num</i> to control the number of data items placed in these sections. See <i>Limiting the Size of Global Pointer Data</i> in Chapter 4 for examples of using -bestGnum and its related options.
-I <i>dirname</i>	Compiler searches the current directory <i>dirname</i> , and the default directory, /usr/include , in that order for the include file.
-I	When specified in addition to the -I <i>dirname</i> , the compiler searches only <i>dirname</i> and does not search the default directory.
-j	Creates a file suffixed with a <i>.u</i> that contains ucode, an intermediate code used by the compiler for internal processing. See the Optimization section in Chapter 4 for examples of using the -j option.
-k <i>option</i>	<i>option</i> is one of the link editor options shown in Table 1-1 later in this chapter. The driver passes it to the ucode loader, which then performs the link action specified by <i>option</i> .
-ko <i>filename</i>	<i>filename</i> is the name of the output file to be created by the ucode loader.
-nocpp	Do not run the C macro preprocessor on C and assembly source files before processing. See also the -cpp option.
-o <i>filename</i>	Assigns the name <i>filename</i> to the program object. When used with the -c option, tells where to leave <i>.o</i> file. The default filename is a.out .
-P	Same as -E options, except puts results in a <i>.i</i> file. Specify both -P and -C to retain comments.

General Compiler Options (2 of 3).

General Compiler Options	
Option Name	Purpose
<code>-p1</code> or <code>-p</code>	Permits program counter (pc) sampling. This option provides operational statistics for use in improving program performance. See Chapter 4 for more details. Note: This option affects only the link editor and is ignored by the compiler front ends. When link editing as a separate step from compilation, be sure to specify this option if pc sampling is desired.
<code>-S</code>	Similar to <code>-c</code> , except produces assembly code in a .s file instead of object code in a .o file.
<code>-std</code>	Issues a warning message when the compiler finds a non-standard feature in the programming language of your source program.
<code>-U name</code>	Overrides a definition of a macro name that you specified with the <code>-D</code> option, or that is defined automatically by the driver.
<code>-v</code>	Lists compiler phases as they are executed. Use this option when you suspect a phase isn't being run as you intended. For example, the option might reveal that you failed to specify a library required by the link editor. For BSD 4.3 users, this option also prints resource usage of each phase.
<code>-V</code>	Prints the version number of the driver and its phases. When reporting a suspected compiler problem, you must include this number.
<code>-w</code>	Suppresses warning messages.

General Compiler Options (3 of 3).

Byte Ordering Options

The compiler can produce program objects executable on target machines with either a big-endian or little-endian byte ordering scheme. By default, the compiler produces program objects executable on target machines with the same byte ordering scheme as the compilation machine. See **Appendix D** for more information on big and little endian byte ordering.

When the byte ordering scheme on the compilation machine differs from that on the target machine, you must specify one of the options shown in the following table:

Byte Ordering Options	
Option Name	Purpose
-EB	Produces an object file for a target machine that uses a big-endian scheme. You should use this option when compiling on a little endian-machine.
-EL	Produces an object file for a target machine that uses a little-endian scheme. you should use this option when compiling on a big-endian machine. When working with the symbol table, note that the auxiliary table has the same byte ordering as the compilation machine.

Debugging Options

The table below lists the compiler options available for debugging source code using **dbx**, whose functions and operations are described in **Chapter 5**.

Debugging Options	
Option Name	Purpose
-g0*	Produces a program object without debugging information. Reduces the size of the program object and should be used when debugging is no longer required. Retains all optimizations.
-g1	Permits accurate, but limited, source-level debugging. This option does most optimizations.
-g or -g2	Permits full source-level debugging. These options often suppress optimizations that might interfere with full debugging.
-g3	Permits full, but inaccurate, debugging on fully optimized code. Debugger output may be confusing or misleading. Specify this option for programs that malfunction only after you attempt to optimize them.
*Default option	

Profiling Option

The compiler system permits the generation of profiled programs that, when executed, provide operational statistics. This is done through compiler option `-p` (which provides pc sampling information) and the `pixie` program (which provides profiles of basic block counts). See **Chapter 4** for details.

Optimizer Options

The table below summarizes the options available for program optimization. However, to fully understand the benefits of optimization and how the compiler achieves optimization, you should read the **Optimization** section in **Chapter 4** of this manual. You should also refer to the `cc(1)`, `f77(1)`, `pc(1)`, `cobol(1)`, or `pl1(1)` manual page, as applicable, in the *User's Reference Manual* for details on the `-O3` option, and the input and output files related to this option.

Optimizer Options	
Option Name	Purpose
<code>-O</code> or <code>-O2</code>	Global optimization. Optimizes within the bounds of individual compilation units. This option executes global optimizer (uopt) phase.
<code>-O0</code>	No optimization. Prevents all optimizations, including the minimal optimization normally performed by the code generator and assembler.
<code>-O1*</code>	The assembler and the code generator perform as many optimizations as possible without affecting compile-time performance.
<code>-O3</code>	Performs global register allocation across the bounds of individual compilation units. Executes the <code>uld</code> , <code>merge</code> , and <code>uopt</code> phases of the compiler system.
*Default option	

Compiler Development Options

In addition to the standard options, each driver also has options that you normally won't use. These options primarily aid compiler development work. For information about how to use these options, consult the appropriate manual page—`cc(1)`, `pc(1)`, `f77(1)`, `cobol(1)`, or `pl1(1)`—in the *User's Reference Manual*.

Including Common Files (Definition Files)

When you write programs, often you have common definition files that you share among a program's modules. Common files define things like known constants or the parameters for system calls (for example, the files that define the object file formats).

Because globally shared things should go in one place, you need a way to put these things in a common place. Definition files (often called header files in the

C programming language) let you share common information between many files in a program.

Many people call these files *#include* or “header” files. These files have a “.h” suffix. Typically, a manual page from the *User’s Reference Manual* tells you to include a specific definition file.

Each supported language handles these files the same way, and you specify these files in your program’s source code.

NOTE: If you intend to debug your program using DBX (Chapter 5), you should not place executable code in an include file. The debugger recognizes an include file as one line of source code; none of the source lines in the file appears during the debugging session.

You can include files in your program source files in either of two ways:

1. In column 1 of your source file, type:

```
#include "filename"
```

where *filename* is the name of the include file. Because you placed *filename* within double quotation marks (“”), the C macro preprocessor searches in sequence the current directory and the default directory */usr/include*.

2. In column 1 of your source file, type:

```
#include <filename>
```

where *filename* is the name of the include file. Because you placed *filename* between the greater-than and less-than signs (<>), the C macro preprocessor skips the current directory and searches only the default directory */usr/include* for the include file.

C, Pascal, FORTRAN 77, and assembly code can reside in the same include files, and then can be conditionally included in programs as required. To set up a shareable include file, you must create a *.h* file and enter the respective code as indicated below:

```
#ifdef LANGUAGE_C
:           ← C code
#endif
#ifdef LANGUAGE_PASCAL
:           ← Pascal code
#endif
#ifdef LANGUAGE_FORTRAN
:           ← Fortran code
#endif
#ifdef LANGUAGE_ASSEMBLY
:           ← MIPS Assembly code
#endif
```

NOTE: When you write your program, you need to include the *.h* file that you created.

Link Editor

This section summarizes the functions of the link editor and how it works. Refer to the `ld(1)` manual page in the *User's Reference Manual* for complete information on the link editor options and libraries.

The link editor combines one or more object files (in the order specified) into one program object file, performing relocation, external symbol resolutions, and all the other processing required to make object files ready for execution. Unless you specify otherwise, the link editor names the program object file *a.out*. You can execute the program object or use it as input for another link editor run.

The link editor supports all the standard command line features of other UNIX system link editors except System V *ifiles*. (An *ifile* holds a description of a load module.)

Running the Link Editor

You can run the link editor by typing `ld` on the command line of your shell or by using one of the driver commands as described in this chapter in the section **Linking Objects**. The syntax of the `ld` command is as follows:

Syntax:

```
ld -options object1 [object2..objectN]
```

NOTE: The assembler driver `as` does not run the link editor. To link edit a program written in assembly language, do either of the following:

- Assemble and link edit using one of the other driver commands (`cc`, for example). The *.s* suffix of the assembly language source file causes the driver to invoke the assembler procedures.
- Assemble the file using `as`, then link edit the resulting object file with the `ld` command.

Specifying Libraries

If you compile multi-language programs, be sure to explicitly load any required runtime libraries. For example, if you write your main program in C, and some procedures in Pascal, you must explicitly load the Pascal library *libp.a* and the main library *libm.a* with the options `-lp` and `-lm` (abbreviations for the libraries *libp.a* and *libm.a*), as shown below, when you link these programs.

```
% cc main.o more.o rest.o -lp -lm
```

To find the Pascal library, the link editor replaces the `-l` with *lib* and adds an *.a* after *p*. Then, it searches the */lib*, */usr/lib* and */usr/local/lib* directories for this library. For a list of the libraries that a language uses, see the associated driver manual page `cc(1)`, `f77(1)`, `pc(1)`, `cobol(1)`, or `pl1(1)` in the *User's Reference Manual*.

You may need to specify libraries when you use UNIX system packages that are not part of a particular language. Most of the manual pages for these packages list the required libraries. For example, the plotting subroutines require the libraries listed in the **plot(3X)** manual page; these libraries are specified as follows:

```
% cc main.o more.o rest.o -lp -lplot
```

To specify a library created with the archiver, type in the name of the library as shown below.

```
% cc main.o more.o rest.o libfft.a -lp
```

NOTE: The link editor searches libraries in the order you specify. Therefore, if you have a library (for example *libfft.a*) that uses data or procedures from **-lp**, you **MUST** specify *libfft.a* first.

Link Editor Options

Table 1.1 on the next pages summarizes the link editor options. Refer also to the list of general options earlier in this chapter and to the **ld(1)** manual page in the *User's Reference Manual* for complete information on options and libraries that affect link editor processing.

Link Editor Options -B	
Option Name	Purpose
-A <i>file</i>	This option produces an object that may be read into an already existing program. The argument, <i>file</i> , is the name of the file whose symbol table is used to base the definition of new symbols. Only newly linked information is entered into the text and data portions of a.out; the new symbol table reflects every symbol defined before and after the incremental load.
-b	Tells ld not to merge symbolic information entries from the same file into one entry for that file. Use this option when a file compiled for debugging has variables with the same names but different attributes. This can occur when compiling two object files that use the same include file, and variables with the same name differ because of conditional compilation statements within the file.
-B <i>num</i>	Sets the starting address of the uninitialized data segment (bss) to the hexadecimal address <i>num</i> . This option is valid only when you've also specified the -N link editor option described later in this table.
-Bstring	Append <i>string</i> to the library name created by the -lx or -klx option. The library is searched both with and without <i>string</i> .
-bestGnum	See -G on the next page.
-D <i>num</i>	Sets the starting address of the data segment (data) to the hexadecimal address <i>num</i> . This option is valid only when you've also specified the -N link editor option.
-e <i>epsym</i>	Sets the default entry point address for the output file to the specified symbol <i>epsym</i> .
-EB	Uses big-endian byte ordering when writing out header and symbol table entries.
-EL	Uses little-endian byte ordering when writing out header and symbol table entries.
-f <i>fill</i>	Sets the fill pattern for "holes" within an output section of an object file; <i>fill</i> is four-byte hexadecimal constant that defines the fill pattern.

Table 1.1 (1 of 4). Link Editor Options.

Link Editor Options	
Option Name	Purpose
-F or -z	Creates a ZMAGIC file (an object file that loads on demand) This is the default. See Chapter 10 of the <i>Assembly Language Guide</i> for more information on ZMAGIC files.
-G num	Specifies the maximum size (in decimal bytes) of a <i>.comm</i> item that should be allocated in the small uninitialized data (<i>sbss</i>) section for reference by the global pointers. The default is 8 bytes.
-bestGnum	Prints the optimum value to be specified as the <i>num</i> value for -G .
-count -nocount -countall	<p>The link editor uses these options in determining which objects are to be included or excluded in computing a value to be specified in the -bestGnum option. For example, you would exclude any object for which you did not have the source code for recompilation. The options are explained below.</p> <p>-count Objects that follow on the command line cannot be recompiled.</p> <p>-nocount Objects that follow on the command line can be recompiled.</p> <p>-countall Overrides any -nocount option appearing after it on the command line.</p> <p>See <i>Limiting the Size of Global Pointer Data</i> in Chapter 4 for examples of using the -bestGnum and related count options.</p>
-jmpopt -nojmpopt	Fill or don't fill the delay slots of jump instructions with the target of the jump and adjust the jump offset to jump past that instruction. Disabled when the -g1 , -g2 or -g flag is present. When enabled, this option can cause an out-of-memory condition in the link editor.

Table 1.1 (2 of 4). Link Editor Options.

Link Editor Options	
Option Name	Purpose
-lx	<p>Specifies the name of a link library, where <i>x</i> is the library name. The link editor searches for lib<i>x</i>.a in /lib, /usr/lib, and /usr/local/lib directories respectively. For example, if you specify curses, the library pathnames can be:</p> <pre> /lib/curses.a /usr/lib/curses.a /usr/local/lib/curses.a </pre> <p>If a library relies on procedures or data from another library, specify that library's name first.</p> <p>If a library resides in a directory other than /lib, /usr/lib,, or /usr/local/lib, use the -L option to specify the appropriate directory for that library.</p> <p>NOTE: if the byte-ordering (endian) scheme of the object module differs from that of the machine on which the link editor executes, the default libraries change. See the ld(1) manual page in the <i>User's Reference Manual</i> for more information.</p>
-L <i>dirname</i>	<p>Searches <i>dirname</i> for libraries specified in the -lx option before searching directories /lib, /usr/lib, and /usr/local/lib.</p> <p>This option must precede the -lx option.</p>
-L	<p>If the link editor doesn't find the library in <i>dirname</i>, then /lib, /usr/lib, and /usr/local/lib are NOT searched. A -L <i>dirname</i> option must be specified with -L.</p>
-m	<p>Produces a link editor memory map in System V format.</p>
-M	<p>Produces a link editor memory map in BSD format.</p>
-n	<p>Creates an NMAGIC* file. The text segment is read-only and shareable by all users of the file.</p>
-N	<p>Creates an OMAGIC* file. The text segment isn't readable and shareable by other users. The data segment follows immediately; after the text segment.</p>
<p>*See Chapter 10 of the Assembly Language Programmer's Guide for more information on NMAGIC and OMAGIC files.</p>	

Table 1.1 (3 of 4). Link Editor Options.

Link Editor Options	
Option Name	Purpose
<code>-o filename</code>	Specifies a name for your object file. If you don't specify a name, the link editor uses <i>a.out</i> as the default.
<code>-p file</code>	Preserves the symbol names listed in file when loading ucode object files. The symbol names in file are separated by blanks, tabs, or new lines. See <i>Optimizing Frequently Used Modules</i> in Chapter 4 for an example.
<code>-r</code>	Performs a partial link-edit, retaining relocation entries. This is required if the object is to be re-link edited with other objects in the future. The option causes the link editor not to define common symbols and to suppress messages on unresolved references.
<code>-s</code>	Strips symbol table information from the program object, reducing its size.
<code>-S</code>	Suppresses non-fatal error reporting.
<code>-T num</code>	Sets the origin for the text segment to the specified hexadecimal number. The default origin is 0x400000. The contents and format of the text segment are described in Chapter 10 of the <i>MIPS Assembly Language Programmer's Guide</i> .
<code>-u symname</code>	Makes <i>symname</i> undefined so that library components that define <i>symname</i> are loaded.
<code>-v</code>	Prints the name of each file as it is processed by the link editor.
<code>-V</code>	Prints the link editor version number. You might need this number, for example, when reporting a suspected bug in the link editor.
<code>-VS num</code>	Puts the specified decimal version stamp <i>num</i> in the object file that the link editor produces.
<code>-x</code>	Retains external and static symbols in the symbol table to allow some debugging facilities. Doesn't retain local (non-global) symbols.

Table 1.1 (4 of 4). Link Editor Options.

Object File Tools

The following tools provide information on object files as indicated:

- **odump**: lists the contents (including the symbol table and header information) of an object file.
- **nm**: lists only symbol table information.
- **file**: provides descriptive information on the general properties of the specified file (for example, the programming language used).
- **size**: prints the size of the `.init`, `.text`, `.rdata`, `.data`, `.sdata`, `.lit8`, `.lit4`, `.bss`, and `.sbss` sections. The format of these sections is described in **Chapter 10** of the *Assembly Language Programmer's Guide*.

The sections that follow describe these tools in detail.

Dumping Selected Parts of Files (odump)

The **odump** tool lists headers, tables, and other selected parts of an object or archive file. Figure 1.3 at the end of this section shows some examples of listings produced by **odump** and the command that produced each listing. As noted in the figure, an explanation of the information provided by **odump** can be found in **Chapters 10** and **11** of the *Assembly Language Programmer's Guide*.

Syntax:

```
odump options filename1 [filename2..filenameN]
```

In the above syntax description, *options* is one or more of the options and sub-options listed in Table 1.2; *filename* is the name of one or more object files whose contents are to be dumped. For more information, see the **odump(1)** manual page in *User's Reference Manual*.

Main odump Options	
Option Name	Purpose
-a	Dumps the archive header of each member of the specified archive library file.
-c	Dumps the string table
-f	Dumps each file header.
-F	Dumps the file descriptor table.
-g	Dumps the global symbols in the symbol table of an archive library file.
-h	Dumps the section headers.
-i	Dumps the symbolic information header.
-l	Dumps line number information.
-o	Dumps each optional header.
-P	Dumps the procedure descriptor table.
-r	Dumps relocation information.
-R	Dumps the relative file index table.
-s	Dumps the section contents.
-t	Dumps symbol table entries.
-L	Interpret and print contents of the .lib sections.

Table 1.2 (1 of 2). Odump Options.

Auxiliary odump Options	
Option Name	Purpose
-d number	Dumps the section number, or a range of section numbers, that starts at the specified <i>number</i> and that ends with the last section number or the number you specify with the +d auxiliary option.
+d number	Dumps the sections in a range that starts with the first section or with the section you specify with the -d option.
-n name	Dumps information only for the named entry name. Use this option with the -h , -s , -r , -l and -t options.
-p	Suppresses the printing of headers.
-t index	Dumps only the indexed symbol table entry. You can specify a range of table entries by using the -t option with the +t option.
+t index	Dumps symbol table entries in a range that ends with the indexed entry. The range begins with the first symbol table entry or with the section that you specify with the -t option.
-v	Dumps information in symbolic rather than numeric representation (for example, in Static rather than 0X02). Use this option with all dump options except -s .
-z name,number	Dumps the line number entry or a range of entries that start at the specified number for the named function.
+z number	Dumps the line number that starts at the function name or the number specified by the -z option and that ends at the number specified at the +z option.

Table 1.2 (2 of 2). Odump Options

```

***STRING TABLE INFORMATION***
[Offset] Name
sam.o:
[1] sam.c
[7] line
[12] string
[19] length
[26] linenumber
[37] LINETYPE
[46] main
[51] argc
[56] argv
[61] line1
[67] fd
[70] i
[72] i
[74] curlinenumber
[88] printline
[98] pline
[104] i
[107] /usr/local/mips/include/stdio.h
[139] _iobuf
[146] _cnt
[151] _ptr
[156] _base
[162] _bufsiz
[170] _flag
[176] _file
[182] _name

***FILE HEADER***
Magic Nscns Time/Date Symptr Nsyms Opthdr Flags
sam.o: 0000540 2 0x1f22b375 0x00000344 96 0x0038 0x0000

***FILE DESCRIPTOR TABLE***
filename lnOffset -----iBase/count----- merge sex
address cbLine sym line pd aux rfd language
sam.o:
sam.c 0x00000000 0 0 0 0 0 0 --- el
23 27 103 2 40 0 C
ps/include/stdio.h0x00000000 0 27 0 2 40 0 merge el
0 11 0 0 36 0 C

```

For an explanation of the contents of this listing, see **Chapters 10 and 11** of the *Assembly Language Programmer's Guide*.

Figure 1.3 (1 of 4). Example of Odump Utility Output (partial).

SECTION HEADER						% odump -h sam.o					
Name	Paddr Flags	Vaddr	Scnptr Size	Relptr Nreloc	Lnnoptr Lnno						
sam.o:											
.text	0x00000000 0x00000020	0x00000000	0x0000009c 0x000001a0	0x0000027c 25	0x00000000 0						
.sdata	0x000001a0 0x00000200	0x000001a0	0x0000023c 0x00000040	0x00000344 0	0x00000000 0						
SYMBOLIC INFORMATION HEADER						% odump -i sam.o					
vstamp	pd	fd	line	string	sym	xstring	dn	rfd	ext	aux	
sam.o:											
0x0015	2	2	103	188	38	80	0	0	12	76	
24	956	2088	932	1820	1060	2008	0	0	2232	1516	
LINE NUMBER INFORMATION						% odump -l sam.o					
Symndx/Paddr Lnno											
sam.o:											
Lines for file sam.c:											
0.	17	1.	17	2.	17						
3.	17	4.	17	5.	24						
6.	24	7.	24	8.	25						
9.	25	10.	25	11.	25						
12.	25	13.	26	14.	26						
15.	26	16.	26	17.	27						
18.	27	19.	30	20.	30						
OPTIONAL HEADER in HEX						% odump -o sam.o					
sam.o:											
0107	0015	01a0	0000	0040	0000	0000	0000	0000	0000	0000	0000
01a0	0000	01e0	0000	fff6	b301	0000	0000	0000	0000	0000	0000
0000	0000	8190	0000								
PROCEDURE DESCRIPTOR TABLE						% odump -P sam.o					
name	address	isym	iline	iopt	regmask	regoff	fpoft	fp			
sam.o:											
sam.c											
main	[0 for 2] 0x00000000	7	0	-1	0x80010000	-284	304	29			
		0	17	51	0x00000000	0		31			
printline	0x00000138	20	78	-1	0x80000000	-12	40	29			
		18	58	63	0x00000000	0		31			
/usr/local/mips/include/stdio.h[2 for 0]											
For an explanation of the contents of this listing, see Chapters 10 and 11 of the <i>Assembly Language Programmer's Guide</i> .											

Figure 1.3 (2 of 4). Example of Odump Utility Output (partial).

```

***RELOCATION INFORMATION***
Vaddr      Symndx  Type  Extern
sam.o:
.text:
0x00000034  1      0      4
0x00000038  1      0      5
0x00000040  0      4      6
0x0000003c  1      8      3
0x00000044  1      9      3
0x00000050  0      4      6
0x00000058  1      1      3
0x00000078  1      0      4
0x0000007c  1      0      5
0x00000088  0      4      6
0x00000084  1      8      3
0x0000008c  1      9      3
0x0000009c  1      5      3
0x000000ac  1     10      3
0x000000ec  0      4      6
0x000000fc  0      4      6
0x00000100  0      1      3
0x00000110  1      5      3
0x0000015c  1      0      4
0x00000160  1      0      5
0x00000168  0      4      6
0x00000170  1      8      3
0x00000178  1      0      4
0x00000180  1      0      5
0x0000017c  1     11      3

.sdata:

***RELATIVE FILE INDEX TABLE***
sam.o:
sam.c      [0 for 0]
/usr/local/mips/include/stdio.h[0 for 0]

***SECTION DATA in HEX***
sam.o:
.text:
27BD FED0 AFBF 0014 AFA4 0130 AFA5 0134 AFB0 0010 8FAE 0130
0000 0000 AFAE 0020 8FAF 0020 0000 0000 29E1 0002 1020 0007
0000 0000 3C01 0000 2424 0030 0C00 0000 2785 8010 0C00 0000
2004 0001 8FB8 0134 2785 8024 8F04 0004 0C00 0000 0000 0000
AFA2 0024 8FB9 0024 0000 0000 1720 0009 0000 0000 8FA8 0134
3C01 0000 2424 0030 8D06 0004 0C00 0000 2785 8026 0C00 0000
2004 0001 27A4 0028 8FA6 0024 0C00 0000 2005 0100 1040 001E

```

For an explanation of the contents of this listing, see **Chapters 10 and 11** of the *Assembly Language Programmer's Guide*.

Figure 1.3 (3 of 4). Example of Odump Utility Output (partial).

```

***SYMBOL TABLE INFORMATION***
[Index] Name      Value      Sclass    Symtype    Ref
sam.o:
[0]      sam.c      0x00000000 0x01      0x0b      0x001b
[1]      line      0x00000108 0x0b      0x07      0x0006
[2]      string   0x00000000 0x0b      0x09      0x000e
[3]      length   0x00000800 0x0b      0x09      0x0004
[4]      lineNumber 0x00000820 0x0b      0x09      0x0004
[5]               0x00000000 0x0b      0x08      0x0001
[6]      LINETYPE 0x00000000 0x0b      0x0a      0x0013
[7]      main     0x00000000 0x01      0x06      0x0017
[8]      argc     0x00000000 0x05      0x03      0x0004
[9]      argv     0x00000004 0x05      0x03      0x0019
[10]     [10]     0x00000014 0x01      0x07      0x0013
[11]     line1    0xffffffff 0x05      0x04      0x001a
[12]     fd       0xffffffff 0x05      0x04      0x001c
[13]     i        0xffffffff 0x05      0x04      0x0004
[14]     i        0x000000ac 0x01      0x07      0x0012
[15]     i        0xffffffff 0x05      0x04      0x0004
[16]     curlinenum 0x000001c8 0x0d      0x02      0x0004
[17]     [17]     0x00000108 0x01      0x08      0x000e
[18]     [18]     0x00000120 0x01      0x08      0x000a
[19]     main     0x00000138 0x01      0x08      0x0007
[20]     printline 0x00000138 0x01      0x06      0x0015
[21]     pline    0x00000000 0x03      0x03      0x0024
[22]     [22]     0x00000000 0x07      0x07      0x0019
[23]     i        0xffffffff 0x05      0x04      0x0004
[24]     [24]     0x0000004c 0x01      0x08      0x0016
[25]     printline 0x00000064 0x01      0x08      0x0014
[26]     sam.c    0x00000000 0x01      0x08      0x0000
[27]     /usr/local/mips/include/stdio.h 0x00000000 0x01      0x0b 0x0026
[28]     _icbuf   0x00000018 0x0b      0x07      0x0025
[29]     _cnt     0x00000000 0x0b      0x09      0x002c
[30]     _ptr     0x00000020 0x0b      0x09      0x0036
[31]     _base    0x00000040 0x0b      0x09      0x0037
[32]     _bufsiz  0x00000060 0x0b      0x09      0x002c
[33]     _flag    0x00000080 0x0b      0x09      0x002b
[34]     _file    0x00000090 0x0b      0x09      0x0030
[35]     _name    0x000000a0 0x0b      0x09      0x0038
[36]     [36]     0x00000000 0x0b      0x08      0x001c
[37]     /usr/local/mips/include/stdio.h 0x00000000 0x01      0x08 0x001b
[38]     iob      0x000001e0 0x15      0x01      0x0039
[39]     fopen    0x00000000 0x06      0x06      0x003f
[40]     fdopen   0x00000000 0x00      0x06      0x0035
[41]     freopen  0x00000000 0x00      0x06      0x0038
[42]     ftell    0x00000000 0x00      0x06      0x003b
[43]     fgets    0x00000000 0x06      0x06      0x004a
[44]     printline 0x00000138 0x01      0x06      0x0014
[45]     main     0x00000000 0x01      0x06      0x0007
[46]     fprintf  0x00000000 0x06      0x06      0x001e
[47]     exit     0x00000000 0x06      0x06      0x0020
[48]     strlen   0x00000000 0x06      0x06      0x0022
[49]     fflush   0x00000000 0x06      0x06      0x0026

```

For an explanation of the contents of this listing, see Chapters 10 and 11 of the *Assembly Language Programmer's Guide*.

Figure 1.3 (4 of 4). Example of Odump Utility Output (partial).

Listing Symbol Table Information (nm)

The **nm** tool prints symbol table information for object files and archive files.

Syntax:

```
nm options filename1 [filename2..filenameN]
```

In the above syntax description, *options* is one or more of characters (listed in Table 1.4) that specify the type of information to be printed; *filename* specifies the object file(s) or archive file(s) from which symbol table information is to be extracted. If you don't specify a file, **nm** assumes *a.out*.

Below is an example of an **nm** statement and the listing it produces. Note that each item in the listing has a key that describes its storage class. The keys are explained in Table 1.3.

Examples:

```
%nm a.out
00004608 S Argc
0000460c S Argv
00004490 d blanks
00004700 b bufendtab
00003330 T cerror
00000cd4 T cleanup
000044e8 D ctype
00001fa0 T doprnt
00000de4 T exit
00001878 T filbuf
00000990 T filbuf
0000c560 N gp
00004228 D iob
00004598 G lastbuf
00001f44 t lowdigit
%
```

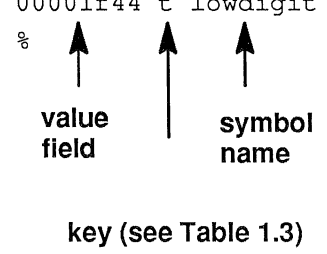


Figure 1.4. Symbol Table in BSD Format (option -B)

The meanings of the character keys shown in an **nm** listing are described below.

Key	Description
N	Nil storage class, which avoids loading of unused external references.
T	External text.
t	Local text.
D	External initialized data.
d	Local initialized data.
B	External zeroed data.
b	Local zeroed data.
A	External absolute data.
a	Local absolute data.
U	External undefined data.
G	External small initialized data.
S	External small zeroed data.
s	Local small zeroed data.
R	External read-only data.
r	Local read-only data.
C	Common data.
E	Small common data.
V	External small common data.

Table 1.3. nm Character Key Meanings.

Symbols from sam.o:							
Name	Value ¹	Class ¹	Type	Size	Indx	Section	
sam.c	00000000	File	ref=27			0	Text
line	00000264	Block	ref=6			1	Info
string	00000000	Member	unsigned char [256]			2	Info
length	00002048	Member	int			3	Info
linenumber	00002080	Member	int			4	Info
	00000000	End	ref=1			5	Info
LINETYPE	00000000	Typdef	struct line			6	Info
main	00000000	Proc	end=20 int			7	Text
argc	00000000	Param	int			8	Abs
argv	00000004	Param	unsigned char **			9	Abs
	00000020	Block	ref=19			10	Text
line1	-0000264	Local	struct line			11	Abs
fd	-0000268	Local	struct _iobuf*			12	Abs
i	-0000272	Local	int			13	Abs
	00000172	Block	ref=18			14	Text
i	-0000280	Local	int			15	Abs
curlinenumber	00000456	Static	int			16	SData
	00000264	End	ref=14			17	Text
	00000288	End	ref=10			18	Text
main	00000312	End	ref=7			19	Text
printline	00000312	Proc	end=26 btNil			20	Text
pline	00000000	Param	struct line*			21	Abs
	00000012	Block	ref=25			22	Text
i	-0000004	Local	int			23	Abs
	00000076	End	ref=22			24	Text
printline	00000100	End	ref=20			25	Text
sam.c	00000000	End	ref=0			26	Text
/usr/local/mips/incl	00000000	File	ref=38			27	Text
_iobuf	00000024	Block	ref=37			28	Info
_cnt	00000000	Member	int			29	Info
_ptr	00000032	Member	unsigned char *			30	Info
_base	00000064	Member	unsigned char *			31	Info
_bufsiz	00000096	Member	int			32	Info
_flag	00000128	Member	short			33	Info
_file	00000144	Member	unsigned char			34	Info
_name	00000160	Member	unsigned char *			35	Info
	00000000	End	ref=28			36	Info

¹For information on these fields, see **Chapter 11** in the *Assembly Language Programmer's Guide*.

Figure 1.5. Symbol Table in System V Format (option -A)..

nm Options	
Option Name	Purpose
-A	Prints the listing in System V format. The default format is that of your operating system.
-B	Prints the listing in BSD format. The default format is that of your operating system.
-a	Prints debugging information (turns BSD output into System V format).
-b	Prints the value field in octal.
-d	Prints the value field in decimal (the default for System V output).
-e	Prints only external and static variables.
-h	Suppresses printing of headers.
-n	Sorts external symbols by name for System V format. Sorts all symbols by value for Berkeley format (by name is the BSD default output).
-o	Prints value field in octal (System V output). Prints the filename immediately before each symbol name (BSD output).
-p	Lists symbols in the order they appear in the Symbol table.
-r	Reverses the sort that you specified for external symbols with the <code>-n</code> and <code>-v</code> options.
-T	Truncates characters in exceedingly long symbol names; inserts an asterisk as the last character of the truncated name. This may make the listing easier to read.
-u	Prints only undefined symbols.
-v	Sorts external symbols by value (default for Berkeley format).
-V	Prints the version number of <code>nm</code> .
-x	Prints the value field in hexadecimal.

Table 1.4 Symbol Table Dump (*nm*) Options.

Determining a File's Type (file)

The **file** tool lists the properties of program source, text, object, and other files.

This tool often erroneously recognizes command files as C programs. It does not recognize Pascal or LISP programs. For more information, see the **file(1)** manual page in *User's Reference Manual*.

Syntax:

```
file filename1 [filename2...filenameN]
```

Example:

```
% file test.o a.out
test.o:mipsel demand paged pure executable not stripped
a.out: mipsel demand paged pure executable not stripped
%
```

Determining a File's Section Sizes (size)

The **size** tool prints information about the *text*, *rdata*, *data*, *sdata*, *bss*, and *sbss* sections of the specified object or archive file(s). The contents and format of section data are described in **Chapter 10** of the *Assembly Language Programmer's Guide*.

Syntax:

```
size options filename1 [filename2..filenameN]
```

In the above syntax description, *options* is in alphabetic character (listed in Table 1.5) that specifies the format of the listing; *filename* specifies the object or archive file(s) whose properties are to be listed. If you don't specify a file, **size** assumes *a.out*.

Below is an example of a **size** statement and the listing it produces.

Example:

```
% size test.o
text data bss    dec    hex
16384 4096 164437 184917 2d255
```

Size Options	
Option Name	Purpose
-A	Prints data section headers in System V format. The default format is determined by the UNIX version running at your installation.
-B	Prints data section headers in Berkeley format. The default format is determined by the UNIX version running at your installation.
-d	Prints the section sizes in decimal.
-o	Prints the section sizes in octal.
-V	Prints the version of size that you are using.
-x	Prints the section sizes in hexadecimal.

Table 1.5. Size Options.

Archiver

An archive library is a file that contains one or more routines in object (.o) file format; the term *object* as used in this chapter refers to an .o file that is part of an archive library file. When a program calls an object not explicitly included in the program, the link editor (**ld**) looks for that object in an archive library. The editor then loads only that object (not the whole library) and links it with the calling program.

The archiver (**ar**) creates and maintains archive libraries and has the following main functions:

- Copying new objects into the library
- Replacing existing objects in the library
- Moving objects about the library
- Copying individual objects from the library into individual object file.

The sections that follow describe the syntax of the **ar** (archiver) command and some examples of how to use it. See the **ar(1)** manual page in the *UNIX User's Reference Manual* for additional information.

Syntax:

```
ar options [posObject] libName [object1...objectN]
```

The following explains the parameters in the above syntax description:

- *options* is one or more characters (listed in Tables 1.6 and 1.7) that specify the action that the archiver is to take. When you specify more than one option character, group the characters together with no spaces between; don't place a dash (-) character before the option characters.
- *posObject* is the name of an object within an archive library. It specifies the relative placement (either before or after *posObject*) of an object that is to be copied into the library or moved within the library. A *posObject* is required when the **m** or **r** options are specified together with the **a**, **b**, or **i** suboptions. Example 4 below shows the use of a *posObject* parameter.
- *libName* is the name of the archive library you are creating, updating, or extracting information from.
- *object* is the name object(s) or object file(s) that you are manipulating.

Examples

1. Create a new library and add routines to it.

```
% ar cr libtest.a mcount.o mon1.o string.o
```

Options **c** suppresses archiver messages during the creation process. Options **r** creates the library *libtest.a* and adds *mcount.o*, *mon1.o*, and *string.o*.

2. Add or replace an object (*.o*) file to an existing library.

```
% ar r libtest.a mon1.o
```

Option **r** replaces *mon1.o* in the library *libtest.a*. If *mon1.o* didn't already exist, the new object *mon1.o* would be added.

CAUTION: If you specify the same file twice in an argument list, it appears twice in the archive.

3. Update the library's *symdef* table.

```
% ar ts libtest.a
```

Option **s** creates the *symdef* table and **t** lists the table of contents.

NOTE: After you create or change a library, you must always use the **s** option to update the *symdef* (symbol definition) table of the archive library. The link editor uses the *symdef* table to locate objects during the link process.

4. Add a new file immediately before a specified file in the library.

```
% ar rb mcount.o libtest.a new.o
```

↑
posObject

Option **r** adds *new.o* in the library *libtest.a*. Option **b** followed by posObject *mcount.o* causes the archiver to place *new.o* immediately before *mcount.o*.

Archiver Options

The table below lists the archiver options. You must specify at least one and *only* one of the following options: **d**, **m**, **p**, **q**, **r**, or **x**. In addition, you can optionally specify the **c**, **l**, **s**, **t**, and **v** options, and any of the archiver suboptions listed in the following tables.

Archiver Options (Part 1)	
Option Name	Purpose
c	Suppresses the warning message that the archiver issues when it discovers that the archive you specified doesn't already exist.
d	Deletes the specified objects from the archive.
l	Puts the archiver's temporary files in the current working directory. Ordinarily the archiver puts those files in /tmp. This option is useful when /tmp is full.
m	Moves the specified files to the end of the archive. If you want to move the object to a specific position in the archive library, specify an a , b or i suboption together with the <i>posObject</i> parameter.
p	Prints the specified object(s) in the archive on the standard output device (usually the terminal screen).
q	Adds the specified object files to the end of the archive. An existing object file with the same name is <i>not</i> deleted, and the link editor will continue to use the old file. This option is similar to the r option (described below) but it is faster. Use it when creating a new library.
r	Adds the specified object files to the archive. This option deletes duplicate objects in the archive. If you want to add the object at a specific position in the archive library, specify an a , b , or i suboption together with the <i>posObject</i> parameter. See Example 4 in the preceding section for an example of using the <i>posObject</i> parameter. See also the u suboption Use the r option when updating existing libraries.

Table 1.6 (1 of 2). Archiver Options.

Archiver Options (Part 2)	
Option Name	Purpose
s	<p>Creates a <i>symdef</i> file in the archive. You must use this option each time you create or change the archive library.</p> <p>If all objects don't have the same endian byte ordering scheme, the archiver issues an error message and doesn't create a <i>symdef</i> table. At least one of the following options must be specified with the <i>s</i> option: m, p, q, r, or t.</p>
t	<p>Prints a table of contents on the standard output (usually the screen) for the specified object or archive file.</p>
v	<p>Lists descriptive information during the process of creating or modifying the archive. When specified with the <i>t</i> option, produces a verbose table of contents.</p>
x	<p>Copies the specified objects from the archive and places them in the current directory. Duplicate files are overwritten. The <i>last modified</i> date is the current date, unless you specify the <i>o</i> suboption. Then the date stamp on the archive file is the last modified.</p> <p>If no objects are specified, copies all the library objects into the current directory.</p>

Table 1.6 (2 of 2). Archiver Options.

The archiver has these suboptions:

Archiver Suboptions		
Suboption Name	Use with...	Purpose
a	m or r	Specifies that the object file follow the <i>posObject</i> file you specify in the ar statement. ¹
b	m or r	Specifies that the object file precede the <i>posObject</i> file you specify in the ar statement. ¹
i	m or r	Same as -b . ¹
o	x	Used when extracting a file from the archive to the current directory. Forces the last modified date of the extracted file to match that of the archive file.
u	r	The archiver replaces the existing object file when the last modified data is earlier (precedes) that of the new object file.
¹		See example 4 in the Examples section preceding this section for an example of the <i>posObject</i> parameter.

Table 1.7. Archiver Options.

This chapter describes the alignment, size, and value ranges for the C and Pascal languages, and how the compiler groups these records in storage.

C Language

This section describes how the compiler maps C variables into storage and contains the following topics:

- Alignment, Size, and Value Ranges
- C Arrays, Structures, and Unions
- Storage Classes

Alignment, Size, and Value Ranges

Table 2.1 describes how the C compiler implements size, alignment, and value ranges for the data types.

Type	Size	Alignment	Value Range	
			Signed	Unsigned
int	32 bits	Word ¹	-2^{31} to $2^{31}-1$	0 to $2^{32}-1$
long	32 bits	Word ¹	-2^{31} to $2^{31}-1$	
enum	32 bits	Word ¹	-2^{31} to $2^{31}-1$	
short	16 bits	Halfword ²	-32,768..32,767	0..65,535
char ⁴	8 bits	Byte	-128..127	0..255
float ⁵	32 bits	Word ¹	See note.	
double ⁶	64 bits	Doubleword ³	See note.	
pointer	32 bit	Word ¹		0 to $2^{32}-1$

¹ Byte boundary divisible by four.
² Byte boundary divisible by two.
³ Byte boundary divisible by eight.
⁴ char is assumed to be unsigned, unless the signed attribute is used.
⁵ IEEE single precision. See note following this table for valid ranges.
⁶ IEEE double precision. See note following this table for valid ranges.

Table 2.1. Size, Alignment, and Value Ranges for C Data Types.

NOTE: Approximate valid ranges for float and double are:

	Maximum Value
float	$3.40282356 \times 10^{38}$
double	$1.7976931348623158 \times 10^{308}$

	Minimum Values	
	Denormalized	Normalized
float	$1.40129846 \times 10^{-46}$	$1.17549429 \times 10^{-38}$
double	$4.9406564584124654 \times 10^{-324}$	$2.2250738585072012 \times 10^{-308}$

For characters to be treated as signed, either use the compiler option `—signed`, or use the keyword **signed** in conjunction with **char**, as shown in the following example:

```
signed char c;
```

The header files `limits.h` and `float.h` (usually found in `/usr/include`) contain C macros that define minimum and maximum values for the various data types. Refer to these files for the macro names and values.

The following sections describe how the data types shown in Table 2.1 affect arrays, structures, and unions.

C Arrays, Structures, and Unions

Arrays. Arrays have the same boundary requirements as the data type specified for the array. The size of an array is the size of the data type multiplied by the number of elements. For example, for the following declaration:

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

the size of the resulting array is 48 ($2 \times 3 \times 8$, where 8 is the size of the **double** floating point type).

Structures. Each member of a structure begins at an offset from the structure base. The offset corresponds to the order in which a member is declared; the first member is at offset 0.

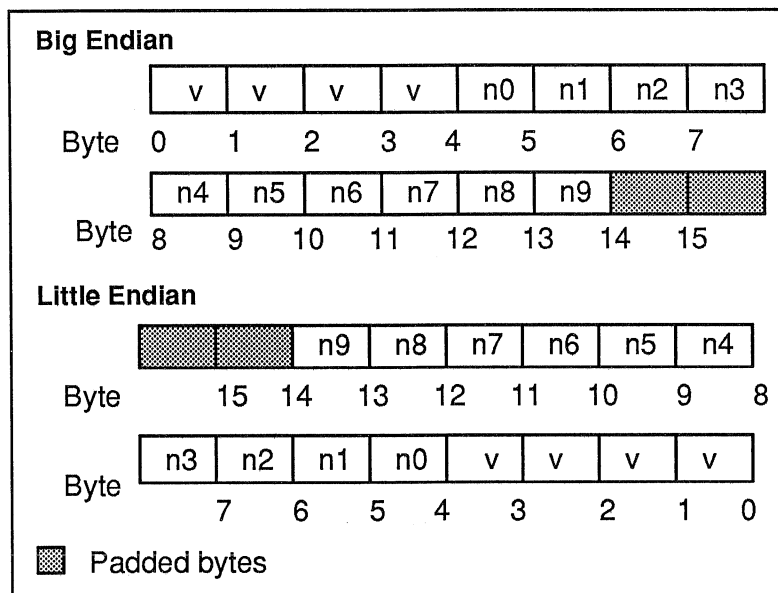
The size of a structure in the object file is the size of its combined members plus padding added, where necessary, by the compiler. The following rules apply to structures:

- Structures must align on the same boundary as that required by the member with the most restrictive boundary requirement. The boundary requirements by degree of restrictiveness are: byte, halfword, word, and doubleword, with doubleword being the most restrictive.
- The compiler terminates the structure on the same alignment boundary on which it begins. For example, if a structure begins on an even-byte boundary, it also ends on an even-byte boundary.

For example, the following structure:

```
struct s {
    int w;
    char n[10];
};
```

is mapped out in storage as follows:



See **Appendix C** for more information on big and little endian byte ordering.

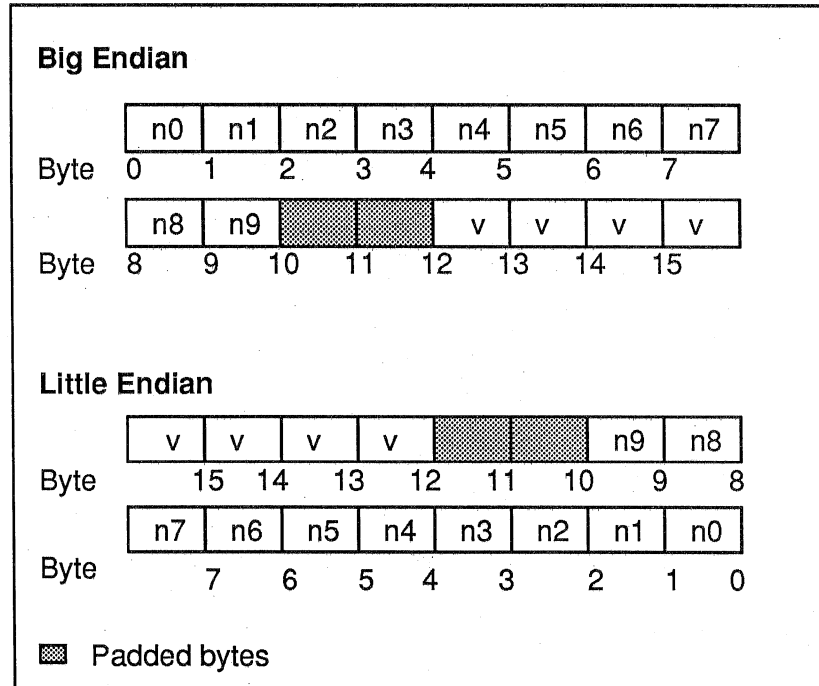
Note that the length of the structure is 16 bytes, even though the byte count as defined by the *int* *v* and the *char* *n* components is only 14. Because *int* has a stricter boundary requirement (word boundary) than *char* (byte boundary), the structure must end on a word boundary (a byte offset divisible by four). The compiler therefore adds two bytes of padding to meet this requirement.

An array of data structures illustrates the reason for this requirement. For example, if the above structure were the element-type of an array, some of the *int* *v* components wouldn't be aligned properly without the two-byte pad.

Alignment requirements may cause padding to appear in the middle of a structure. For example, by rearranging the structure in the last example to the following:

```
struct s {
    char n[10]
    int w;
}
```

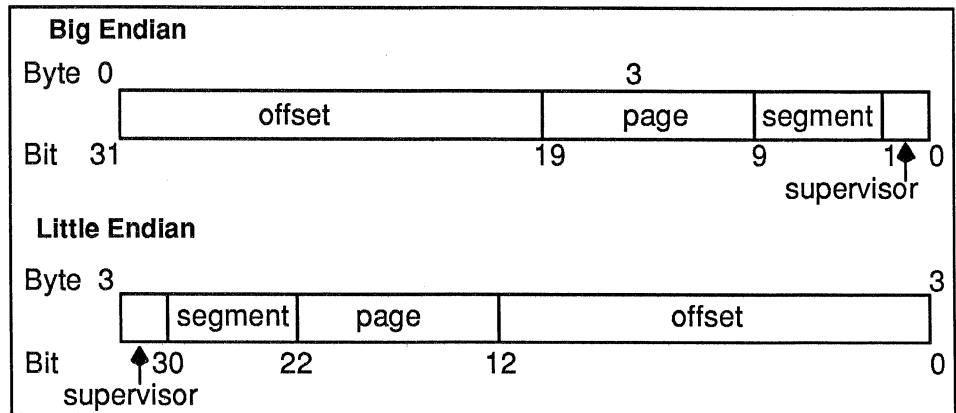
the compiler maps the structures as follows:



Note that the size of the structure remains 16 bytes, but two bytes of padding follow the *n* component to align *v* on a word boundary. See **Appendix C** for more information on big and little endian byte ordering.

Bit fields are packed from the most significant bit to least significant bit in a word and can be no longer than 32 bits; bit fields can be signed or unsigned. The following structure:

```
typedef struct {
    unsigned offset      :12;
    unsigned page       :10;
    unsigned segment    : 9;
    unsigned supervisor  : 1;
} virtual_address;
```



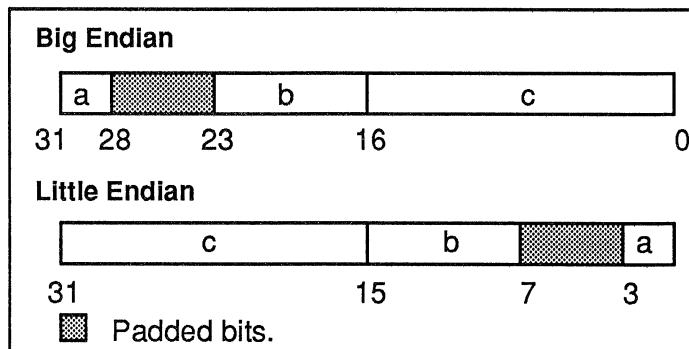
is mapped out as follows:

The compiler moves fields that overlap a word boundary to the next word. See **Appendix C** for more information on big and little endian byte ordering.

The compiler aligns a nonbit field that follows a bit-field declaration to the next boundary appropriate for its type. For example, the following structure:

```
struct {
    unsigned      a :3;
    char          b;
    short         c;
} x;
```

is mapped out as follows:



Note that five bits of padding are added after *unsigned a* so that *char b* aligns on a byte boundary, as required. See **Appendix C** for more information on big and little endian byte ordering.

Unions. A union must align on the same boundary as the member with the most restrictive boundary requirement. The boundary requirements by increasing degree of restrictiveness are: byte, halfword, word, and doubleword. For example, a union containing **char**, **int**, and **double** data types must align on a doubleword boundary, as required by the **double** data type.

Storage Classes

Auto. An **auto** declaration indicates that storage is allocated at execution and exists only for the duration of that block activation.

Static. The compiler allocates storage for a **static** declaration at compile time. This allocation remains fixed for the duration of the program. Static variables reside in the program bss section if they are not initialized, otherwise they are placed in the data section.

Register. The compiler allocates variables with the **register** storage class to registers. For programs compiled using the **—O** (optimize) option, the optimization phase of the compiler tries to assign all variables to registers, regardless of the storage class specified.

Extern. The **extern** storage class indicates that the variable refers to storage defined elsewhere in an external data definition. The compiler doesn't allocate storage to **extern** variable declarations; it uses the following logic in defining and referencing them:

Extern is omitted. If an initializer is present, a definition for the symbol is emitted. Having two or more such definitions among all the files comprising a program results in an error at link time or before. If no initializer is present, a common definition is emitted. Any number of common definitions of the same identifier may coexist.

Extern is present. The compiler assumes that declaration refers to a name defined elsewhere. A declaration having an initializer is illegal. If a declared identifier is never used, the compiler doesn't issue an external reference to the linker.

Volatile. The **volatile** storage class is specified for those variables that may be modified in ways unknown to the compiler. For example, **volatile** might be specified for an object corresponding to a memory mapped input/output port or an object accessed by an asynchronously interrupting function. Except for expression evaluation, no phase of the compiler optimizes any of the code dealing with **volatile** objects.

NOTE. If a pointer specified as **volatile** is assigned to another pointer without the volatile specification, the compiler treats the other pointer as non-volatile. In the following example:

```
volatile int *i;
int *j;
.
.
(volatile*) j = i;
3108282356*10
```

the compiler treats *j* as a non-volatile pointer and the object it points to as non-volatile, and may optimize it.

The compiler option **—volatile** causes all objects to be compiled as volatile.

Pascal

Alignment, Size, and Value Ranges

This section describes how the Pascal compiler implements size, alignment, and value ranges for the various data types.

Table 2.2 shows the value ranges for the Pascal scalar types; Tables 2.3, 2.4, and 2.5, which start on the next page, show the size and alignment for the various scalar types.

Scalar Type	Value Ranges
boolean	0 or 1
char	0..127
integer	$-2^{31} .. 2^{31} - 1$
cardinal	$0 .. 2^{32} - 1$
real	See note 1.
double	See note 1.

Table 2.2. Pascal Value Ranges.

NOTE 1: Approximate valid ranges for real and double are:

	Maximum Value
real	$3.40282356 * 10^{38}$
double	$1.7976931348623158 * 10^{308}$

	Minimum Values	
	Denormalized	Normalized
real	$1.40129846 * 10^{-46}$	$1.17549429 * 10^{-38}$
double	$4.9406564584124654 * 10^{-324}$	$2.2250738585072012 * 10^{-308}$

NOTE 2: Enumerated types with n elements are treated the same as the integer subrange $0..n-1$.

	Unpacked Records or Arrays *	
Type	Size	Alignment
boolean	8	byte
char	8	byte
integer	32	word
cardinal	32	word
pointer	32	word
file	32	word
real	32	word
double	64	doubleword
subrange of		
0..255 or -128..127	8	byte
0..65535 or -32768..32767	16	halfword
$0..2^{32}-1$ $-2^{31}..-2^{31}-1$	32	word
set of char set of char subrange	128	word
set of a..b	see NOTE	word
* Variables or fields.		

Table 2.3. Size and Alignment of Pascal Unpacked Records or Arrays (Variables or Fields).

NOTE: The compiler uses the following formula for determining the size of the set of a..b:

$$\text{size} = \lfloor b/32 \rfloor - \lfloor a/32 \rfloor + 1 \text{ words}$$

(The notation $\lfloor x \rfloor$ indicates the floor of x ; i.e., the largest integer not greater than x .)

See the section **Rules for Set Sizes** at the end of this chapter for rules on specifying the upper and lower bounds of sets.

Scalar Type	Packed Arrays	
	Size	Alignment
boolean	8	byte
char	8	byte
integer	32	word
cardinal	32	word
pointer	32	word
file	32	word
real	32	word
double	64	doubleword
subrange of		
0..1 or -1..0	1	bit
0..3 or -2..1	2	2-bit
0..15 or -8..7	4	4-bit
0..255 or -128..127	8	byte
0..65535 or -32768..32767	16	halfword
$0..2^{32}-1$ $-2^{31}..2^{31}-1$	32	word
set of char set of char subrange	128	word
set of a..b	See NOTE	

Table 2.4. Size and Alignment of Pascal Packed Arrays.

NOTE: The set of $a..b$ is aligned on an n -bit boundary where n is computed as follows:

$$n = \lceil \log(\text{size}) \rceil$$

For example, the set of 0..2 has a size of 3 bits and will align on a 4-bit boundary.

(The notation $\lceil x \rceil$ indicates the ceiling of x ; i.e., the smallest integer not less than x .)

The compiler uses the minimum number of bits possible in creating the *set of a..b* field. The following formula is used:

$$\begin{aligned} &\text{if } b - \lfloor a/32 \rfloor * 32 + 1 \leq 32 \text{ then} \\ &\text{size} = b - \lfloor a/32 \rfloor * 32 + 1 \text{ bits} \\ &\text{else} \\ &\text{size} = (\lfloor b/32 \rfloor - \lfloor a/32 \rfloor + 1) * 32 \text{ bits} \end{aligned}$$

See the section **Rules for Set Sizes** at the end of this chapter for rules on specifying the upper and lower bounds of sets.

Packed Records		
Scalar Type	Size	Alignment
boolean	1	bit
char	8	bit
integer	32	word
cardinal	32	word
pointer	32	word
file	32	word
real	32	word
double	64	doubleword
subrange of	See Note.	bit/word*

Table 2.5. Size and Alignment of Pascal Packed Records.

NOTE: The compiler uses the minimum number of bits possible in creating a subrange field in a packed record. For the *subrange of a..b*:

$$\begin{aligned} &\text{If } a \geq 0 \text{ then size} = \lceil \log_2 (b+1) \rceil \text{ bits} \\ &\text{If } a < 0 \text{ then size} = \max(\lceil \log_2 (b+1) \rceil, \lceil \log_2 (-a) \rceil) + 1 \text{ bits} \end{aligned}$$

To avoid crossing a word boundary, the compiler moves data types aligned to bit boundaries in a packed record, to the next word.

Pascal Arrays, Records and Variant Records

Arrays. Arrays have the same boundary requirements as the data type specified for element of the array. The size of an array is the size of the data type multiplied by the number of elements. For example, for the following declaration:

```
x: array [1..2, 1..3] of double;
```

the size of the resulting array is 48 bytes (2*3*8, where 8 is the size of the **double** floating point type in bytes).

Records. Each member of a record begins at an offset from the structure base. The offset corresponds to the order in which a member is declared; the first member is at offset 0.

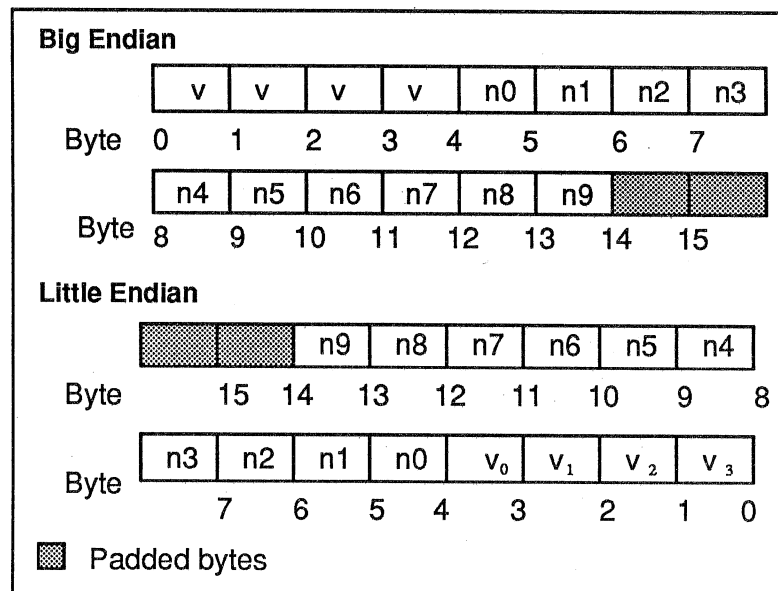
The size of a record in the object file is the size of its combined members plus padding added, where necessary, by the compiler. The following rules apply to records:

- Records must align on the same boundary as that required by the member with the most restrictive boundary requirement. The boundary requirements by degree of restrictiveness are: byte, halfword, word, and doubleword, with doubleword being the most restrictive.
- The compiler terminates the record on the same alignment boundary on which it begins. For example, if a record begins on an even-byte boundary, it also ends on an even-byte boundary (i.e., the size is a multiple of the alignment).

For example, the following structure:

```
type S=record
  v:integer
  n:array[1..10] of char
end;
```

is mapped out in storage as follows:



See **Appendix C** for more information on big and little endian byte ordering.

Note that the length of the structure is 16 bytes, even though the byte count as defined by the *v:integer* and the *n:array[1..10] of char* components is only 14. Because *integer* has a stricter boundary requirement (word boundary) than *char* (byte boundary), the structure must end on a word boundary (a byte offset divis-

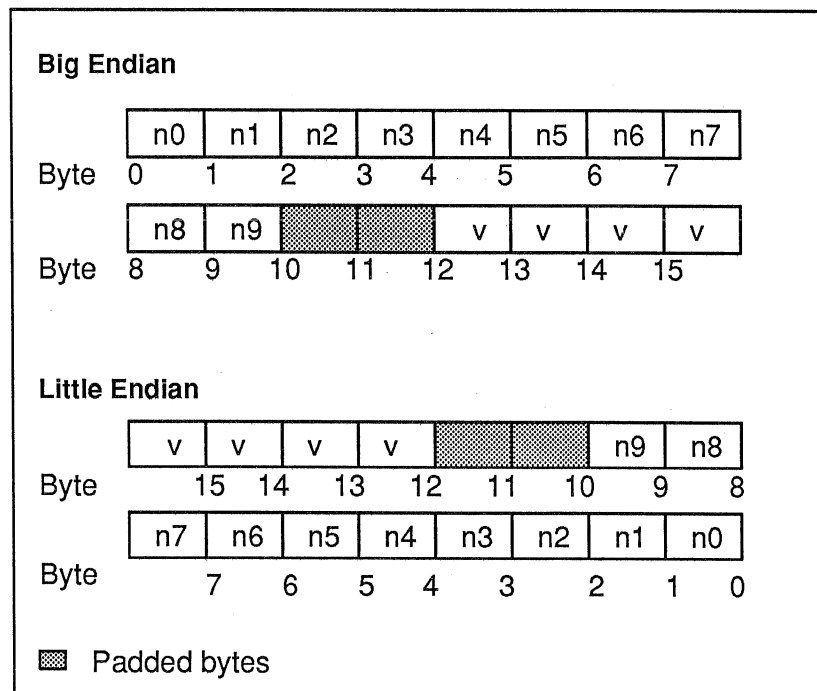
ible by four). The compiler therefore adds two bytes of padding to meet this requirement.

An array of data structures illustrates the reason for this requirement. For example, if the above structure were the element-type of an array, some of the v :integer components wouldn't be aligned properly without the two-byte pad.

Alignment requirements may cause padding to appear in the middle of a structure. For example, by rearranging the structure in the last example to the following:

```
type S=record
    n:array [1..10] of char;
    v:integer
end;
```

the compiler maps the structures as follows:



Note that the size of the structure remains 16 bytes, but two bytes of padding follow the n component to align v on a word boundary.

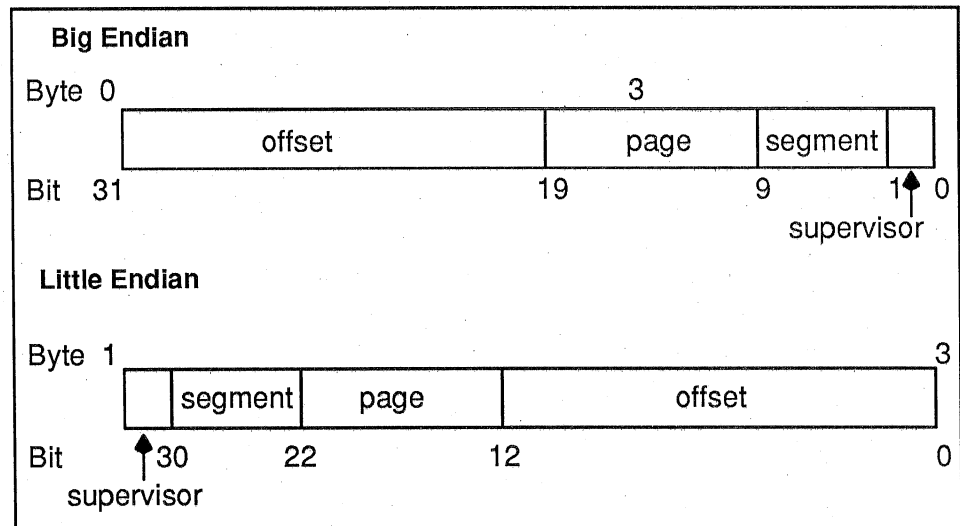
Ranges in a packed record are packed from the most significant bit to least significant bit in a word. The following packed record:

```

type virtual_address=packed record
  offset:          0..4095; (* 12 bits *)
  page:           0..1023; (* 10 bits *)
  segment:       0..511;  (* 9 bits *)
  supervisor:    0..1;    (* 1 bit *)
end;

```

is mapped out as follows:



See **Appendix C** for more information on big and little endian byte ordering.

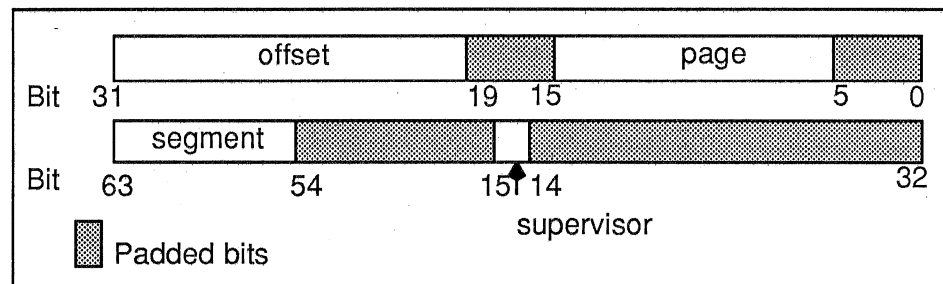
Ranges in an unpacked record are packed from the most significant bit to the least significant bit but each range is aligned to the appropriate boundary as indicated in table 2.4. The following unpacked record:

```

type virtual_address = record
  offset:          0..4095; (* 12 bits *)
  page:           0..1023; (* 10 bits *)
  segment:       0..511;  (* 9 bits *)
  supervisor:    0..1;    (* 1 bit *)
end;

```

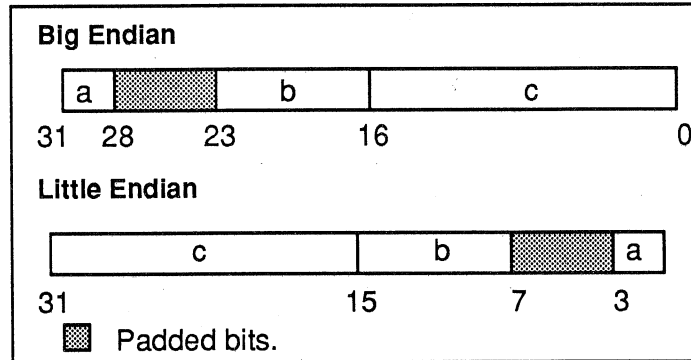
is mapped out as follows (big-endian):



For unpacked records, the compiler aligns a non-range element that follows a range declaration to the next boundary appropriate for its type. For example, the following structure:

```
var x: record
  a: 0..7; (* 3 bits packed *)
  b: char; (* 8 bits *)
  c: -32768..32767; (* 16 bits *)
end;
```

is mapped out as follows:



Note that five bits of padding are added after *a* so that *b* aligns on a byte boundary, as required. (See **Appendix C** for more information on big and little endian byte ordering.)

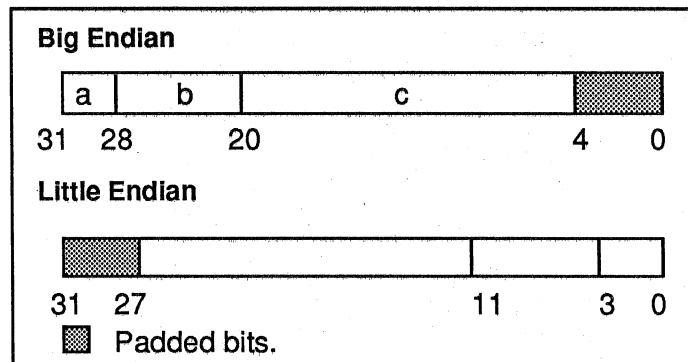
Variant Records. A variant record must align on the same boundary as the member with the most restrictive boundary requirement. The boundary requirements by increasing degree of restrictiveness are: byte, halfword, word, and doubleword. For example, a variant record containing **char**, **integer**, and **double** data types must align on a doubleword boundary, as required by the **double** data type.

For a packed record, booleans, chars, and ranges are bit-aligned, and all other types are word or double-word aligned as is appropriate for the type (see table 2.4).

The previous record done as a packed record:

```
var x: packed record
  a: 0..7; (*3 bits *)
  b: char; (* 3 bits *)
  c: -32768..32767; (* 16 bits *)
end;
```

is mapped out as follows:



Rules for Set Sizes

The maximum number of elements permitted in a set ranges between 481 and 512. This variance is due to the way Pascal implements sets. For efficient accessing of set elements, Pascal expects the lower-bound of a set to be a multiple of 32. If for the set specified:

set of $a..b$

a is not a multiple of 32, Pascal "adds" elements to the set from a down to the next multiple of 32 less than a . For example, the set:

set of $5..31$

would have internal padding elements 0..4 added. These padding elements are inaccessible to the program. This implementation sacrifices some space for a fast, consistent method of accessing set elements.

the padding required to pad the lower bound down to a multiple of 32 varies between 0 and 31 elements.

For the *set of $a..b$* to be a valid set in Pascal, the following condition must be met:

$$size = (b - 32 \lfloor a/32 \rfloor + 1) \leq 512$$

The table below shows some example sets and whether they are valid by the above equation.

Specification	Lower	Upper	Set size	Valid Size
set of 1..511	0*	511	512	Yes
set of 0..511	0	511	512	Yes
set of 1..512	0*	512	513	No
set of 31..512	0*	512	513	No
set of 32..512	32	512	481	Yes
set of 32..543	32	543	512	Yes

*As padded down to by Pascal.

This chapter describes the coding interfaces between C and Pascal; it gives rules and examples for calling and passing arguments among these languages.

You may need to refer to **Chapter 2** for detailed information on how the variables of the various languages appear in storage. For information on interfaces between FORTRAN programs and programs written in C or Pascal, refer to the *Part I of the FORTRAN Programmer's Guide and Language Reference* manual.

Pascal/C Interface

General Considerations

In general, calling C from Pascal and Pascal from C is fairly simple. Most data types have natural counterparts in the other language. However, differences do exist in the following areas:

- single-precision floating point, procedure, and function parameters
- Pascal by-value arrays
- file variables
- passing string data between C and Pascal
- passing variable arguments

These differences are discussed in the following sections.

Single-precision floating point. In function calls, C automatically converts single-precision floating point values to double precision, whereas Pascal passes single-precision floating by-value arguments directly. Follow these guidelines when you wish to pass double-precision values between C and Pascal routines:

- If possible, write the Pascal routine so that it receives and returns double-precision values, or
- If the Pascal routine cannot receive a double-precision value, write a Pascal routine to accept double-precision values from C, then have that routine call the single-precision Pascal routine.

There is no problem passing single-precision values by reference between C and Pascal.

Procedure and function parameters. C function variables and parameters consist of a single pointer to machine code, whereas Pascal procedure and function parameters consist of a pointer to machine code, and a pointer to the stack

frame of the lexical parent of the function. Such values can be declared as structures in C. To create such a structure, put the C function pointer in the first word, and 0 in the second. C functions cannot be nested, and thus have no lexical parent; therefore, the second word is irrelevant.

You cannot call a C routine with a function parameter from Pascal.

Pascal by-value arrays. C never passes arrays by value. In C, an array is actually a sort of pointer, and so passing an array actually passes its address, which corresponds to Pascal by-reference (VAR) array passing. In practice this is not a serious problem because passing Pascal arrays by value is not very efficient, and so most Pascal array parameters are VAR anyway. When it is necessary to call a Pascal routine with a by-value array parameter from C, pass a C structure containing the corresponding array declaration.

File variables. The Pascal text type and the C stdio package's FILE* are compatible. However, Pascal passes file variables only by reference; a Pascal routine cannot pass a file variable by value to a C routine. C routines that pass files to Pascal routines should pass the address of the FILE* variable, as with any reference parameter.

Strings. C and Pascal programs handle strings differently. In Pascal, a string is defined to be a packed array of characters, where the lower bound of the array is 1, and the upper bound is some integer greater than 1. For example:

```
var s: packed array[1..100] of char;
```

where the upperbound (100 in this case) is large enough to efficiently handle most processing requirements. This differs from the C style of indexing arrays from 0 to MAX-1. In passing an array, Pascal passes the entire array as specified, padding to the end of the array with spaces.

Most C programs treat strings as pointers to a single character and use pointer arithmetic to step through the string. A null character ($\backslash 0$ in C) terminates a string in C; therefore, when passing a string from Pascal to C, always terminate the string with a null character (chr(0) in Pascal).

The following example shows a Pascal routine that calls the C routine `atoi` and passes the string `s`. Note that the routine ensures that the string terminates with a null character.

```

type
  astrindex = 1 .. 20;
  astring = packed array [astrindex] of char;

  function atoi(var c: astring): integer; external;

program ptest(output);
var
  s: astring;
  i: astrindex;
begin
  argv(1, s); { This predefined Pascal function
               is a MIPS extension }
  writeln(output, s);
  { Guarantee that the string is null-terminated
    (but may bash the last character if the argument
    is too long). "lbound" and "hbound" are MIPS
    extensions. }
  s[hbound(s)] := chr(0);
  for i := lbound(s) to hbound(s) do
    if s[i] = ' ' then
      begin
        s[i] := chr(0);
        break;
      end;
  writeln(output, atoi(s));
end.

```

Checks for null character.

For more information on `atoi`, see the `atof(3)` (for BSD) or `strtol(3c)` (for System V) man page in the *UNIX Programmer's Manual*. See Figure 3.3 for another example of passing strings between C and Pascal.

Variable number of arguments. C functions can be defined that take a variable number of arguments (*printf* and its variants are examples). Such functions cannot be called from Pascal.

Type checking. Pascal checks certain variables for errors at execution time, whereas C doesn't. For example, in a Pascal program, when a reference to an array exceeds its bounds, the error is flagged (if runtime checks aren't suppressed). You could not expect a C program to detect similar errors when you pass data to it from a Pascal program.

One main routine. Only one main routine is allowed per program. The main routine can be written either in Pascal or C. Here are examples of C and Pascal main routines:

Pascal	C
<pre> program p(input,output); begin writeln("hi!"); end. </pre>	<pre> main() { printf("hi\n!"); } </pre>

Calling Pascal from C

To call a Pascal function from C, write a C extern declaration to describe the return value type of the Pascal routine; then write the call with the return value type and argument types as required by the Pascal routine. See Figure 3.1 for an example.

C return values. Table 3.1 below serves as a guide to declaring the return value type.

If Pascal function returns:	Declare C function as:
integer ¹	int
cardinal ²	unsigned int
char	char
boolean	char
enumeration	unsigned, or corresponding enum enum (C's enum are signed)
real	none
double	double
pointer type	corresponding pointer type
record type	corresponding structure or union type
array type	corresponding array type

¹Applies also to subranges with lower bound <0.
²Applies also to subranges with lower bounds >=0.

Table 3.1. Declaration of Return Value Types.

To call a Pascal procedure from C, write a C extern declaration of the form

```
extern void name();
```

and then call it with actual arguments with appropriate types. Table 3.2 serves as a guide for what values to pass corresponding to the Pascal declarations. C does not permit declaration of the formal parameter types, but instead infers them from the types of the actual arguments passed. See Figure 3.2 for an example.

C to Pascal arguments. Table 3.2 shows the C argument types to declare in order to match those expected by the called Pascal routine.

If Pascal expects:	C argument should be:
integer	integer or char value $-2^{31}..2^{31}-1$
cardinal	integer or char value $0..2^{32}-1$
subrange	integer or char value in subrange
char	integer or char (0..255)
boolean	integer or char (0 or 1 only)
enumeration	integer or char (0..N-1)
real	none
double	float or double
procedure	struct {void *p(); int *I}
function	struct {function-type *f(); int *I}
pointer types ¹	pointer type und <0. := lbound(s)
reference parameter	pointer to the appropriate type
record types	structure or union type
by-reference array parameters	corresponding array type
by-reference text	FILE**
by-value array parameters	structure containing the corresponding array
¹ See note below.	

Table 3.2. Pascal to C Data Types.

NOTE: To pass a pointer to a function in a call from C to Pascal, you must pass a structure by value; the first word of the structure must contain the function pointer and the second word a zero. Pascal requires this format because it expects an environment specification in the second word.

Example: Calling a Pascal function. Figure 3.1 shows an example of a C routine calling a Pascal function.

```
Pascal routine

function bah (
    var f: text;
    i: integer
): double;
begin
    ...
end {bah};

C declaration of bah
extern double bah();

C call
int i; double d;
FILE *f;
d = bah(&f, i);
```

Figure 3.1. Calling a Pascal Function from C.

Example: Calling a Pascal procedure. Figure 3.2 shows an example of a C routine calling a Pascal procedure.

```
Pascal routine

type
    int_array = array[1..100] of integer;
procedure zero (
    var a: int_array;
    n: integer
): integer;
begin
    ...
end {zero};

C declaration
extern void zero();

C call
int a[100]; int n;
zero(a, n);
```

Figure 3.2. Calling a Pascal Procedure from C.

Example: Passing strings to a Pascal procedure. Figure 3.3 is an example of a C routine that passes strings to a Pascal procedure, which then prints them; the example illustrates two points:

- The Pascal routine must check for the null [chr(0)] character, which indicates the end of the string passed by the C routine.
- The Pascal routine must not write to *output*, but instead uses the *stdout* file-stream descriptor passed by the C routine.

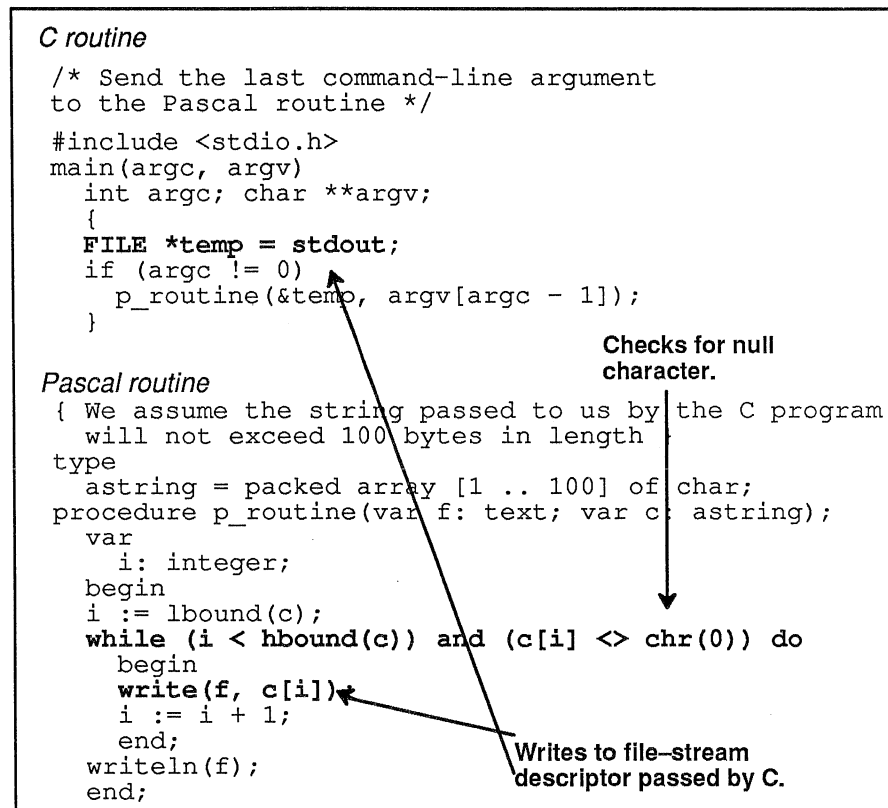


Figure 3.3. Passing Strings to a Pascal Procedure from C.

Calling C from Pascal

Pascal to C arguments. To call a C routine from Pascal, write a Pascal declaration describing the C routine. Write a procedure declaration or, if the C routine returns a value, a function declaration. Write parameter and return value

declarations corresponding to the C parameter types, using the table below as a guide.

If C expects:	Pascal parameter should be:
int ¹	integer
unsigned int ²	cardinal
short ³	integer (or -32768..32767)
unsigned short	cardinal (or 0..65535)
char ⁴	char
signed char	integer (or -128..127)
float	double
double	double
enum type	corresponding enumeration type
string (char *)	packed character array passed by reference (VAR)
pointer to function	none
FILE *	none
FILE **	text, passed by reference (VAR)
pointer type	corresponding pointer type or corresponding type passed by reference (VAR)
struct type	corresponding record type
union type	corresponding record type
array type	corresponding array type passed by reference (VAR)

¹ Same as types signed int, long, signed long, signed
² Same as types unsigned, unsigned long
³ Same as type signed short
⁴ Same as type unsigned char

Table 3.3. Pascal Parameter Data Type Expected by C.

Note: A Pascal routine cannot pass a function pointer to a C routine.

Example: Calling a C procedure. Figure 3.4 shows an example of calling a C procedure from Pascal.

```

C routine:
void bah (i, f, s)
    int i;
    float f;
    char *s;
{
    ...
}

Pascal declaration:
procedure bah (
    i: integer;
    f: double;
    var s: packed array[1..100] of char);
external;

Pascal call:
str := "abc\0";
bah(i, 1.0, str)

```

Figure 3.4. Calling a C Procedure from Pascal.

Example: Calling a C function. Figure 3.5 shows an example of calling a C function from Pascal.

```

C routine:
float humbug (f, x)
    FILE **f;
    struct scrooge *x;
{
    ...
}

Pascal declaration:
type
    scrooge_ptr = ^scrooge;
function humbug (
    var f: text;
    x: scrooge_ptr
): double;
external;

Pascal call:
x := humbug(input, sp);

```

Figure 3.5. Calling a C Function from Pascal.

Example: Passing arrays. Figure 3.6 shows an example of calling a C function from Pascal.

```
C routine:
int sum (a, n)
    int a[];
    unsigned n;
{
    ...
}

Pascal declaration:
type
    int_array = array[0..100] of integer;
function sum (
    var a: int_array;
    n: cardinal
    ): integer;
external;
avg := sum(samples, hbound(samples) + 1) /
        (hbound(samples)+1);
```

Figure 3.6. Passing Arrays Between Pascal and C.

Improving Program Performance

This chapter describes facilities that can help reduce the execution time of your programs; it contains the following major sections:

- Profiling, which describes the advantages of the profiler and how to use it. The profiler isolates those portions of your code where execution is concentrated and provides reports that indicate where you should devote your time and effort for coding improvements.
- Optimization, which describes the compiler optimization facility and how to use it. The section also gives examples showing optimization techniques.
- Limiting the Size of Global Data Area, which describes the global data area and how, through controlling the size of variables and constants that the compiler places in this area, you can improve program performance.

Introduction

The best way to produce efficient code is to follow good programming practices:

- *Choose good algorithms and leave the details to the compiler.*
- *Avoid tailoring your work for any particular release or quirk of the compiler system.*

As technological advances cause MIPS to make changes to the current compiler system, anything you tailor now might negatively affect future program performance. Moreover, tailored code might not work at all with new versions of the system. To take action on possible compiler inefficiencies, report them directly to MIPS.

Profiling

This section describes the concept of profiling, its advantages and disadvantages, and how to use the profiler.

Overview

Profiling helps you find the areas of code where most of the execution time is spent. In the typical program, execution time is confined to a relatively few sections of code; it's profitable to concentrate on improving coding efficiency in only those sections. The compiler system provides the following profile information:

- pc sampling (*pc* stands for *program counter*), which highlights the execution time spent in various parts of the program.

You obtain pc sampling information by link editing the desired source modules using the `-p` option and then executing the resulting program object, which generates profile data in raw format.

- invocation counting, which gives the number of times each procedure in the program is invoked.
- basic block counting, which measures the execution of basic blocks (a basic block is a sequence of instructions that is entered only at the beginning and which exits only at the end). This option provides statistics on individual lines.

You obtain invocation counting and basic block counting information using the `pixie` program. `Pixie` takes your source program and creates an equivalent program containing additional code that counts the execution of each basic block. Executing `pixie` and the equivalent program generate the profile data in raw format.

Using the `prof` program, you can create a formatted listing of the raw profile data. The listings can indicate where to correct sub-optimal coding, substitute better algorithms, or substitute assembly language. The listings also indicate if your program has exercised all portions of the code.

Figure 4.1 gives an example of a pc sampling listing produced from a program compiled with the `-p` compiler option. The `prof` program produced the listing from the raw profile data using the `-procedure` option.

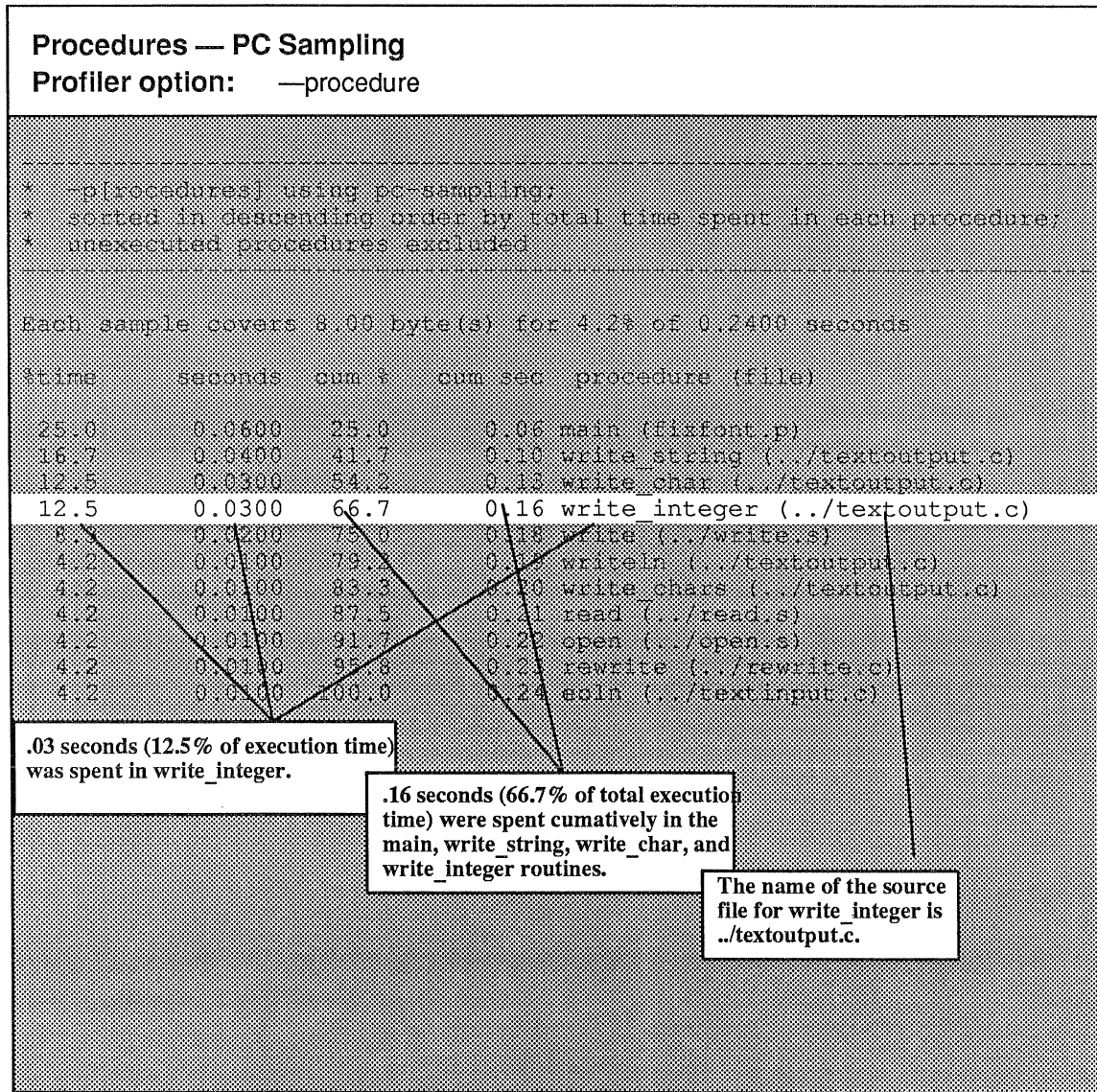


Figure 4.1. Profiler Listing for PC Sampling

Figures 4.2 through 4.6 shows listings from raw data produced by `pixie`. The `prof` option used is given at the top of each figure.

Procedures — Invocation Counting

Profiler option:—pixie -invocation

```

-----
* -i[nvocations] using basic-block counts;
* the called procedures are sorted in descending order by number of
* calls; a '?' in the columns marked '#calls' or 'line' means that data
* is unavailable because part of the program was compiled without
* profiling.
-----

```

called procedure #calls %calls from line, calling procedure (file):

called procedure	#calls	%calls	from line,	calling procedure (file):
eoln	4017	81.51	37	main (pix.p)
	453	9.19	35	main (pix.p)
	428	8.69	19	main (pix.p)
	30	0.61	17	main (pix.p)
write_char	4014	81.75	43	main (pix.p)
	453	9.23	45	main (pix.p)
	442	9.00	42	main (pix.p)
	1	0.02	47	main (pix.p)
		0.37		
		0.63		
		0.40		
	0	0.00		
	0	0.00		
	0	0.00		
write_string	453	24.59	31	main (pix.p)
	453	24.59	29	main (pix.p)
	453	24.59	31	main (pix.p)
	453	24.59	31	main (pix.p)
	30	1.63	23	main (pix.p)
	0	0.00		
write_integer	453	50.00	31	main (pix.p)
	453	50.00	31	main (pix.p)
eof	453	93.40	45	main (pix.p)
	30	6.19	23	main (pix.p)
	1	0.21	28	main (pix.p)
	1	0.21	14	main (pix.p)
writeln	453	93.60	29	main (pix.p)
	30	6.20	23	main (pix.p)
	1	0.21	47	main (pix.p)
readln	453	93.79	39	main (pix.p)
	30	6.21	21	main (pix.p)
sbrk	4	66.67	207	morecore (./malloc.c)
	1	16.67	110	malloc (./malloc.c)
	1	16.67	115	malloc (./malloc.c)
close	4	100.00	108	fclose (./flsbuf.c)
fflush	4	100.00	107	fclose (./flsbuf.c)
	0	0.00	49	_filbuf (./filbuf.c)

eoln was called 4017 times from line 37 of main. This presented 81.51% of the calls to eoln.

The source code for main is the file pix.p.

Figure 4.2. Profiler Listing for Procedure Invocations.

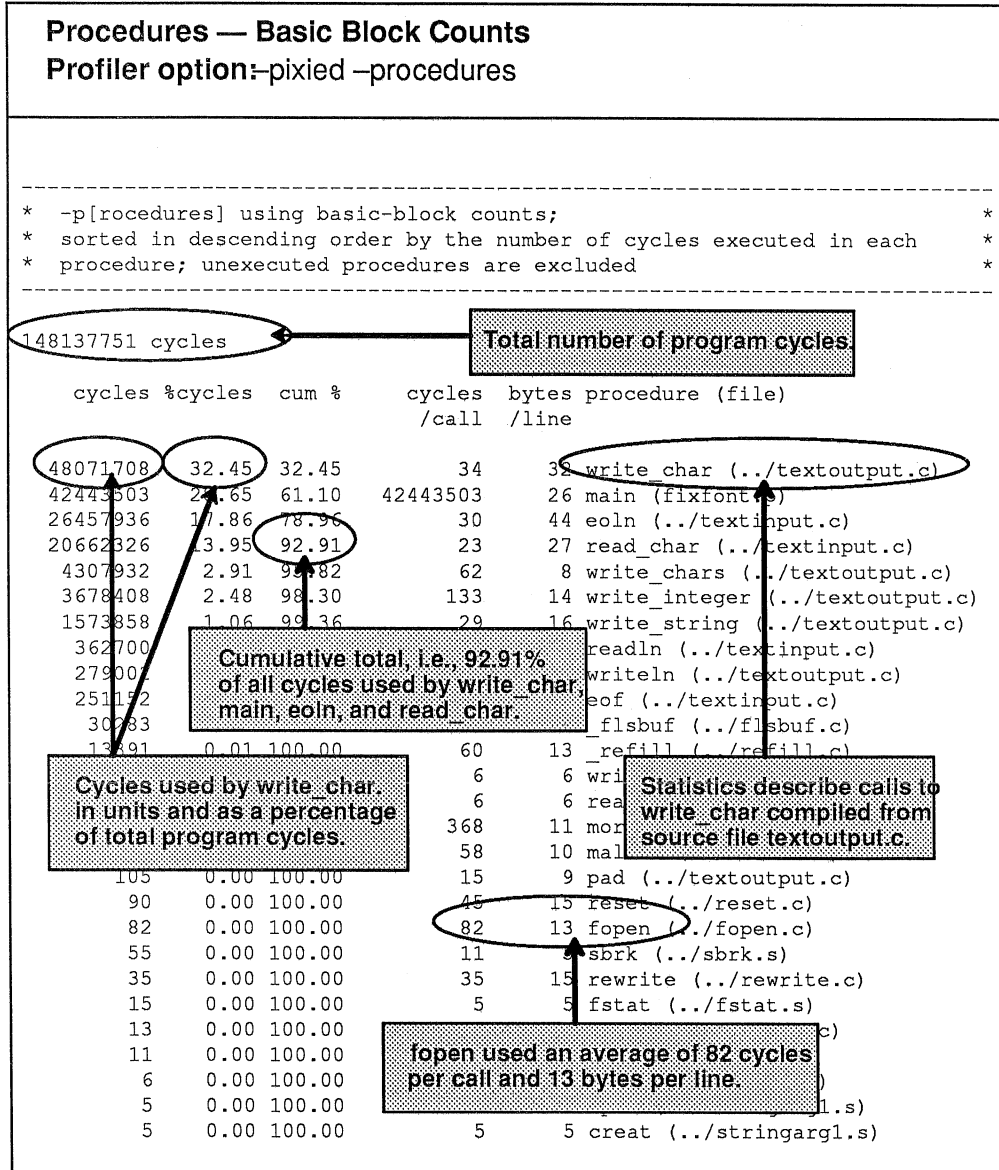


Figure 4.3. Profiler Listing for Procedures Based on Basic Block Counts.

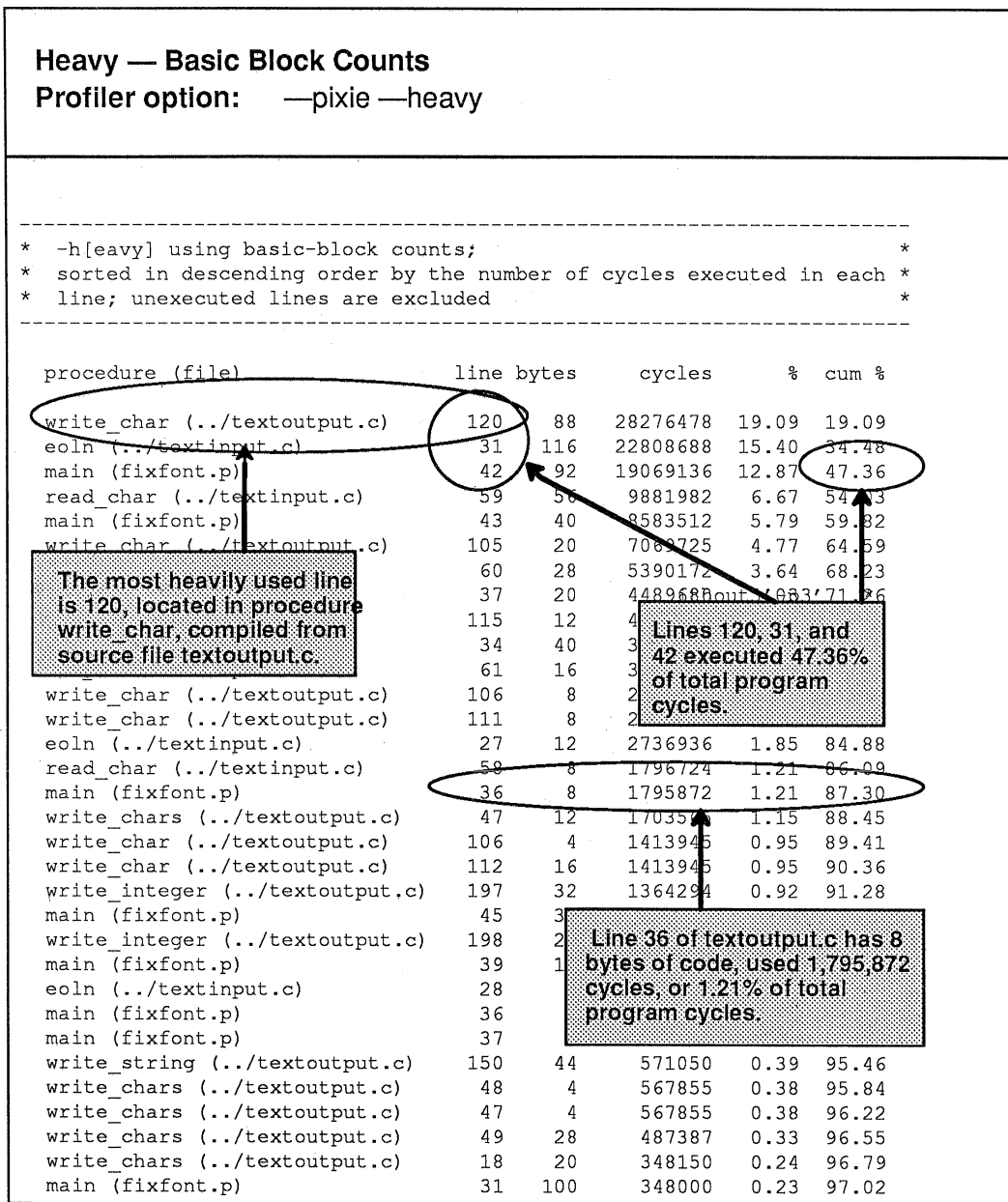


Figure 4.5. Profiler Listing for Heavy Line Usage.

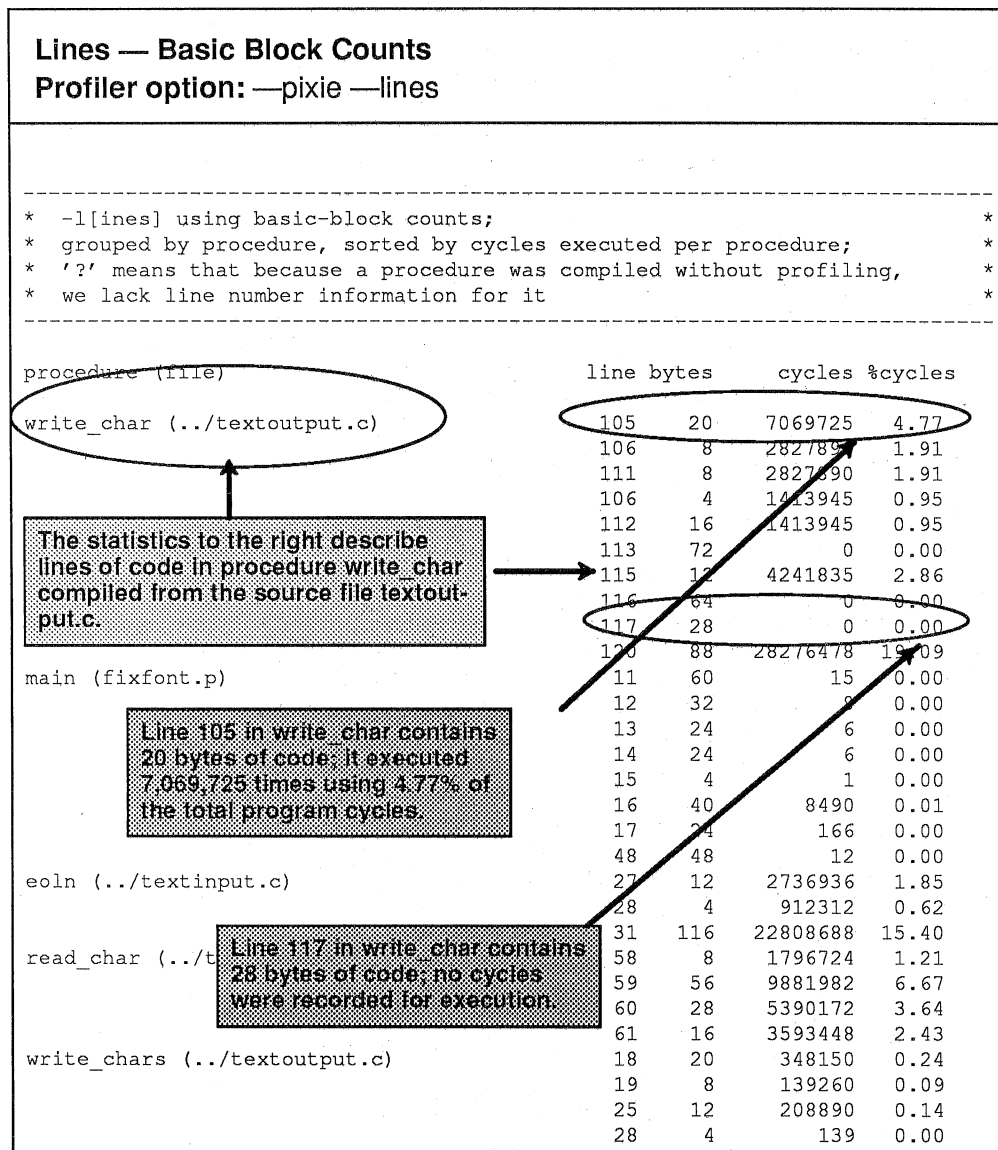


Figure 4.6. Profiler Listing for Line Information.

How Basic Block Counting Works

Figure 4.7 on the next pages gives the steps to follow in obtaining basic block counts. Details of the steps shown in the figure are as follows:

1. Compile and link–edit. Do *not* use the —p option. For example:

```
cc -c myprog.c
cc -o myprog myprog.o
```

2. Run the profiling program **pixie**. For example:

```
pixie -o myprog.pixie myprog
```

Pixie takes *myprog* and writes an equivalent program containing additional code that counts the execution of each

basic block. **Pixie** also generates a file (*myprog.Addr*s) that contains the address of each of the basic blocks. For more information, see the **pixie(1)** section in the *User's Reference Manual*.

3. Execute *myprog.pixie*, which was generated by **pixie**. For example:

```
myprog.pixie
```

This program generates the file *myprog.Counts*, which contains the basic block counts.

4. Run the profile formatting program **prof**, which extracts information from *myprog.Addr*s and *myprog.Counts*, and prints it in an easily readable format. For example:

```
prof -pixie myprog myprog.Addr
```

s myprog.Counts

NOTE: Specifying *myprog.Addr*s and *myprog.Counts* is optional; **pixie** searches by default for with names in having the format

*program_name.Addr*s and *program_name.Counts*.

You can run the program several times, altering the input data, and create multiple profile data files, if you desire. See the section **Averaging Prof Results** later in this chapter for an example.

You can include or exclude information on specific procedures within your program by using the `—only` or `—exclude` **prof** options (Table 4.1).

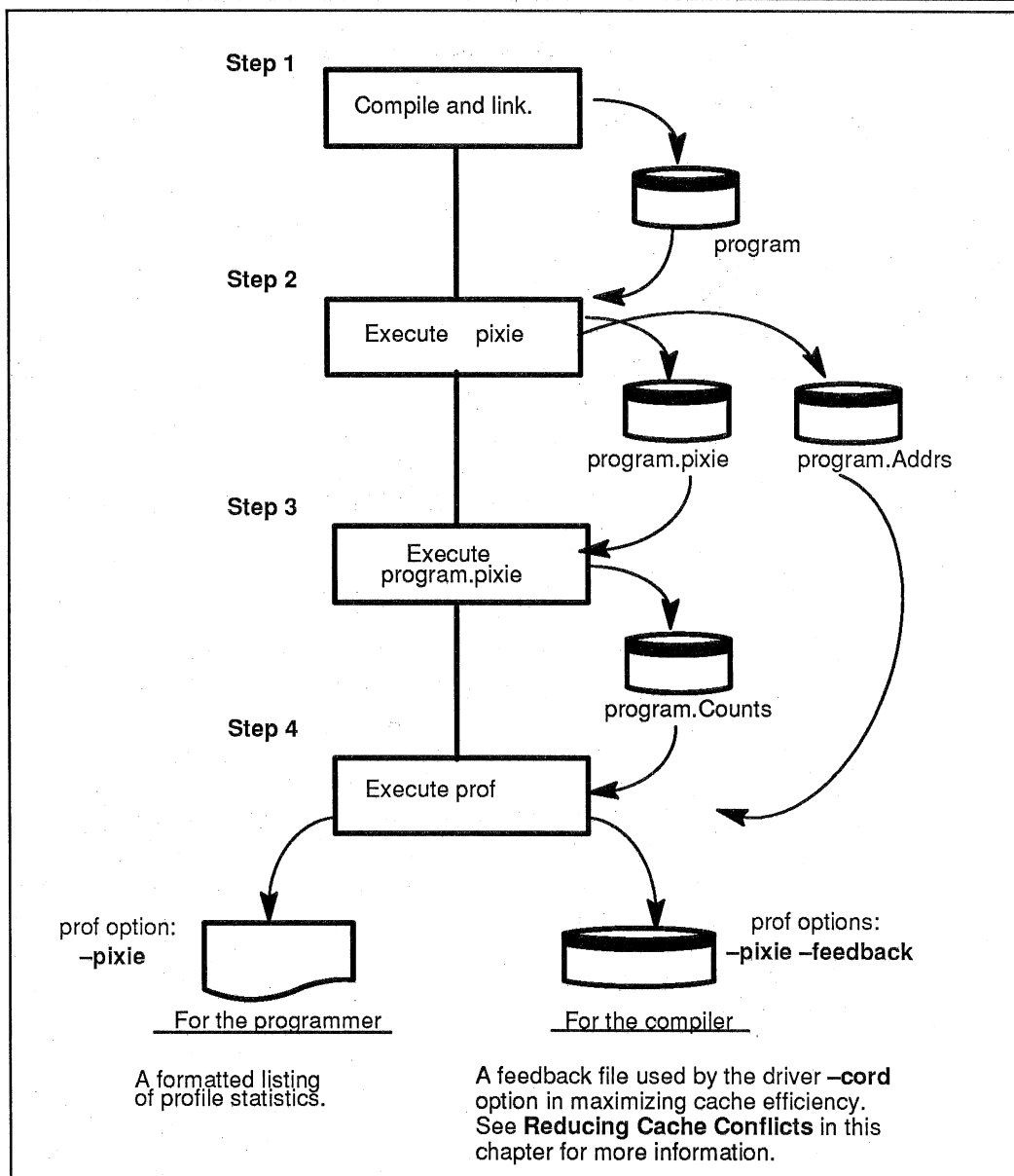


Figure 4.7. How Basic Block Counting Works.

Averaging Prof Results

A single run of a program may not produce the typical results you require. You can repeatedly run the version of your program created by `pixie`, varying the input with each run,; then, you can then use the resulting `.Counts` files to produce a consolidated report. For example:

1. Compile and link—edit. Do *not* use the `—p` option. For example:

```
cc -c myprog.c  
cc -o myprog myprog.o
```

2. Run the profiling program **pixie**. For example:

```
pixie -o myprog.pixie myprog
```

This step produces the *myprog.Addr*s file to be used in Step 4, as well as the modified program *myprog.pixie*.

3. Run the profiled program as many times as desired. Each time you run the program, a *myprog.Counts* file is created; rename this file before executing the next sample run. For example:

```
myprog.pixie < input1 > output1  
mv myprog.Counts myprog1.Counts  
myprog.pixie < input2 > output2  
mv myprog.Counts myprog2.Counts  
myprog.pixie < input3 > output3  
mv myprog.Counts myprog3.Counts
```

4. Create the report as shown below.

```
prof -pixie myprog myprog.Addr
```

s myprog[123].Counts

prof takes an average of the basic block data in the *myprog1.Counts*, *myprog2.Counts*, and *myprog3.Counts* files to produce the profile report.

How PC-Sampling Works

Figure 4.8 gives the steps to follow in obtaining pc sampling information.

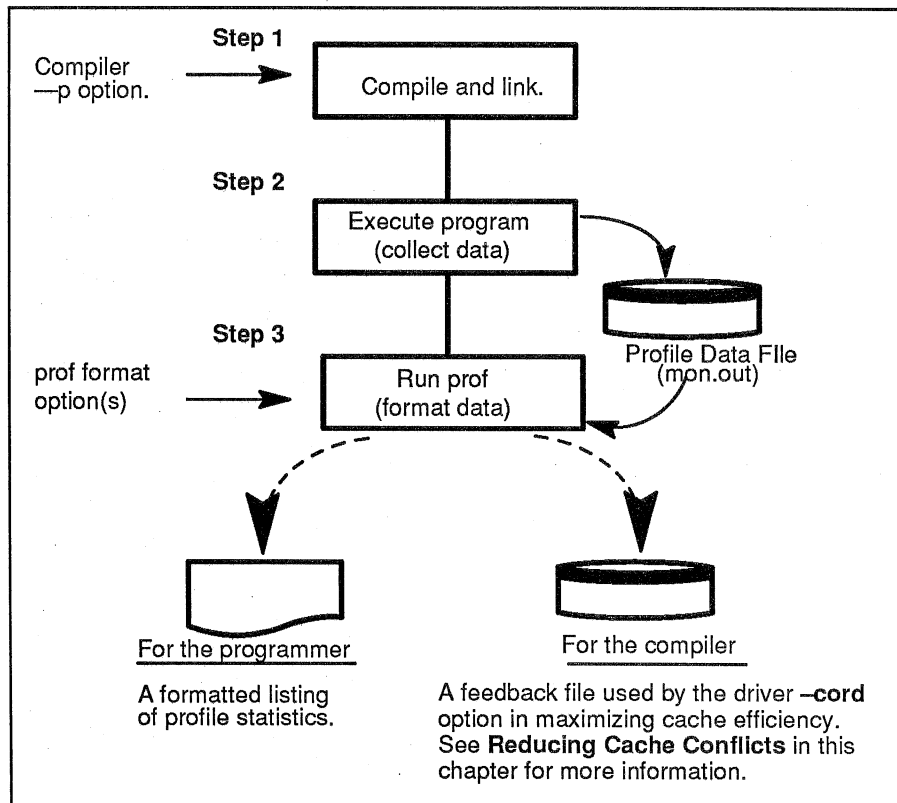


Figure 4.8. How PC-Sampling Works.

Details of the steps shown in Figure 4.8 are as follows:

1. Compile and link-edit using the `-p` option. For example:

```
cc -c myprog.c
cc -p -o myprog myprog.o
```

Note that the `-p` profiling option must be specified during the link editing step to obtain pc sampling information.

2. Execute the profiled program. During execution, profiling data is saved in the *profile data file* (the default is *mon.out*).

```
myprog
```

You can run the program several times, altering the input data, and create multiple profile data files, if you desire. See the section **Averaging Prof Results** later in this chapter for an example.

3. Run the profile formatting program **prof**, which extracts information from the profile data file(s) and prints it in an easily readable format.

```
prof -procedure myprog mon.out
```

For more information on **prof**, see the **prof(1)** section in the *User's Reference Manual*.

You can include or exclude information on specific procedures within your program by using the `—only` or `—exclude` profiler options (Table 4.1).

Creating Multiple Profile Data Files

When you run a program using `pc-sampling`, raw data is collected and saved in the profile data file *mon.out*. If you wish to collect profile data in several files, or specify a different name for the profile data file, set the environment variable `PROFDIR` as follows:

C Shell

```
setenv PROFDIR string
```

Bourne Shell

```
PROFDIR = string ; export PROFDIR
```

This causes the results to be saved in the file *string/pid.progname*, where *pid* is the process id of the executing program and *progname* is its name as it appears in `argv[0]`; *string* is the name of a directory you must create before you run the program.

Running the Profiler (prof)

The profiler program converts the raw profiling information into either a printed listing or an output file for use by the compiler. To run the program, type in **prof** followed by the optional parameters indicated below:

```
prof [options] [pname] { [profile_filename ... ] |
    [pname.Addr [pname.Counts] }
```

The **prof** parameters are summarized below:

options is one of the keyword or keyword abbreviations shown in Table 4.1. (You can specify either the entire name or the initial character of the option, as indicated in the table.)

pname specifies the name of your program. The default file is *a.out*.

profile_filename specifies one or more files containing the profile data gathered when the profiled program executed. If you specify more than one file, **prof** sums the statistics in the resulting profile listings.

pname.Addr (produced by running **pixie**) and *pname.Counts* (produced by running the **pixie**-modified version of the program).

The **prof** program takes defaults for *profile_filename* as follows:

- If you don't specify *profile_filename*, the profiler looks for the *mon.out* file; if this file doesn't exist, it looks for the profile input data file(s) in the directory specified by the `PROFDIR` environment variable (see the preceding section **Creating Multiple Profile Data Files**).

- If you don't specify *profile_filename*, but do specify `-pixie`, then **prof** looks for *pname.Addr*s and *pname.Counts* and provides basic block count information if these files are present.

You might wish to consider using the `—merge` option when you have more than one profile data file; this option merges the data from several profile files into one file. See Part 2 of Table 4.1 for information on the `—merge` option.

Profile List Program (prof) Options	
Name	Result
<code>-p[rocedures]</code>	Lists the time spent in each procedure. See Figures 4.3 for a sample output listing.
<code>-pixie</code>	<i>Basic block counting.</i> Indicates that information is to be generated on basic block counting, and that the <i>Addr</i> s and <i>Counts</i> file produced by pixie are to be used by default. See Figure 2.3 through 2.6 for examples of sample output.
<code>-i[nvocations]</code>	<i>Basic block counting.</i> Lists the number of times each procedure is invoked. The <code>-exclude</code> and <code>-only</code> options described below apply to callees, but not to callers. See Figure 4.2 for sample output.
<code>-l[ines]</code>	<i>Basic block counting.</i> List statistics for each line of source code. See Figure 4.6 for sample output.
<code>-o[nly] proc_name</code>	Reports information on only the procedure specified by <i>procedure_name</i> , rather than on the entire program. You may specify more than one <code>-o</code> option. If you specify uppercase <code>-O</code> , prof uses only the name procedure(s), rather than the entire program, as the base upon which it calculates percentages. Excludes information on the procedure(s) (and their descendants) specified by <i>procedure_name</i> . If you specify uppercase <code>-E</code> for Exclude, prof also omits that
<code>-e[xclude] procedure_name</code>	<i>Basic block counting.</i> Prints a list of procedures that are never invoked.
<code>-z[ero]</code>	<i>Basic block counting.</i> Prints a list of procedures that are never invoked.

Table 4.1 (1 of 3). Options for the Profile List Program (prof).

Profile List Program (prof) Options																																																																							
Name	Result																																																																						
-q[uit] n -q[uit] n% -q[uit] ncum%	<p>Allows you to condense output listings by truncating unwanted lines. You can truncate by specify n in three different ways:</p> <p>n n is an integer. All Lines after n line are truncated.</p> <p>n% n is an integer followed by the percentage sign. All lines after the line containing n% calls in the %calls column are truncated.</p> <p>n n is an integer followed by the characters <i>cum</i> (for <i>cumulative</i>) and a percentage sign. All lines after the line containing ncum% calls in the cum% column are truncated.</p> <p>Below are three examples of using the -q option. Any one of the three specifications shown below would eliminate the shaded area shown in the shaded area of the sample listing.</p> <pre>-prof -q 4 -prof -q 13% -prof -q 92cum%</pre> <table> <thead> <tr> <th></th> <th>calls</th> <th>%calls</th> <th>cum%</th> <th></th> </tr> </thead> <tbody> <tr> <td>48071708</td> <td>32.45</td> <td>32.45</td> <td>6.0090</td> <td></td> </tr> <tr> <td>42443503</td> <td>28.65</td> <td>61.10</td> <td>5.3054</td> <td></td> </tr> <tr> <td>26457936</td> <td>17.86</td> <td>78.96</td> <td>3.3072</td> <td></td> </tr> <tr> <td>20662326</td> <td>13.95</td> <td>92.91</td> <td>2.5828</td> <td></td> </tr> <tr> <td>4307932</td> <td>2.91</td> <td>95.82</td> <td>0.5385</td> <td></td> </tr> <tr> <td>3678408</td> <td>2.48</td> <td>98.30</td> <td>0.4598</td> <td></td> </tr> <tr> <td>1573658</td> <td>1.06</td> <td>99.36</td> <td>0.1967</td> <td></td> </tr> <tr> <td>362700</td> <td>0.24</td> <td>99.61</td> <td>0.0453</td> <td></td> </tr> <tr> <td>279002</td> <td>0.19</td> <td>99.80</td> <td>0.0349</td> <td></td> </tr> <tr> <td>251152</td> <td>0.17</td> <td>99.97</td> <td>0.0314</td> <td></td> </tr> <tr> <td>30283</td> <td>0.02</td> <td>99.99</td> <td>0.0036</td> <td></td> </tr> <tr> <td>13391</td> <td>0.01</td> <td>100.00</td> <td>0.0017</td> <td></td> </tr> <tr> <td>2923</td> <td>0.00</td> <td>100.00</td> <td>0.0004</td> <td></td> </tr> </tbody> </table>		calls	%calls	cum%		48071708	32.45	32.45	6.0090		42443503	28.65	61.10	5.3054		26457936	17.86	78.96	3.3072		20662326	13.95	92.91	2.5828		4307932	2.91	95.82	0.5385		3678408	2.48	98.30	0.4598		1573658	1.06	99.36	0.1967		362700	0.24	99.61	0.0453		279002	0.19	99.80	0.0349		251152	0.17	99.97	0.0314		30283	0.02	99.99	0.0036		13391	0.01	100.00	0.0017		2923	0.00	100.00	0.0004	
	calls	%calls	cum%																																																																				
48071708	32.45	32.45	6.0090																																																																				
42443503	28.65	61.10	5.3054																																																																				
26457936	17.86	78.96	3.3072																																																																				
20662326	13.95	92.91	2.5828																																																																				
4307932	2.91	95.82	0.5385																																																																				
3678408	2.48	98.30	0.4598																																																																				
1573658	1.06	99.36	0.1967																																																																				
362700	0.24	99.61	0.0453																																																																				
279002	0.19	99.80	0.0349																																																																				
251152	0.17	99.97	0.0314																																																																				
30283	0.02	99.99	0.0036																																																																				
13391	0.01	100.00	0.0017																																																																				
2923	0.00	100.00	0.0004																																																																				

Table 4.1 (2 of 3). Options for the Profile List Program (prof).

Profile List Program (prof) Options	
Name	Result
-h[eavy]	Basic block counting. Same as the -lines option, but sorts the lines by their frequency of use. See Figure 4.5 for a sample output listing.
-c[lock]n	Basic block counting. Lists the number of seconds spent in each routine, based on the CPU clock frequency <i>n</i> , expressed in megahertz; <i>n</i> defaults to 8.0 if omitted. Never use the default if the next argument <i>program_name</i> or <i>profile_name</i> begins with a digit. See Figure 4.4 for a sample output listing.
-t[estcoverage]	Basic block counting. Lists line numbers that contain code that is never executed.
-m[erge]filename	This option is useful when multiple input files of profile data (normally in <i>mon.out</i>) are used. The option causes the profiler to merge the input files into filename, making it possible to specify the name of the merged file (instead of several file names) on subsequent profiler runs.
-f[eedback] filename	Produces a file used by the driver -cord option to maximize cache efficiency. See Reducing Cache Conflicts in this chapter for details.

Table 4.1 (3 of 3). Options for the Profile List Program (prof).

Optimization

This section gives background on the compiler optimization facilities and describes their benefits, the implications of optimizing and debugging, and the major optimizing techniques.

Global optimizer. The global optimizer is a single program that improves the performance of RISCompiler object programs by transforming existing code into more efficient coding sequences. Although the same optimizer processes all compiler optimizations, it does distinguish between the various languages supported by the RISCompiler system programs to take advantage of the different language semantics involved.

Today, most compilers perform certain code optimizations, although the extent to which they perform these optimizations varies widely. The MIPS RISCompiler system performs more extensive optimizations compared with the average compiler available. These advanced optimizations are the results of the latest research into better and more powerful compiler techniques.

The compiler system performs both machine-independent and machine-dependent optimizations. RISComputers and other machines with RISC architectures provide a better target for machine-dependent optimizations. This is because

the low-level instructions of RISC machines provide more optimization opportunities than the high-level instructions in other machines. Even optimizations that are machine-independent have been found to be effective on machines with RISC architectures. Although most of the optimizations performed by the global optimizer are machine-independent, they have been specifically tailored to the RISC/os environment.

Benefits. The primary benefits of optimization, of course, are faster running programs and smaller object code size. However, the optimizer can also speed up development time. For example, your coding time can be reduced by leaving it up to the optimizer to relate programming details to execution time efficiency. This frees you up to focus on the more crucial global structure of your program. Moreover, programs often yield optimizable code sequences regardless of how well you write your source program.

Optimization and debugging. Optimize your programs only when they are fully developed and debugged. Although the optimizer doesn't alter the flow of control within a program, it may move operations around so that the object code doesn't correspond to the source code. These changed sequences of code may create confusion when using the debugger.

Optimization and bounds checking. The compiler option `—C`, which performs bounds checking in Pascal programs, inhibits some optimizations. Therefore, unless bounds checking is crucial, you shouldn't specify the `pc —C` option when you optimize a Pascal program.

Loop optimization. Optimizations are most useful in program areas that contain loops. The optimizer moves loop-invariant code sequences outside loops so that they are performed only once instead of multiple times. Apart from loop-invariant code, loops often contain loop-induction expressions that can be replaced with simple increments. In programs composed of mostly loops, global optimization can often reduce the running time by half.

The following examples show the results of loop optimization. The source code below was compiled with and without the `—O` compiler optimization option:

```
void
left(a, distance)
  char a[];
  int distance;
  {
  int j, length;

  length = strlen(a) - distance;
  for (j = 0; j < length; j++)
    a[j] = a[j + distance];
  }
```

The following listings show the unoptimized and optimized code produced by the compiler. Note that the optimized version contains fewer total instructions and fewer instructions that reference memory. Wherever possible, the optimizer replaces load and store instructions (which reference memory) with the faster computational instructions that perform operations only in registers.

Unoptimized:

loop is 13 instructions long using 8 memory references.

```
# 8      for (j=0; j<length; j++)
        sw      $0, 36($sp)      # j = 0
        ble     $24, 0, $33      # length >= j
$32:
# 9      a[j] = a[j+distance];
        lw      $25, 36($sp)     # j
        lw      $8, 44($sp)     # distance
        addu   $9, $25, $8      # j+distance
        lw      $10, 40($sp)    # address of a
        addu   $11, $10, $9     # address of a[j+distance]
        lbu    $12, 0($11)     # a[j+distance]
        addu   $13, $10, $25    # address of a[j]
        sb     $12, 0($13)     # a[j]
        lw      $14, 36($sp)    # j
        addu   $15, $14, 1     # j+1
        sw     $15, 36($sp)    # j++
        lw      $3, 32($sp)    # length
        blt    $15, $3, $32    # j < length
$33:
```

Optimized:

loop is 6 instructions long using 2 memory references.

```
# 8      for (j=0; j<length; j++)
        move    $5, $0         # j = 0
        ble     $4, 0, $33    # length >= j
        move    $2, $16        # address of a[j]
        addu   $6, $16, $17   # address of a[j+distance]
$32:
# 9      a[j] = a[j+distance];
        lbu    $3, 0($6)      # a[j+distance]
        sb     $3, 0($2)      # a[j]
        addu   $5, $5, 1      # j++
        addu   $2, $2, 1      # address of next a[j]
        addu   $6, $6, 1      # address of next a[j+distance]
        blt    $5, $4, $32    # j < length
$33:                                     # address of next a[j+distance]
```

Register allocation. MIPS RISComputer architecture emphasizes the use of registers. Therefore, register usages have significant impact on program performance. For example, fetching a value from a register is significantly faster than fetching a value from storage. Thus, to perform its intended function, the optimizer must make the best possible use of registers.

In allocating registers, the optimizer selects those data items most suited for registers, taking into account their frequency of use and their location in the program structure. In addition, the optimizer assigns values to registers so that their contents move minimally within loops and during procedure invocations.

Optimizing separate compilation units. The optimizer processes one procedure at a time. Large procedures offer more opportunities for optimization, since more inter-relationships are exposed in terms of constructs and regions.

However, because of their size, large procedures require more time than smaller-`feedbackfilename` ones.

The *uload* and *umerge* phases of the compiler permit global optimization among separate units in the same compilation. Often, programs are divided into separate files, called modules or compilation units, which are compiled separately. This saves compile time during program development, since a change requires recompilation of only one compilation unit rather than the entire program.

Traditionally, program modularity restricted the optimization of code to a single compilation unit at a time rather than over the full breadth of the program. For example, calls to procedures that reside in other modules couldn't be fully optimized together with the code that called them.

The *uload* and *umerge* phases of the compiler system overcomes this deficiency. The *uload* phase links multi-compilation units into a single compilation unit. Then, *umerge* orders the procedures for optimal processing by the global optimizer (`uopt`).

Optimization Options

Figure 4.10 on the next page shows the major processing phases of the compiler and how the compiler `—On` option determines the execution sequence. The table below summarizes the functions of each of the `—O` options.

Option	Result
<code>—O3</code>	<p>The <i>ulink</i> and <i>umerge</i> phases process the output from the compilation phase of the compiler, which produces symbol table information and the program text in an internal format called ucode.</p> <p>The <i>ulink</i> phase combines all the ucode files and symbol tables, and passes control to <i>umerge</i>. <i>Umerge</i> reorders the ucode for optimal processing by <code>uopt</code>. Upon completion, <i>umerge</i> passes control to <code>uopt</code>, which performs global optimizations on the program.</p>
<code>—O2</code>	Ulink and <i>umerge</i> are bypassed, and only the global optimizer (<code>uopt</code>) phase executes. It performs optimization only within the bounds of individual compilation units.
<code>—O1</code>	Ulink, <i>umerge</i> , and <code>uopt</code> are bypassed. However, the code generator and the assembler perform basic optimizations in a more limited scope.
<code>—O0</code>	Ulink, <i>umerge</i> , and <code>uopt</code> are bypassed, and the assembler bypasses certain optimizations it normally performs.

NOTE: The `—O3` options is not available for the `cobol` driver when the edition of this manual was printed. You should refer to the `cc(1) f77(1) pl1(1)`, or

pc(1) manual page, as applicable, in the *User's Reference Manual* for details on the `-O3` option and the input and output files related to this option.

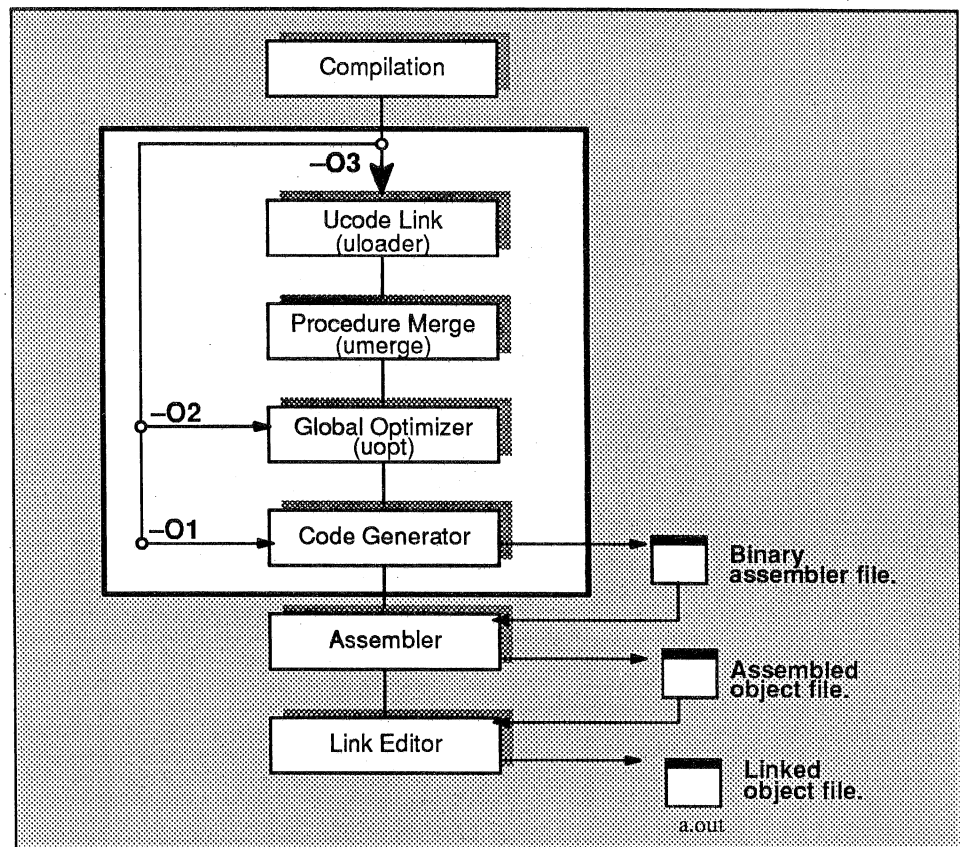


Figure 4.10. Optimization Phases of the Compiler.

Full Optimization (`-O3`)

This section provides examples using the `-O3` option. The examples given assume that the program *foo* consists of three files: *a.c*, *b.c* and *c.c*.

To perform procedure merging optimizations (`-O3`) on all three files, type in the following:

```
% cc -O3 -o foo a.c b.c c.c
```

If you normally use the `-c` option to compile the *.o* object file, follow these steps:

1. compile each file separately using the `-j` option by typing in the following:

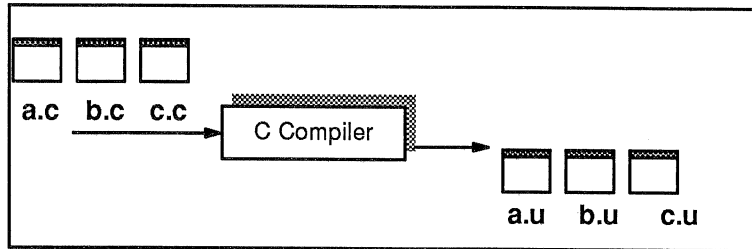
```
% cc -j a.c
```

```
% cc -j b.c
```

```
% cc -j c.c
```

The `-j` option causes the compiler driver to produce a *.u* file (the standard compiler front-end output, which is made up of ucode);

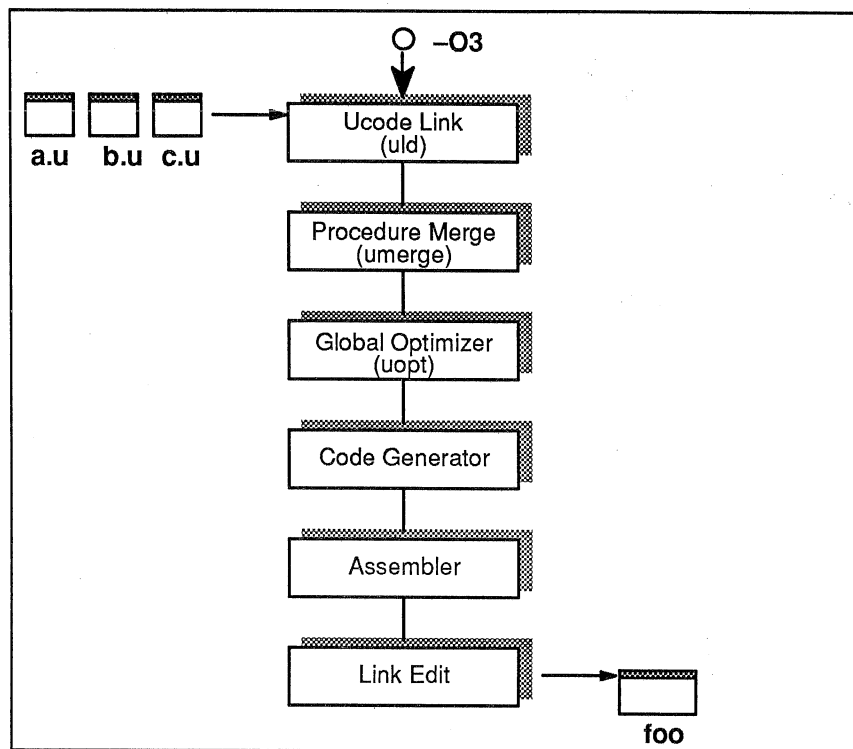
ucode is an internal language used by the compiler). None of the remaining compiling phases are executed, as illustrated below. The figure below illustrates the results after execution of the three commands shown above.



2. Enter the the following statement to perform optimization and complete the compilation process.

```
% cc -O3 -o foo a.u b.u c.u
```

The figure below illustrates the results of executing the above statement.



Optimizing Large Programs

If you wish to ensure that *all* program modules are optimized *regardless* of size, specify the driver **-Olimit** option at compilation.

Because compilation time increases by the square of the program size, the **RIS-C** Compiler system enforces a top limit on the size of a program that can be optimized. This limit was set for the convenience of users who place a higher pri-

ority on the compilation turnaround time than on optimizing an entire program. The `—Olimit` option removes the top limit and allows those users who don't mind a long compilation to fully optimize their programs.

Optimizing Frequently Used Modules

You may want to compile and optimize modules that are frequently called from programs written in the future. This can reduce the compile and optimization time required when the modules are needed.

In the examples that follow, *b.c* and *c.c* represent two frequently used modules that you wish to compile and optimize, retaining all the necessary information to link them with future programs; *future.c* represents one such program.

1. Compile *b.c* and *c.c* separately by entering the following statements:

```
% cc -j b.c
% cc -j c.c
```

The `—j` option causes the front end (first phase) of the compiler to produce two ucode files *b.u* and *c.u*.

2. Create manually a file containing the external symbols in *b.c* and *c.c* to which *future.c* will refer. Each symbolic name must be separated by at least one blank. Consider the following skeletal contents of *b.c* and *c.c*.

<pre>b.c foo() { . . } bar() { . . } zot() { . . } struct { . . } work;</pre>	<pre>c.c x() { . . } help() { . . } struct { . . } ddata; y() { . . }</pre>
---	---

In this example, *future.c* will call or reference only *foo*, *bar*, *x*, *ddata*, and *y* in the *b.c* and *c.c* procedures. A file (named *extern* for this example) must be created containing the following symbolic names:

```
foo bar x ddata y
```

(The structure *work*, and the procedures *help* and *zot* are used internally only by *b.c* and *c.c*, and thus aren't included in *extern*.)

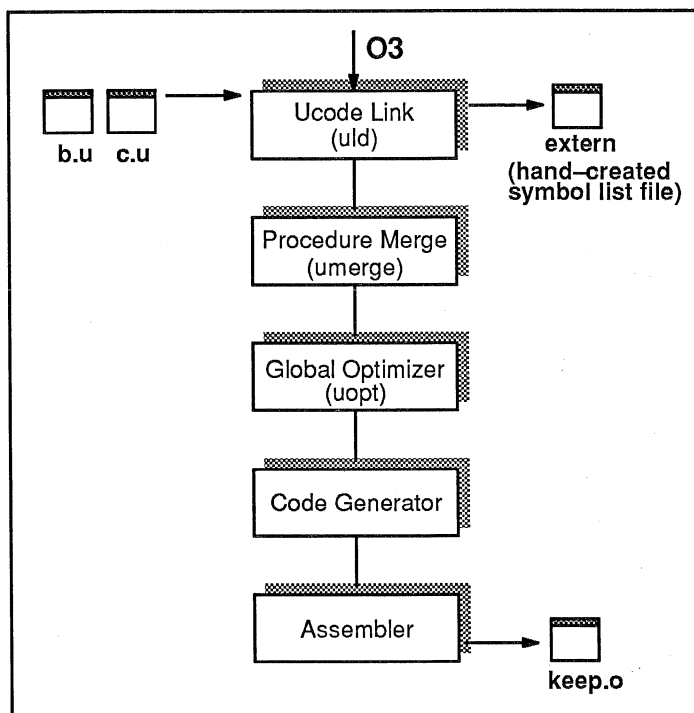
If you omit an external symbolic name, an error message is generated (see Step 4 below).

- Now, optimize the *b.u* and *c.u* modules (Step 1) using the *extern* file (Step 2) as follows:

```
% cc -c -O3 -kp extern b.u c.u -o keep.o
```

In the `—kp` option, *k* designates that the link editor option *p* is to be passed to the ucode loader.

The figure below illustrates Step 3.



- Create a ucode file and an optimized object code file (*foo*) for *future.c* as follows:

```
% cc -j future.c
% cc -O3 future.u keep.o -o foo
```

The following message may appear; it means that the code in *future.c* is using a symbol from the code in *b.c* or *b.c* that was not specified in the file *extern*.

```
zot: multiply defined hidden external (should have been
preserved)
```

Go to Step 5 if this message appears.

5. Include *zot*, which the message indicates is missing, in the file *extern* and recompile as follows:

```
% cc -O3 -c -kp extern b.u c.u -o keep.o
% cc -O3 future.u keep.o -o foo
```

Building a Ucode Object Library

Building a ucode object library is similar to building a *coff* object library. First, compile the source files into ucode object files using the compiler driver option `-j` and using the archiver just as you would for *coff* object libraries. Using the above example, to build a ucode library (*libfoo.b*), type in the following:

```
% cc -j a.c
% cc -j b.c
% cc -j c.c
% ar crs libfoo.b a.u b.u c.u
```

Conventional names exist for ucode object libraries (*libx.b*) just as they do for *coff* object libraries (*libx.a*).

Using Ucode Object Libraries

Using ucode object libraries is similar to using *coff* object files.^a To load from a ucode library, specify a `-klx` option to the compiler driver or the ucode loader. For example, to load from the ucode library the file created in the previous example, type in the following:

```
% cc -O3 file1.u file2.u -klfoo -o output
```

Remember that libraries are searched as they are encountered on the command line, so the order in which you specify them is important. If a library is made from both assembly and high level language routines, the ucode object library contains code only for the high level language routines and not all the routines as the *coff* object library. In this case, you must specify to the ucode loader both the ucode object library and the *coff* object library, in that order to ensure that all modules are loaded from the proper library.

If the compiler driver is to perform both a ucode load step and a final load step, the object file created after the ucode load step is placed in the position of the first ucode file specified or created on the command line in the final load step.

Improving Global Optimization

This section contains coding hints recommended to increase optimizing opportunities for the global optimizer (*uopt*). You should read through the recommendations in this section and, where possible, apply them to your code.

C, Pascal, and FORTRAN Programs

Do not use indirect calls. Avoid indirect calls (calls that use routines or pointers to functions as arguments). Indirect calls cause unknown side effects (that is, change global variables) that can reduce the amount of optimization.

C and Pascal Programs

Function return values. Use functions to return values instead of reference parameters.

Do while and repeat. Use **do while** (for C) and **repeat** (for Pascal) instead of **while** or **for** when possible. For **do while** and **repeat**, the optimizer doesn't have to duplicate the loop condition in order to move code from within the loop to outside the loop.

Unions and variant records. Avoid unions (in C) and variant records (in Pascal) that cause overlap between integer and floating point data types. This keeps the optimizer from assigning the fields to registers.

Use local variables. Avoid **global** variables. In C programs, declare any variable outside of a function as **static**, unless that variable is referenced by another source file. Minimizing the use of global variables increases optimization opportunities for the compiler.

Value parameters. Use value parameters instead of reference parameters or global variables. Reference parameters have the same degrading effects as the use of pointers (see below).

Pointers and aliasing. Aliases can often be avoided by introducing local variables to store dereferenced results. (A *dereferenced* result is the value obtained from a specified address.) Dereferenced values are affected by indirect operations and calls, whereas local variables aren't. Therefore, they can be kept in registers. The following example shows how the proper placement of pointers and the elimination of aliasing lets the compiler produce better code.

General Compiler Options	
Option Name	Purpose
<code>-S</code>	Similar to <code>-c</code> , except produces assembly code in a <code>.s</code> file instead of object code in a <code>.o</code> file.
<code>-std</code>	Issues a warning message when the compiler finds a non-standard feature in the programming language of your source program.
<code>-U name</code>	Overrides a definition of a macro name that you specified with the <code>-D</code> option, or that is defined automatically by the driver.
<code>-v</code>	Lists compiler phases as they are executed. Use this option when you suspect a phase isn't being run as you intended. For example, the option might reveal that you failed to specify a library required by the link editor. For BSD 4.3 users, this option also prints resource usage of each phase.
<code>-V</code>	Prints the version number of the driver and its phases. When reporting a suspected compiler problem, you must include this number.
<code>-w</code>	Suppresses warning messages.
<code>-p1</code> or <code>-p</code>	Permits program counter (pc) sampling. This option provides operational statistics for use in improving program performance. See Chapter 4 for more details. Note: This option affects only the link editor and is ignored by the compiler front ends. When link editing as a separate step from compilation,, be sure to specify this option if pc sampling is desired.

Consider the following example, which uses pointers. Because the statement `*p++=0` might modify `len`, the compiler, for optimal performance, cannot place it in a register, but instead must load it from memory on each pass through the loop.

```

Source Code:
int len = 10;
char a[10];

void
zero()
{
    char *p;
    for (p = a; p != a + len; ) *p++ = 0;
}

Generated Assembly Code:
# 8 for (p = a; p != a + len; ) *p++ = 0;
    move    $2, $4          # p = a
    lw     $3, len
    addu   $24, $4, $3
    beq    $24, $4, $33    # a + len != a
$32:
    sb     $0, 0($2)       # *p = 0
    addu   $2, $2, 1       # p++
    lw     $25, len
    addu   $8, $4, $25
    bne    $8, $2, $32    # len + a != p
$33:

```

Two different methods can be used to increase the efficiency of this example: using subscripts instead of pointers and using local variables to store unchanging values.

Using subscripts instead of pointers. The use of subscripting in the procedure `azero` eliminates aliasing; the compiler keeps the value of `len` in a register, sav-

ing two instructions, and still uses a pointer to access *a* efficiently, even though a pointer isn't specified in the source code.

Source Code:	
<pre>void azero() { int i; for (i = 0; i != len; i++) a[i] = 0; }</pre>	
Generated Assembly Code:	
<pre> for (i = 0; i != len; i++) a[i] = 0; move \$2, \$0 # i = 0 beq \$4, 0, \$37 # len != 0 la \$5, a \$36: sb \$0, 0(\$5) # *a = 0 addu \$2, \$2, 1 # i++ addu \$5, \$5, 1 # a++ bne \$2, \$4, \$36 # i != len \$37:</pre>	

Using local variables. Specifying *len* as a local variable or formal argument (as shown below) ensures that aliasing can't take place and permits the compiler to place *len* in a register.

Source Code:	
<pre>char a[10]; void lpzero(len) { int len; { char *p; for (p = a; p != a + len;) *p++ = 0; } }</pre>	
Generated Assembly Code:	
<pre># 8 for (p = a; p != a + len;) *p++ = 0; move \$2, \$6 # p = a addu \$5, \$6, \$4 beq \$5, \$6, \$33 # a + len != a \$32: sb \$0, 0(\$2) # *p = 0 addu \$2, \$2, 1 # p++ bne \$5, \$2, \$32 # a + len != p \$33:</pre>	

In the previous example, the compiler generates slightly more efficient code for the second method.

Pascal Programs Only

Packed arrays. Packed arrays prevent moving induction expressions from within a loop to outside the loop. Use packed arrays only when space is crucial.

C Programs Only

Write straightforward code. For example, don't use ++ and -- operators within an expression. When you use these operators for their values rather than for their side-effects, you often get bad code.

For example:

Bad	Good
<pre>while (n--){ . . . }</pre>	<pre>while (n != 0) { n--; . . . }</pre>

Use register declarations liberally. The compiler automatically assigns variables to registers. However, specifically declaring a **register** type lets the compiler make more aggressive assumptions when assigning register variables.

Addresses. Avoid taking and passing addresses (& values). This can create aliases, make the optimizer store variables from registers to their home storage locations, and significantly reduce optimization opportunities that would otherwise be performed by the compiler.

VARARGs. Avoid functions that take a variable number of arguments. This causes the optimizer to unnecessarily save all parameter registers on entry.

Improving Other Optimization

The global optimizer processes programs *only* when you explicitly specify the **-O2** or **-O3** option at compilation. However, the code generator and assembler phases of the compiler *always* perform certain optimizations (certain assembler optimizations are bypassed when you specify the **-O0** option at compilation).

This section contains coding hints that, when followed, increase optimizing opportunities for the other passes of the compiler.

C, Pascal, and FORTRAN Programs

1. Use tables rather than **if-then-else** or **switch** statements.

For example:

OK	More Efficient
<pre>if (i == 1) c = "1"; else c = "0";</pre>	<pre>c = "01"[i];</pre>

2. As an optimizing technique, the compiler puts the first four parameters of a parameter list into registers where they re-

main during execution of the called routine. Therefore, you should always declare as the first four parameters those variables that are most frequently manipulated in the called routine with floating point parameters preceding non-floating point.

3. Use word-size variables instead of smaller ones if enough space is available. This may take more space but it is more efficient.

C Programs Only

1. Rely on libc functions (for example, *strcpy*, *strlen*, *strcmp*, *bcopy*, *bzero*, *memset*, and *memcpy*). These functions were hand-coded for efficiency.
2. Use the **unsigned** data type for variables wherever possible for the following reasons: (1) because it knows the variable will always be greater than or equal to zero (≥ 0), the compiler can perform optimizations that would not otherwise be possible, and (2) the compiler generates fewer instructions for multiply and divide operations that use the power of two. Consider the following example:

```
int i;
unsigned j;
...
return i/2 + j/2;
```

The compiler generates six instructions for the signed $i/2$ operations:

```
000000 20010002 li r1,2
000004 0081001a divr4,r1
000008 14200002 bner1,r0,0x14
00000c 00000000 nop
000010 03fe000d break      1022
000014 00001812 mflo      r3
```

The compiler generates only one instruction for the unsigned $j/2$ operation:

```
000018 0005c042 srlr24,r5,1 # j / 2
```

In the example, $i/2$ is an expensive expression; however, $j/2$ is inexpensive.

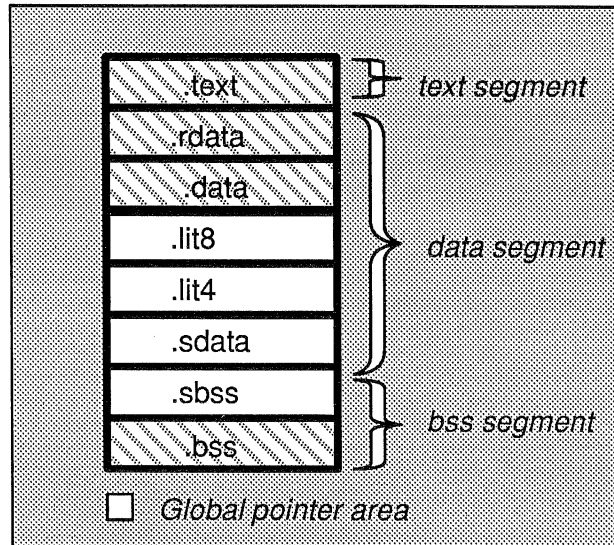
Pascal Programs Only

Predefined functions. Use predefined functions as much as possible. For example,

- use **max** and **min** rather than **if-then-else**.
- Also, use **shift** and bit-wise **and** instead of **div** and **mod**.

Limiting the Size of Global Data Area

The compiler places constants and variables in the `.lit8`, `.lit4`, `.sdata` and `.sbss` portions of the data and bss segments shown in the figure below. This area is referred to as the *global data area*.



(The `.rdata`, `.data`, `.lit8`, `.lit4`, and `.sdata` sections contain initialized data, and the `.sbss` and `.bss` sections reserve space for uninitialized data that is created by the kernel loader for the program before execution and filled with zeros. For more information on section data, see [Chapter 10](#) of the *Assembly Language Programmer's Guide*.)

Purpose of Global Data

In general, the compiler system emits two machine instructions to access a global datum. However, by using a register as a global pointer (called `$gp`), the compiler creates the 65536-byte global data area where a program can access any datum with a single machine instruction—only half the number of instructions required without a global pointer.

To maximize the number of individual variables and constants that a program can access in the global data area, the compiler first places those variables and constants that take the fewest bytes of memory. By default, the variables and constants occupying eight or fewer bytes are placed in the global data area, and those occupying more than eight bytes are placed in the `.data` and `.bss` sections.

Controlling the Size of Global Data Area

The more data that the compiler places in the global data area, the faster a program executes. However, if the data to be placed in the global data area exceeds 65536 bytes, the link editor prints an error message and doesn't create an executable object file. For most programs, the eight-byte default produces optimal results. However, the compiler provides the `-G` option to let you change the default size. For example, the specification

—G 12

causes the compiler to place only those variables and constants that occupy 12 or fewer bytes in the global data area.

Obtaining Optimal Global Data Size

The compiler places some variables in the global data area regardless of the setting of the `—G` option. For example, a program written in assembly language may contain `.sdata` directives that cause variables and constants to be placed into the global data area regardless of size. Moreover, the `—G` option doesn't affect variables and constants in libraries and objects compiled beforehand. To alter the allocation size for the global data area for data from these objects, you must recompile them specifying the `—G` option and the desired value.

Thus, two potential problems exist in specifying a maximum size in the `—G` option:

- Using a value that is too small can reduce the speed of the program.
- Using a value that is too large can cause more than the maximum 65536 bytes to be placed in the data area, creating an error condition and producing an unexecutable object module.

The link editor `—bestGnum` option helps overcome these problems by predicting an optimal value to specify for the `—G` option. The next sections give examples of using the `—bestGnum` option and the related `—nocount` and `—count` options.

Examples (Excluding Libraries)

When using the `—bestGnum` option exclusive of `—nocount` and `—count`, the compiler driver assumes that you cannot recompile any libraries to which it would link automatically; the driver causes the link editor not to consider these libraries when predicting the optimal maximum size. However, if you link to other system-supplied libraries, you must specify `—nocount` before the library. For example:

```
cc -bestGnum foo.c -nocount -lm
```

If you specify the option as shown below:

```
pc -bestGnum bogus.p
```

the compiler produces a message giving the best value for `—G`; if all program data fits into the global data area, a message indicates this. For example:

```
All data will fit into the global data area
Best -G num value to compile with is 80 (or greater)
```

Because all data fits into the global data area, no recompilation is necessary. Consider the following example, which specifies 70000 as the maximum size of a data item to be placed in the global data area:

```
pc ersatz.p -G 70000 -bestGnum
```

The above example produces the following messages:

```
gp relocation out-of-range errors have occurred and bad
object file produced (corrective action must be taken)
Best -G num value to compile with is 1024
```

In this example, the link editor doesn't produce an executable load module and recommends a recompilation as specified below:

```
pc real.p -G 1024
```

Example (Including Libraries)

You can explicitly specify that the link editor either include or exclude specific libraries in predicting the `—G` value. Consider the following example:

```
cc -o plotter -bestGnum plotter.o -nocount libiee.a
-count liblaser.a
```

In the above example, the link editor assumes that *libiee.a* cannot be recompiled and will continue to occupy the same space in the global data area. It assumes that *plotter.o* and *liblaser.a* can be recompiled and produces a recommended `—G` value to use upon recompilation.

Reducing Cache Conflicts

RISComputer hardware provides two high-speed caches—one for program data and the other for instructions—that temporarily hold data or instructions frequently used by the processor. During execution, instruction or data from specified memory locations are placed in the cache. Because the cache is much smaller than memory, a single cache location is shared by many distinct memory locations. The first cache location is shared by the 0th, 64KBth, 128KBth, ... memory locations. This mapping of every memory location to exactly one cache location is called a *direct mapped cache*.

A cache conflict occurs when a program references two instructions or data items that compete for the same location in the respective data or instruction cache. Normally this is no problem, except when the references are made repeatedly, as in a loop. Such repeated hits can degrade performance.

A serious instruction conflict could occur if, from within a loop, a call is made to a function that is a multiple of the cache size away. Basically, the function is placed in the cache, removing the instructions from the calling loop. Upon return, the calling loop replaces the instructions of the function, and this continues until the end of the loop.

You can eliminate major instruction cache misses within your programs by using the `—cord` driver option in combination with the `pixie` and `prof` programs. This option attempts to place the most frequently executed sections of code in

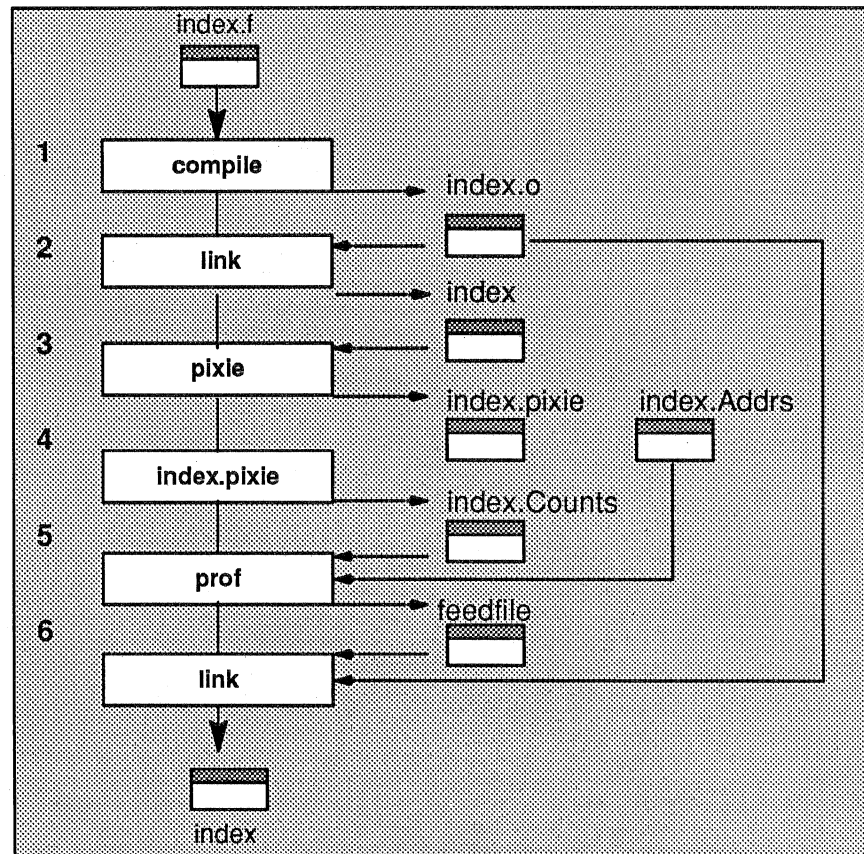
memory so that they don't conflict with each other. Here are the processing steps necessary to implement the `—cord` option:

```

1  f77 -c -O index.f
2  f77 -o index index.o
3  pixie -o index.pixie index
4  index.pixie
5  prof index -feedback feedfile
6  f77 -o index index.o -feedback feedfile -cord

```

The figure below illustrates the steps shown in the example for the reorganization of the procedures compiled from the source program *index.f*.



For more information, see the `prof(1)`, `pixie(1)`, or the `—cord` option in the applicable driver manual page—`cc(1)`, `pc(1)`, `f77(1)`, `pl1(1)`, or `cobol(1)`—in the *User's Reference Manual*.

Filling Jump Delay Slots

In jump instructions, there is a jump delay or latency of one instruction, which is called a *jump delay slot*. Whenever possible, the compiler inserts an instruction in the delay slot to avoid stalls in the execution pipeline of instructions. (See delay slot in the *MIPS R2000 RISC Architecture* manual for a detailed discussion.) The `—jmpopt` enables the compiler to fill additional delay slots at the cost of requiring more memory by the link editor. The default is `nojmpopt`.

For programs requiring the ultimate in performance, you should specify the **—jmpopt**. Then, the link editor attempts to insert executable instruction into those delay slots that that the compiler could not fill.

This chapter describes the source-level debugger DBX and tells how to use it. The debugger works for C, FORTRAN 77, Pascal, assembly language, and machine code. This chapter provides both reference and guide information on operating the debugger; the **dbx(1)** manual page in the *UNIX Programmer's Manual* provides additional reference information. The following topics are covered in this chapter:

Introduction. Introduces new users to the debugger and discusses general debugging issues, including where to start and how to isolate errors. It gives tips to people who are new to source-level debugging. If you're experienced at using debuggers, you might want to skip this part.

Running DBX. Shows how to run the debugger, including how to compile a program for debugging, and how to invoke and quit DBX.

Using DBX Commands. Describes the command syntax, expression precedence, data types, and constants, and lists the most common commands.

Working with the DBX Monitor. Describes how to use history, edit the command line, type multiple commands, and use facilities that help you complete program symbol names.

Controlling DBX. Describes how to work with variables, how to create command aliases, record and playback input and output, invoke a shell from DBX, and use the DBX status feature.

Examining Source Programs. Shows you how to specify source directories, move to a specified procedure or source file, list source code, search through the source code, call an editor from DBX, print symbolic names, and print type declarations.

Controlling Your Program. Describes how to run and rerun a program, execute single lines of code, return from procedure calls, start at a specified line, continue after a breakpoint, and assign values to program variables.

Setting Breakpoints. Describes how to set and remove breakpoints and continue executing a program after a breakpoint.

Examining Program State. Describes how to print stack traces, move up and down the activation levels of the stack, print register and variable values, and print information about the activation levels in the stack.

Debugging at the Machine Level. Describes the command that you use to debug machine code, including how to examine memory addresses and disassemble source code.

Introduction

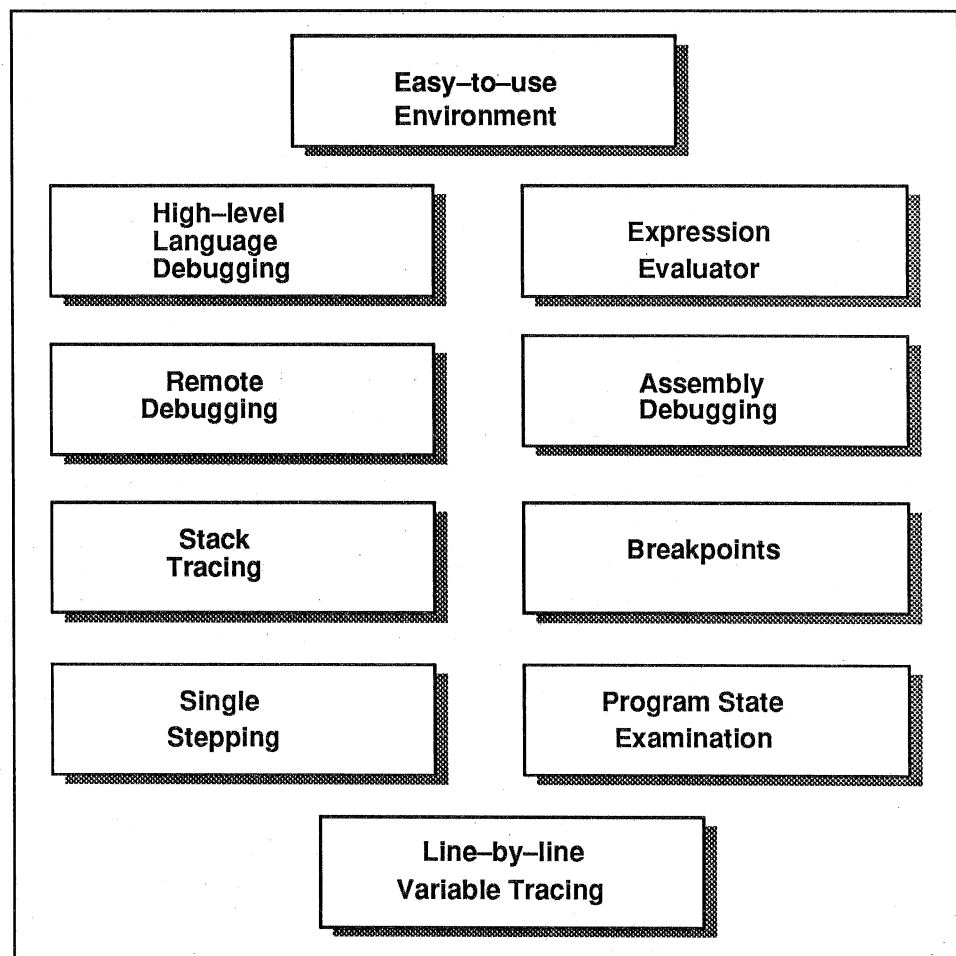
This section introduces the debugger and some debugging concepts; it also gives you some tips about how to approach a debugging session, including where to start, how to isolate errors, and how to avoid some common pitfalls.

If you're an experienced user, you may want to skip this section and go to the **DBX Command Summary** section at the end of the chapter, which contains a reference summary of all debugger commands.

Why Use a Source-Level Debugger?

A source-level debugger like DBX lets you trace problems in your program object at the source code, rather than at the machine code level. With DBX, you control a program's execution, symbolically monitoring program control flow, variables, and memory locations. You can also use DBX to trace the logic and flow of control to acquaint yourself with a program written by someone else.

The figure below summarizes the capabilities of DBX.



What Are Activation Levels?

This chapter frequently refers to the term *activation level*. Activation levels define the currently active scopes (usually procedures) on the stack. An activation stack is a list of calls that starts with the initial program (often `main`). The most recently called procedure or block is numbered 0. The next procedure called is numbered 1. The last activation level is always the main program—that is, the program that controls your whole program.

Activation levels can also consist of blocks that define local variables within procedures. You see activation levels when you do stack traces (see the `where` command) and when you move around the activation stack (see the `up`, `down`, and `func` commands). The example below shows the result of a `where` command.

Example:

```
>0 printline(pline = 0x7fff5b80) ["sam.c":58,0x2f7]
    printline is the most recently called
    procedure from $block1.

1 $block1["sam.c":47,0x2bb]
    $block1 defines its own local variables,
    even though it is part of main.

2 main(argc = 2, argv = 0x7fffeba0) ["sam.c":47,0x2bb]
    main is the main program.
```

Isolating Program Failures

DBX finds only runtime errors—you should fix compile errors before you start a debugging session.

To save time, you should start a debugging session using the more general commands (listed below), rather than debugging line by line. For example, if a program fails during execution, you would:

1. Invoke the program under DBX.
2. Do a stack trace using the `where` command to locate the point of failure.

NOTE: If you haven't stripped symbol table information from your program object, you can do a stack trace even when you don't compile your program with the debug flag `-g`.

3. Set breakpoints to isolate the error, using the `stop` commands.
4. Print the values of variables to see where a variable might have gone awry, using the `print` command.

If you still cannot find your error, you need to use other commands. **Using DBX Commands** in this chapter describes how to use each command.

Incorrect Output Results

If a program successfully terminates, but produces incorrect values or output, follow these steps:

1. Set a breakpoint where you think the problem is happening—for example, the code that generates the value or output.
2. Run the program.
2. Do a stack trace using the **where** command.
3. Print the values for the variables that you think might be causing the problem. Use the **print** command.
4. Return to *Step 1* until you find the problem.

Avoiding Some Pitfalls

The debugger cannot solve all problems. For example, if your program has a bad algorithm or incorrect logic, the debugger can only help you find the problem, not solve it. When information displayed by the debugger *appears* confusing or incorrect, taking the action listed below may correct the situation:

- Separate lines of source code into logical units wherever possible (for example, after **if** conditions); the debugger might not recognize a source statement written with several others on the same line.
- If executable code appears to be missing, it may have been contained in an include file. The debugger treats include files as a single line. If you wish to debug this code, remove it from the include file and compile it as part of your program.
- Make sure you recompile the source code after changing it, otherwise the source code displayed by the debugger won't match the executable code.
- If you stop the debugger by pressing **CTRL-Z**, and then resume the same debugging session, the debugger continues with the same object module specified at the start of the session. This means that, if you stop the debugger to fix a problem in your code, recompile, and return, the debugger won't reflect the change. You must start a new session as described in the section **Invoking DBX (dbx)**.
- When printing an expression that has the same name as a **dbx** keyword, you must enclose the expression within parentheses. For example, in order to print *output*, a keyword in the **play-back** and **record** commands, you must specify:

```
print (output)
```

- If the debugger won't display any variables and executable code, make sure you compiled the program with the appropriate `-g` compiler flag.

Running DBX

This part of **Chapter 5** describes how to run DBX, including how to do these things:

- compile your program for debugging
- create a command file
- invoke DBX from the shell
- end your debugging session

Compiling Your Program for Debugging

To use the debugger, you need to specify the `-g` option at compilation as described in **Chapter 1**. This option inserts symbol table information in your program object, which DBX uses to list source lines.

Don't optimize your program until it is fully developed and debugged. Although the optimizer doesn't alter the flow of control within a program, it may move operations around so that the object code doesn't correspond to the source code. These changed sequences of code may create confusion when you use the debugger.

You can do limited debugging on code compiled without the `-g` flag. These commands still work without `-g`:

- `stop in PROCEDURE`
- `stepi`
- `continue`
- `conti`
- `(ADDRESS)/<COUNT><MODE>`
- `tracei`

Although you can do limited debugging on this code, you will have an easier time if you recompile the code for debugging. The debugger does not warn you when you select an object file that you compiled without the `-g` flag.

Building a Command File

You can create a command file, called the `.dbxinit` file, using the system editor. You can put any DBX command in the file; when you invoke DBX, the commands are executed (you'll be prompted for required input). You can use a command file to customize your DBX environment or to specify a set of frequently used DBX commands.

DBX looks for *.dbxinit* first in the current directory and then in your home directory. If the file resides in the home directory, set the UNIX system HOME environment variable.

Here's an example of a *.dbxinit* file:

```
set $page = 5
set $lines = 20
set $prompt = "DBX>"
alias du dump
```

Invoking DBX (dbx)

You invoke DBX from the shell command line by entering **dbx** and the optional parameters shown below. After invocation, DBX sets the current function to the first procedure of the program.

Syntax:

Command	Select this command to..
<code>dbx [options] [objfile corefile]</code>	Invoke DBX from the shell command line.

If you don't explicitly specify *object_file*, DBX uses *a.out* by default. If you specify *core_file*, DBX lists the point of program failure. For core files, you can do stack traces and look at the code; however, you cannot run a program continuously—for example, set breakpoints or continue.

The *options* specifications are shown in the following table.

Option	Select this option to...
-I <i>dirname</i>	Tell DBX to look in the specified directory for source files. To specify multiple directories, you must use a separate -I for each. Unless you specify this option when you invoke DBX, it looks for source files in the current directory and in the object file's directory. From DBX, you can change directories with the <code>use</code> command.
-c <i>filename</i>	Select a command file other than your <i>dbxinit</i> file.
-i	Use interactive mode. This option does not treat <code>#s</code> as comments in a file. It also prompts for source even when it reads from a file. It has extra formatting as if for a terminal.
-r	Run your program immediately upon entering DBX.
-k	Turn on kernel debugging.

Example:

```

% dbx
dbx version 3 of 3/30/86 14:51
Type 'help' for help.
enter object file name (default is 'a.out'): sam
reading symbolic information...
main:23          if (argc <2)    {
(dbx)

```

Ending DBX (quit)

Use the **quit** command to end a debugging session.

Syntax:

Command	Select this command to...
quit q	End your debugging session.

Example:

```

(dbx) quit
%

```

In the above example, after entering *quit*, DBX prompts you to confirm that you want to exit.

Using DBX Commands

This part of **Chapter 5** covers these topics:

- command syntax
- expression and precedence
- data types and constants
- common debugging commands

DBX Command Syntax

This section describes the format of DBX commands. The following general conventions are used in each description:

- Words in lower-case typewriter font are literals, and you must type them as they are shown.
- Words in italics indicate variable values that you specify.
- Square brackets ([]) surrounding an argument mean that the argument is optional.

- DBX variable names appear in italics.
- DBX lets you type up to 10240 characters on an input line.
- You can continue long lines with a backslash (\). If the line gets too long, DBX prints an error message (see `fgets(1)` in the *UNIX Programmer's Manual*).
- The maximum string length is also 10240.

In addition to these general conventions, the words in upper-case typewriter font indicate variables for which specific rules apply. These words are given in the Table 5.1 on the next page.

The example below of the `stop in` command illustrates the syntax conventions just described:

```
stop VAR in PROCEDURE if EXP
```

You would type *stop*, *in*, and *if* as shown. You would type the values for *VAR*, *PROCEDURE* and *EXP* as defined in Table 5.1.

Syntax	You specify:
FILE	File name.
INT	Integer value.
EXP	Any expression including program variable names for the command. Expressions can contain DBX variables; for example, (<code>\$listwindow + 2</code>). If you want to use the words <i>in</i> , <i>to</i> or <i>at</i> in an expression, you must surround them with parentheses; otherwise, DBX assumes that these words are debugger key words.
PROCEDURE	Procedure name or an activation level on the stack.
LINE	A source code line number.
^ (caret)	Press the control key on your keyboard. Usually, you do as you simultaneously press another key.
ADDRESS	Any expression specifying a machine address.
REGEX	A regular expression string. See the <code>ed(1)</code> manual page in the <i>UMIPS Programmer's Manual</i> .
SIGNAL	A UNIX system signal. For Berkely, see the <code>sigvec(2)</code> manual page in the <i>UMIPS Programmer's Manual</i> . For SystemV, see the <code>signals(2)</code> manual page.

Table 5.1 (1 of 2). Keywords Used in Command Syntax Descriptions.

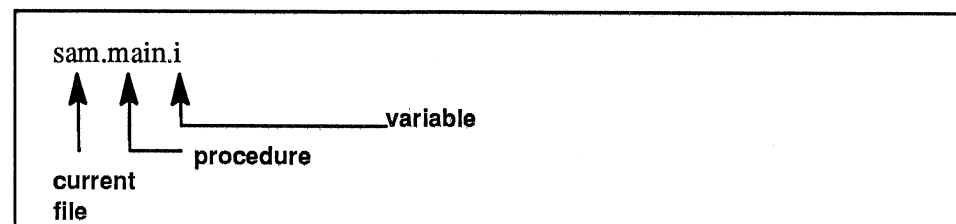
Syntax	You specify:
STRING	Any ASCII string.
DIR	A directory name.
ARGS	Program arguments (maximum allowed by DBX is 1000; however, system limits may also apply).
VAR	Valid program variable or DBX predefined variable (see Table 5.2). For machine-level debugging, VAR can also be an address. You must qualify program variables with duplicate names as described in the section Qualifying Variable Names .
NAME	DBX command name.
COMMAND_LIST	One or more commands, each separated by semicolons.

Table 5.1 (2 of 2). Keywords Used in Command Syntax Descriptions.

Qualifying Variable Names

DBX qualifies your variables by the file, the procedure, a block, or a structure. When you use commands like **print** to print a variable's value, DBX tells you the scope of the variable when the scope could be ambiguous (for example, you have a variable by the same name in different procedures). If you see the wrong variable, you can specify the full scope of your variable manually by separating scopes with periods.

Example:



DBX Expressions and Precedence

DBX recognizes expression operators from C, Pascal, and FORTRAN 77. Operators follow the C language precedence.

Debugger Operators		
Operator	Syntax	Description
#	("FILE" #EXP)	Uses the specified line number (#EXP) in that file.
	(PROCEDURE #EXP)	Uses the specified line number (#EXP) in that procedure.
	(#EXP)	Takes line number (#EXP) and returns the address for that line.

The following tables show language operators; note that // (instead of /) is used for divide.

C Language Operators	
Unary	&, +, -, *, sizeof() ~, //, (type), (type *)
Binary	<<, >>, ", !, ==, !=, <=, >=, <., >., &, &&, !, , +, -, *, %, [], ->

CAUTION: The `sizeof` operator specifies the number of bytes retrieved to get an element, not $(\text{number_of_bits}+7)/8$.

Pascal Language Operators	
Unary	not, ;-
Binary	, <=, >=, <., >., and, or, +, -, *, //, div, mod, [], ..

FORTRAN Operators	
Unary	-
Binary	+, -, *, //

NOTE: FORTRAN array subscripting must use [] instead of ().

DBX Data Types and Constants

DBX commands can use the following built-in data types:

Data Types	
Data Types	Description
\$address	pointer
\$unsigned	unsigned integer
\$char	character
\$boolean	boolean
\$real	double precision real
\$integer	signed integer
\$float	single precision real
\$double	double precision real
\$uchar	unsigned character
\$short	16-bit integer

You can also use the built-in data types for type coercion—for example, to see a variable as a type that the language you're using doesn't support.

Input Constants	
Constant	Description
false	0
true	nonzero
nil	0
0x <i>number</i>	hexadecimal
0 <i>number</i>	decimal
0 <i>number</i>	octal
<i>number</i>	decimal
<i>number</i> .. <i>[number]</i> [e E][+ -EXP]	float

NOTE: Overflow on non-float uses the right-most digits. Overflow on float uses the left-most of the mantissa and the highest or lowest exponent possible.

NOTE: The `$octin` DBX variable changes the default to octal. The `$hexin` variable changes the default to hexadecimal. See **Predefined DBX Variables**.

Output Constant	
Constant	Description
default	decimal

NOTE: The *\$octints* DBX variable changes the default to octal. The *\$hexints* variable changes the default to hexadecimal. See **Predefined DBX Variables**.

Basic DBX Commands

DBX offers many commands; however, for most debugging sessions, you need to use only those shown in the next table.

Common Debugging Commands	
Command	Select this command to...
/REGEX	Search ahead in your source file for a specific string.
?REGEX	Search back in your source file for a specific string.
continue	Continue executing your program.
down [EXP]	Move down the activation levels of the stack.
dump	Get all information that DBX has about a procedure.
func PROCEDURE	Select the procedure you want to look at.
list	Look at the 10 lines preceding and following the current line.
list EXP	Look at line specified by EXP.
print EXP	Print the value of any variable.
quit	End the debugging session.
run	Run the program being debugged.
rerun	Run the program again with the same arguments you specified to the run command.
step [EXP]	Step the specified number of lines.
stop at LINE	Stop at specified line in the source.
stop in PROCEDURE	Set a breakpoint at the beginning of a procedure.
up [EXP]	Move up the activation levels of the stack.
where	Do a stack trace to see what procedures are currently active.

Working with the DBX Monitor

This part of **Chapter 5** covers these topics:

- the history feature
- DBX command line editing
- typing of multiple commands

- DBX completion of program symbol names

Using History (history & ! commands)

The DBX history feature is similar to the C shell's history feature; however, DBX history doesn't work in the middle of the line or with !\$. It works only at the beginning of the line.

You can set the number of history lines using the `$lines` variable. The default is 20. To reset this variable, use the `set` command. See **Setting DBX Variables**.

To see a list of the commands in your history list, type the `history` (alias `h`) command. To repeat a previous command, use one of the exclamation point (!) commands.

Syntax:

Command	Select this command to...
<code>history</code>	Print the items in your history list.
<code>!string</code>	Repeat the most recent command that starts with the specified string.
<code>!INT</code>	Repeat the command associated with the specified integer.
<code>!-INT</code>	Repeat the command that occurred the specified integer before the most recent command.

Example:

```
(dbx) history
 10 print x
 11 print y
 12 print z
(dbx) !12
(!12 = printz)
123
(dbx)
```

Editing on the DBX Command Line

DBX provides commands that permit you to edit the command line. These commands make it possible for you to correct mistakes without having to retype an entire command. See the *UNIX Programmer's Manual* page `lsh(1)` for a full

description of the editing commands. Here are the ones that you'll most commonly use:

DBX Command Line Editing	
Command	Select this command to...
carriage return	Repeat the last command you issued to DBX. You can turn this feature off by setting the <i>\$repeatmode</i> variable to 0. See Setting DBX Variables .
^A	Move the cursor to the beginning of the command line.
^B	Move the cursor back one character.
^D	Delete the character at the cursor.
^E	Move the cursor to the end of the line.
^F	Move the cursor ahead one character.
^H ,DELETE	Delete the character immediately preceding the cursor.
^N	Move ahead one line. (This line comes from the history list.)
^P	Move back one line. (This line comes from the history list.)

NOTE: In the above table, the notation **^** represents the CTRL key. For example **^A** is the same as pressing the CTRL and A keys.

Typing Multiple Commands

By using a semicolon (;) as a separator, you can type multiple commands on the same command line. This can be useful when you use the **when** command. See **Writing Conditional Code in DBX (when)**.

Syntax:

Command	Select this command to...
COMMAND; COMMAND	Type multiple commands on the same line.

Example:

```
(dbx) stop at 58; rerun
[1] stop at 58 "sam.c":58
[1] stopped at [printline:58,0x2f8] pline->string
(dbx)
```

Completing Program Symbol Names

When you want to save typing or when you don't remember a symbol's full name, DBX can complete the name for you. If DBX finds a unique completion, it adds the completion to the input line; otherwise, it lists all possible completions and lets you choose the appropriate one.

To use this feature, type the first part of the name and press ^Z.

Syntax:

Command	Select this command to...
STRING^Z	Complete a symbol name or to see what symbol names contain the specified string.

Example:

```
(dbx) i^z
ioctl.ioctl .ioctl isatty.isatty .isatty i int
(dbx) i
```

↑
the list of all symbols
that begin with i in the
program

NOTE: You'll often see things that are data types and library symbols.

```
(dbx) print file^z
(dbx) print file_header_ptr
0x124ac
(dbx)
```

↑
dbx completes the
symbol name for you

Controlling DBX

This part of **Chapter 5** covers these topics:

- how to set and unset DBX variables
- the predefined DBX variables
- how to create and remove command aliases
- the predefined DBX aliases
- how to record and play back input and output
- how to invoke a shell from DBX
- how to check and delete DBX status items

Setting DBX Variables (`set`)

The `set` command defines a DBX variable, sets an existing DBX variable to a different type, or displays a list of existing DBX predefined variables.

You cannot define a debugger variable that has the same name as a program variable. You can see the setting for a single variable by using the `print` command. The DBX predefined variables are listed Table 5.2 in the section **Predefined DBX Variables**.

Syntax:

Command	Select this command to...
<code>set</code>	Display a list of DBX predefined variables.
<code>set VAR = EXP</code>	Assign a new value to a variable or to define a new variable.

Example:

```

(dbx) set
$listwindow      10 ← old value
$datacache       1
$main            "main"
$pagewindow      22
$page            1
$maxstrlen       128
$cursorline      24
more (no?) no
(dbx) set $listwindow = 15
(dbx) set

$listwindow      15 ← new value
$datacache       1
$main            "main"
$pagewindow      22
$page            1
$maxstrlen       128
$cursorline      24
more (no?) no
(dbx)

```

Removing Variables (unset)

Use the *unset* command to remove the specified DBX variable from the list. To see a full list of DBX variables, use the *set* command or see **Predefined DBX Variables**.

Syntax:

Command	Select this command to...
unset VAR	Unset the value of a DBX variable.

Example:

```

(dbx) set test = 5
(dbx) set
$listwindow      10
$datacache      1
$main           "main"
$pagewindow     22
$test           5
$maxstrlen      128
$cursorline     24
more (no?) no
(dbx) unset $page
(dbx) set
$listwindow      15
$datacache      1
$main           "main"
$pagewindow     22
$maxstrlen      128
$cursorline     24
more (no?) no
(dbx)

```

← new variable on list

← new variable removed from list

Predefined DBX Variables

The predefined DBX variables are listed in Table 5.2 starting on the next page. The variables that are preset, but which you can change, are indicated by **I**, **B**, and **S** notations in the *Key* column. Variables that only DBX can set, but are available to you for information, are indicated by an **R**; the table below summarizes the notations in the *Key* column.

Key	Description
I	integer
B	boolean
S	ASCII character string
R	reset exclusively and periodically by the debugger

Debugger Variables			
Key	Variable	Default	Description
S	<i>\$addrfmt</i>	"0x%x"	Specifies the format for addresses. You can set this to anything you can format with a C language printf statement.
B	<i>\$casesense</i>	1	Specifies whether source searching and variables are case sensitive. A nonzero value means case sensitive; a 0 means not case sensitive.
I R	<i>\$curevent</i>	none	Shows the last event number as seen in the status feature.
I R	<i>\$curline</i>	none	Shows the current line in the source code.
I R	<i>\$cursrcline</i>	none	Shows the last line you listed plus 1.
	<i>\$curpc</i>		Shows the current address. Used with the <i>wi</i> and <i>li</i> aliases.
B	<i>\$datacache</i>	1	Caches information from the data space so that DBX only has to check the data space once. If you're debugging the operating system, you need to set this variable to 0; otherwise, set it to a nonzero value.
	<i>\$debugflag</i>	0	An internal debug flag used to debug DBX.
S R	<i>\$defaultout</i>	" "	Shows the name of the file that DBX uses to store information when you set the record output command.
S R	<i>\$defaultin</i>	" "	Shows the name of the file that DBX uses to store information when you set the record input command.
	<i>\$defin</i> <i>\$defout</i> <i>\$dispix</i>		Used internally by dbx.

Table 5.2 (1 of 3). Predefined DBX Variables.

Debugger Variables			
Key	Variable	Default	Description
B	<i>\$hexchars</i>	0	Displays hexadecimal when set to a non-zero value. Hexadecimal overrides octal.
B	<i>\$hexin</i>	0	Changes the default input constants to hexadecimal when set to a non-zero value.
B	<i>\$hexints</i>	0	Changes the default output constants to hexadecimal when set to a non-zero value. Hexadecimal overrides octal.
B	<i>\$hexstrings</i>	0	Specifies whether you want to see all strings printed in hexadecimal. A 1 means use hexadecimal; a 0 means use characters.
I R	<i>\$historyevent</i>	none	Shows the current history number.
I	<i>\$lines</i>	20	Specifies the size of your DBX history list.
I	<i>\$listwindow</i>	10	Specifies how many lines the list command lists.
S	<i>\$main</i>	"main"	Specifies the name of the procedure where execution is to begin. DBX by default starts at the procedure <i>main</i> unless you specify otherwise.
I	<i>\$maxstrlen</i>	128	Specifies how many characters of a string that DBX will print for pointers to strings.
B	<i>\$octints</i>	0	Changes the default output constants to octal when set to a non-zero value. Hexadecimal overrides octal.
B	<i>\$octin</i>	0	Changes the default input constants to octal when set to a non-zero value. Hexadecimal overrides octal.
B	<i>\$page</i>	1	Specifies whether to page long information. A nonzero value turns on paging; a 0 turns it off.

Table 5.2 (2 of 3). Predefined DBX Variables.

Debugger Variables			
Key	Variable	Default	Description
I	<i>\$pagewindow</i>	22	Specifies how many lines you see when you look at information that runs longer than one screen. Change this variable to match the number of lines on your terminal. If you set this variable to 0, it assumes a minimum of one line.
B	<i>\$pimode</i>	0	Prints input when you use the playback input command.
B	<i>\$sprintdata</i>	0	Specifies whether to print the value of the registers where you disassemble instructions. A nonzero value means print the register value; a 0 means don't print the value.
B	<i>\$sprintwide</i>	0	Specifies whether you want to print variable values (for example, structures or arrays) in vertical or wide format. A nonzero value means wide format; a 0 means vertical format.
S	<i>\$prompt</i>	"(dbx)"	Sets the prompt for DBX.
B	<i>\$regstyle</i>	1	Specifies the type of register names to be used. A value of 1 specifies hardware names; a 0 specifies software names as defined by the file <i>regdefs.h</i> . This variable does not affect coprocessor register names.
B	<i>\$repeatmode</i>	1	Specifies whether you want DBX to repeat the last command when you press the carriage return. A nonzero value means repeat the last command; a 0 means don't repeat the last command.
B	<i>\$romode</i>	0	Records input when you use the record output command.

Table 5.2 (3 of 3). Predefined DBX Variables.

Creating Command Aliases (alias)

Use the **alias** command to see a list of all current aliases or to define a new alias.

DBX lets you create an alias for any debugger command. Enclose multi-word command names within double or single quotation marks. You can also define a macro as part of an alias, as shown in the example in this section.

DBX has a group of predefined aliases, which you can modify or delete; you can also add your own aliases. You can include all aliases in the *.dbxinit* file if you want to use them in future debugging sections.

For a complete list of predefined aliases, see **Predefined DBX Aliases**.

Syntax:

Command	Select this command to...
<code>alias</code>	See a list of all existing aliases.
<code>alias NAME1 [(ARG1. .ARGN)] "NAME2"</code>	Define a new alias. NAME1 specifies the alias. NAME2 specifies the command. ARG1 ...ARGN let you specify arguments to the alias.

Example:

```
(dbx) alias ok(x) "stop at x"
(dbx) ok(58)
[1] Stop at 58 "sam.c" ← breakpoint set at line 58
(dbx)
```

Removing Command Aliases (unalias)

The **unalias** command removes an alias from a command. You must specify the alias you want to remove; otherwise, you get a syntax error. This effect lasts only for the current debugging session. For a full list of predefined DBX aliases, see **Predefined DBX Aliases**.

Syntax:

Command	Select this command to...
<code>unalias "name"</code>	Remove an alias from a command, where name is the alias name.

Example:

```
(dbx) alias
h      history
si     stepi
Si     nexti
ni     nexti
pi     playback input
ro     record output
ri     record input
a      assign
t      where
j      status
bp     stop in
b      stop at
g      goto
s      step
```

```
More (n if no)?n
```

```
(dbx) unalias h
```

```
(dbx) alias
```

```
si     stepi
Si     nexti
ni     nexti
pi     playback input
ro     record output
ri     record input
a      assign
t      where
j      status
bp     stop in
b      stop at
g      goto
s      step
```

```
More (n if no)?n
```

```
(dbx)
```

the user decides to unalias h from history and it disappears from the list

Predefined DBX Aliases

To list current aliases, use the **alias** command. You can override any predefined alias by redefining it with the **alias** command or by removing it from the list with the **unalias** command. The tables on the following pages shows the debugger predefined aliases.

Debugger Aliases		
Alias	Command	Select this alias to...
?	help	Print a list of all DBX commands.
a	assign	Assign a value to a program variable.
b	stop at	Set a breakpoint at a specified line.
bp	stop in	Stop in a specified procedure.
c	continue	Continue program execution after a breakpoint.
d	delete	Delete the specified item from the status list.
e	file	Look at the specified source file.
f	func	Move to the specified activation level on the stack.
g	goto	Go to the specified line and begin executing the program there.
h	history	List all items currently on your history list.
j	status	See what items are on your status list.
l	list	List the next 10 lines of source code.
n or S	next	Step over the specified number of lines without stepping into procedure calls.
ni or Si	nexti	Step over the specified number of assembly code instructions without stepping into procedure calls.
p	print	Print the value of the specified expression or variable.
pd	printf"%d\n"	Print the value of the specified expression or variable in decimal.

Debugger Aliases		
Alias	Command	Select this alias to...
pi	playback input	Replay DBX commands that you saved with the record input command.
po	printf “%o\n”	Print the value of the specified expression or variable in octal.
pr	printregs	Print values for all registers.
px	printf “%x\n”	Print the value of the specified variable or expression in hexadecimal.
q	quit	End your debugging session.
r	rerun	Run your program again with the same arguments that you specified with the runcommand .
ri	record input	Record every command you type in a file.
ro	record output	Record all debugger output in the specified file.
s	step	Step the next number of specified lines.
si	stepi	Step the next number of specified lines of assembly code instructions.
t	where	Do a stack trace.
u	list \$curline-15:10	List the previous 10 lines.
w	list \$curline-5:10	List the 5 lines preceding and following the current line.
W	list \$curline-10:20	List the 10 lines preceding and following the current line.
wi		List the 5 machine instructions preceding and following the machine instruction.

Recording Input (record input)

Use the **record input** command to record your debugger input. This command provides an excellent means for creating a command file. You can use **record input** with the **source** or **playback input** commands to repeat a sequence of command multiple times. See **Playing Back the Input**.

The syntax of **record input** is shown below.

Syntax:

Command	Select this command to...
<code>record input [filename]</code>	Record everything you type to DBX in a file.

DBX saves the recorded input in *filename*. If you omit *filename*, DBX saves the recorded input in a temporary file, which is deleted at the end of the DBX session. The name of the temporary file is in the system variable *\$defaultin*; you can display the temporary filename using the **print** command as follows:

```
print $defaultin
```

Use the temporary file to repeat previously executed DBX commands only in the current debugging session; specify *filename* to create a command file for use in subsequent DBX sessions. You can see whether you've set **record input** by issuing the **status** command. Use the **delete** command to stop **record input**.

Example:

```
(dbx) record input
[2] record input /tmp/dbxt0013516 (0 lines)
(dbx) status
[1] record input /tmp/dbxt0013516 (0 lines)
(dbx) stop in printline
[2] stop in printline
(dbx) when i = 19 {stop}
[3] traceif i = 19 {stop      }
(dbx)
```

The following is recorded in the temporary file after executing the above commands:

```
status
stop in printline
when i = 19 {stop}
```

Recording Output (record output)

Use the **record output** command to record DBX output during your debugging session. For example, you might want to use this command when you have a large array that doesn't fit the screen. You can record the information in a file and look at it later. If you want to record your input as well, set the DBX variable *\$rimode*. Use the **playback output** command to look at the recorded information, or use any UNIX system editor.

Syntax:

Command	Select this command to...
<code>record output [filename]</code>	Record all DBX output in a file.

DBX saves the recorded output in *filename*. If you omit *filename*, DBX saves the recorded output in a temporary file, which is deleted at the end of the DBX session. The name of the temporary file is in the system variable *\$defaultout*; you can display the temporary filename using the **print** command as follows:

```
print $defaultout
```

Use the temporary file when you need to refer to the saved output only during the current debugging session; specify *filename* to save information required after exiting the current debugging session.

You can see whether you've set **record output** by issuing the **status** command. Use the **delete** command to stop **record output**.

Example:

```
(dbx) record output code ← filename
[3] record output code (0 lines)
(dbx) stop at 25
[4] stop at "sam.c":25
(dbx) run sam.c
[4] stopped at [main:25,8x1b0] if (i<2) {
(dbx)
```

Output:

The above example writes the following output in the file *code*:

```
[3] record output code (0 lines)
(dbx) [4] stop at "sam.c":25
(dbx) [4] stopped at [main:25,0x21b0] if (i<2) {
```

Playing Back the Input (source or playback input)

Use these commands to replay the commands that you recorded with the **record input** command. If you don't specify a filename, DBX uses the current temporary file that it created for the **record input** command. If you set the DBX variable *\$pimode* to 1, the commands are printed out as they are played back.

Syntax:

Command	Select this command to...
<pre> playback input [filename] source [FILE] </pre>	Execute the commands from the specified file.

Example:

```

(dbx) playback input
status
[1] record input /tmp/dbxt0013516 (1 lines)
[2] stop in printline
[3] traceif i = 19 {stop}
stop in printline
[4] stop in printline
when i = 19 {stop}
[5] traceif i=19 {stop }
(dbx)

```

Playing Back the Output (playback output)

This command displays output saved with the **record output** command. The **playback output** command works the same as the UNIX system `cat` command. If you don't specify *filename*, DBX uses the current temporary file created for the **record output** command.

Syntax:

Command	Select this command to...
<pre> playback output [filename] </pre>	Print the commands from the specified file.

Example:

```

(dbx) playback output code ← the file name
[3] record output code (0 lines)
(dbx) [4] stop at "sam.c":25
(dbx) [4] stopped at [main:25,0x1b0] if(i<2){
(dbx)

```

↑
the contents of
the file

Invoking a Shell from DBX (sh)

To invoke a sub-shell, type **sh** at the DBX prompt, or type **sh** and a shell command at the DBX prompt. If you invoke a sub-shell, type **exit** or press **^D** to return to DBX.

Syntax:

Command	Select this command to...
sh	Call a shell from DBX.
sh [SHELL COMMAND]	Execute the shell command.

Example:

```
(dbx) sh
%
% date
Tue Apr 8 17:25:15 PST 1986
% exit
(dbx) sh date
Tue Apr 8 17:29:34 PST 1986
(dbx)
```

← calls a shell

← executes the shell date command

Checking the Status (status)

Use the **status** command to check which, if any, of these commands you currently have set:

- **stop** or **stopi** commands for breakpoints
- **trace** or **tracei** commands for line-by-line variable tracing
- **when** command
- **record input** and **record output** commands for saving information in a file

Syntax:

Command	Select this command to...
status	Check the status of commands.

Example:

```
(dbx) status
[4] trace i in printline
[3] print pline^ at "sam.c":58
[2] stop in printline
[1] record output /tmp/dbxt0018898 (0 lines)
(dbx)
```



the status
item number

Deleting Status Items (delete)

Use the **delete** command to remove items from the status list.

Syntax:

Command	Select this command to...
delete EXP1,...EXPN	Delete the specified status item (EXP) from the status list.
delete all	Delete all status items.

Example:

```
(dbx) status
[4] trace i in printline
[3] print pline^ at "sam.c":58
[2] stop in printline
[1] record output /tmp/dbxt0018898 (0 lines)
(dbx) delete 4
(dbx) status
[3] print pline at "sam.c":58
[2] stop in printline
[1] record output /tmp/dbxt0018890 (0 lines)
(dbx)
```



the status
item number

Examining Source Programs

This section describes how to list and edit source programs while using DBX; it gives details on the DBX commands that allow you to do the following:

- specify source directories
- move to a procedure
- change source files
- list your source code
- search for strings in your source code
- call an editor from DBX
- print symbol names
- print type declarations for variables

Specifying Source Directories (use)

Unless you specified the **-I** option at invocation, DBX looks for source files in the current directory or in the object file's directory. The **use** command lets you change the directory and list the directories currently in use. The command recognizes absolute and relative pathnames (for example, `./`); however, it doesn't recognize the C shell tilde (`~`) syntax—for example, `~john/src`.

Syntax:

Command	Select this command to...
<code>use</code>	List the current directories.
<code>use DIR1 ... DIRN</code>	Specify different directories.

Example:

```
(dbx) use
. ← current directory
(dbx) use /usr/local/lib
(dbx) use
/usr/local/lib ← new directory
(dbx)
```

Moving to a Specified Procedure (func)

The **func** command moves you up or down the activation stack, as appropriate. The function can be a procedure name or an activation level number. To find the name or activation number for a specific procedure, do a stack trace with the **where** command. You can also move through the activation stack by using the **up** and **down** commands. For a definition of activation levels, see **What Are Activation Levels?**

The **func** command changes the current line, the current file, and the current procedure. This changes the scope of the variables you can access. You can use the **func** command when a program isn't executing—for example, when you want only to examine source code.

Syntax:

Command	Select this command to...
<code>func</code>	Print the current activation levels.
<code>func PROCEDURE</code>	Move to the activation level specified by the procedure name.
<code>func EXP</code>	Move to the activation level specified by the expression.

Example:

```
(dbx) where
> 0 printline [pline = 0x7fff5b80) ["sam.c":58, 0x2f7]
  1 $block1 ["sam.c":47, 0x2bb]
  2 main(argc=2, argv=0x7fffeba0) ["sam.c":47, 0x2bb]
(dbx) func 2
main 47 printline(&line1)
(dbx) func main
(dbx)
the activation level
the procedure name
the procedure's arguments
the source file name
the current program counter
the current line
```

Specifying Source Files (file)

The **file** command changes the current source file to a file you specify. The new file becomes the current file, which you can search, list, and perform other operations on.

CAUTION: Before setting a breakpoint or trace, use the **func** command to get the correct procedure, because the **file** command cannot be specific enough for the debugger to access the information necessary to set a breakpoint.

Syntax:

Command	Select this command to...
file	Print the name of the file you're currently using.
file FILE	Change the current file to the specified file.

Example:

```
(dbx) file
sam.c ←————— current file
(dbx) file data.c
(dbx) file
data.c ←————— new file
(dbx)
```

Listing Your Source Code (list)

The `list` command displays lines of source code. The DBX variable `$listwindow` defines the number of lines DBX lists by default. The `list` command uses the current file, procedure, and line unless otherwise specified. It moves the current line forward.

Syntax:

Command	Select this command to...
list	List lines for <code>\$listwindow</code> lines starting at the current line.
list EXP	List the specified line.
list EXP:INT	List the specified number of lines (INT), starting at the specified line (EXP).
list procedure	List the specified procedure for <code>\$listwindow</code> lines.

For example:

```
(dbx) list 53:2 ←————— the user specified
      53          a list starting at
      54 LINETYPE *pline;    line 53 for two lines
(dbx)
```

If you use the predefined alias `w`, (see **Predefined DBX Aliases**), you get something like this:

```
(dbx) w
      53
      54 LINETYPE      *pline;
      55
      56 {
      57 fprintf(stderr, #53d. (%d) %s", pline->linenumber
>* 58 pline->string; ← current line
      59 fflush(stdout);
      60 } /* printline */
(dbx)
```

NOTE: `>` shows the current line and `*` shows the location of the program counter (pc) at this activation level.

Searching Through the Code (/ and ?)

The `/` and `?` commands search for regular expressions in source code. The slash (`/`) searches forward; the question mark (`?`) searches back from the current line. Both commands wrap around the end of the file if necessary, searching the entire file, from the point of invocation back to the same point. If you set the DBX variable `$casesense` on, DBX distinguishes upper-case letters from lower-case.

Syntax:

Command	Select this command to...
<code>/REGEX</code>	Search ahead in the code for the specified regular expression.
<code>?REGEX</code>	Search back in the code for the specified regular expression.

Example:

```
(dbx) /lines
      continue;      /*don't count blank lines */
(dbx) /lines
      line1.length=i
(dbx)
      continue;      /*don't count blank lines */
(dbx)
```

Calling an Editor from DBX (edit)

The `edit` command lets you make changes to your source code from within DBX. For the changes to become effective, you must exit DBX, recompile your program, and, if you want to continue debugging, restart DBX.

Syntax:

Command	Select this command to...
<code>edit</code>	Call an editor from DBX on the current file.
<code>edit [filename]</code>	Call an editor and edit the specified file.

The `edit` command loads the editor that you set as an environment variable `EDITOR`. If you don't set the environment variable, DBX assumes the `vi` editor. When you exit the editor, it returns you to the DBX prompt.

Printing Symbolic Names (which and whereis)

The `which` and `whereis` commands print program variables as described below. These commands are useful for programs that have multiple variables with the same name, occurring in different scopes. The commands follow the rules described in the section **Qualifying Variable Names**.

Syntax:

Command	Select this command to...
<code>which VAR</code>	Print the default version of the variable.
<code>whereis VAR</code>	Print all versions of the specified variable.

Example:

```
(dbx) which i
sam.main.i
(dbx) whereis i
sam.println.i sam.main.$block1.i sam.main.i
(dbx)
```

Printing Type Declarations (whatis)

The `whatis` command lists the type declaration for variables and procedures in your program.

Syntax:

Command	Select this command to...
<code>whatis VAR</code>	Print the type declaration for the specified variable or procedure in your program.

Example:

```
(dbx) whatis main
int main(argc, argv)
int argc;
unsigned char **argv;
(dbx) whatis i
int i;
(dbx)
```

Controlling Your Program

The commands in this section control program execution by performing the following functions:

- run and rerun a program
- step through a program one line at a time
- return from a procedure call
- start at a specified line
- continue after a breakpoint
- assign values to program variables

These functions are described in detail in the sections that follow.

Running Your Program (run and rerun)

The **run** or **rerun** starts program execution. You can specify arguments to either command. Arguments to these commands override previous arguments; however, if you don't specify arguments to the *rerun* command, it uses the last set of arguments. If you don't specify arguments to the *run* command, it runs the program without arguments.

These commands can be used to redirect program input and output. This feature works like redirection in the C shell. The optional parameter *<FILE1* redirects input to your program from the specified file. *>FILE2* redirects output from the program to the specified file. The optional parameter *>&FILE2* redirects stderr and stdout output to the specified file.

NOTE: This output differs from the output you save with the **record output** command. That command saves debugger (not program) output in a file. See **Recording the Output (record output)**.

Syntax:

Command	Select this command to...
run [ARG1, ... ARGN] [<FILE1] [>FILE2] run [ARG1, ... ARGN] [<FILE1] [>&FILE2]	Run your program with the specified arguments.
rerun [ARG1...ARGN] [<FILE1] [>FILE2] rerun [ARG1...ARGN] [<FILE1] [>&FILE2]	Rerun your program with the arguments you specified to the run command or with new arguments.

The arguments to the *run* command specify any program arguments that your program might have.

Example:

```
(dbx) run sam.c ← the argument is sam.c
0. (19) #include<stdio.h>
1. (14) struct line {
2. (22) char string[256];
...

```

Example:

```
(dbx) rerun
0. (19) #include<stdio.h>
1. (14) struct line {
2. (22) char string[256];
.
.
.
program terminated normally
(dbx)

```

Executing Single Lines of Your Code (step and next)

The **step** and **next** commands execute a fixed number of source code lines as specified by EXP. If you don't specify EXP for **step** and **next**, DBX executes one source code line. If you specify EXP, DBX executes the source code lines as follows:

- For **step** and **next**, DBX does not take comment lines into consideration in interpreting EXP. The program executes EXP source code lines, regardless of the number of comment lines interspersed among them.
- For **step**, DBX considers EXP to apply to *both* the current procedure *and* to called procedures. The program stops after executing EXP source lines in the current procedure and any called procedures.

- For **next**, DBX considers EXP to apply to *only* the current procedure. The program stops after executing EXP source lines in the current procedure, regardless of the number of source lines executed in any called procedures.

Use **step** and **next** to execute source lines after a breakpoint.

Syntax:

Command	Select this command to...
<code>step [EXP] *</code>	Execute the specified number of lines of source code. EXP refers to the number of lines to be executed in <i>both</i> the current procedure and any called procedures.
<code>next [EXP] *</code>	Execute the specified number of lines of source code. EXP refers to the number of lines to be executed in <i>only</i> the current procedure, regardless of any called procedures executed.
*Default is 1.	

Example:

```
(dbx) rerun
[3] stopped at [printline:58,0x2f8] pline->string);
(dbx) step 2
0 (19) #include <stdio.h>
[$block1:48,0x2bc] } /*while*/
(dbx) step
[$block1:41,0x260] i=strlen(line1.string);
(dbx)
```

↑
**\$block1 gets created
because it defines the
scope for its own local
variables**

Returning from a Procedure Call (return)

The **return** command is for use in a called procedure when stepping through the code one instruction at a time. The command causes the remaining instructions in the procedure to be executed and the program to stop at the first instruction on return from that procedure.

Syntax:

Command	Select this command to...
return	Execute the current procedure and return to the next sequential line in the calling procedure.
return PROCEDURE	Execute the program until DBX returns to the specified procedure.

Example:

```
(dbx) rerun
[6] stopped at [println:58, 0x2f8] pline->string);
(dbx) return
0 (19) #include <stdio.h>
stopped at [$block1:48,0x2bc] } /*while*/
(dbx)
```

Starting at a Specified Line (goto)

The **goto** command shifts program execution to a line you specify. This command is useful in a **when** statement—for example, to skip a line that you know has problems.

Syntax:

Command	Select this command to...
goto LINE	Go to a specified line and continue execution.

Example:

```
(dbx) when at 58 {goto 43}
[1] start "sam.c":48 at "sam.c":58
(dbx)
```

Continuing after a Breakpoint (cont)

The **cont** command resumes program execution after a breakpoint. The **cont to** and **cont in** are in effect at the time you issue them; when your program again reaches a breakpoint, you can reissue either **cont to** or **cont in** to continue, if desired. If **SIGNAL** is specified as a parameter (see below), DBX sends the specified signal to the program and continues.

Syntax:

Command	Select this command to...
cont	Continue from the current line.
cont to LINE	Continue until the specified line.
cont in PROCEDURE	Continue until the specified procedure.
cont SIGNAL	Continue from the current line and send the signal.
cont SIGNAL to LINE	Continue until reaching the specified line and send the signal.
cont SIGNAL in PROCEDURE	Continue until reaching the specified procedure and send the signal.

Example:

```
(dbx) stop in printline
[1] stop in printline
(dbx) rerun
[1] stopped at [printline:58,0x2f8] pline->string);
(dbx) cont
    0 (19)#include <stdio.h>
[1] stopped at [printline:58,0x2f8] pline ->string);
(dbx)
```

Assigning Values to Program Variables (assign)

The **assign** command changes the value of existing program variables.

Syntax:

Command	Select this command to...
assign EXP1 = EXP2	Assign a new value to a program variable.

Example:

```
(dbx) print i
19 ←———— the value of i
(dbx) assign i = 10
10 ←———— the new value of i
(dbx) assign *($integer*)0x455 = 1 ←———— coerce the
1                                     address to be
(dbx)                                     an integer
                                           and assign a
                                           1 to it
```

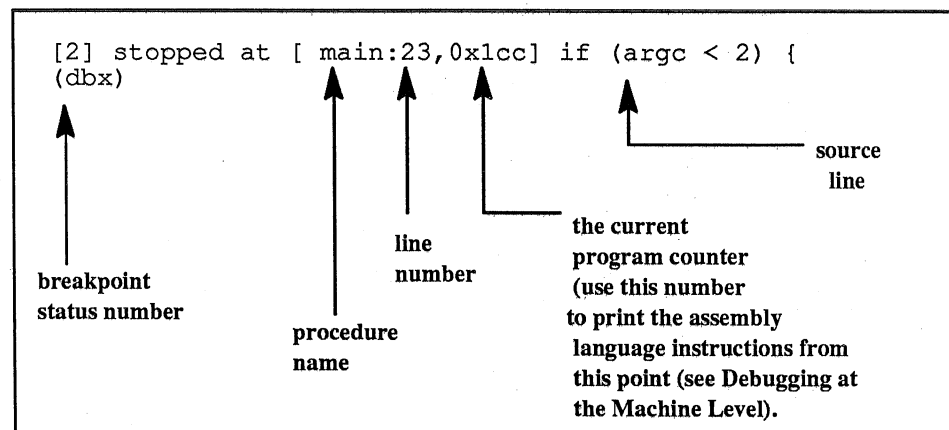
Setting Breakpoints

A breakpoint stops program execution and lets you examine the program's state at that point. This part of **Chapter 5** shows you how to do the following:

- set breakpoints at lines
- set breakpoints in procedures
- continue running your program after a breakpoint
- stop for signals

Overview

When a program stops at a breakpoint, the debugger displays an informational message. For example, if you set a breakpoint in the sample program *sam.c* (see **Sample Program** at the end of the chapter) at line 23 in the main procedure, you get this message:



Before setting a breakpoint in a program that has multiple files, be sure that you're setting the breakpoint in the right file.

To select the right procedure, follow these steps:

1. Use the **func** command and specify a procedure name. This command moves you to the file that contains the specified procedure. See **Controlling Your Program**.
2. List the lines of the procedure. Use the **list** command. See **Controlling Your Program**.
3. When you see the procedure or line you want, use a **stop** command to set a breakpoint.

Setting Breakpoints at Lines (stop at)

The **stop at** command sets a breakpoint at a specific line. DBX stops only at lines that have executable code. If you specify an unexecutable line, DBX sets the breakpoint at the next executable line. If you specify the VAR parameter,

the debugger prints the variable and stops only when VAR changes; if you specify **if EXP**, DBX stops only when EXP is true.

Syntax:

Command	Select this command to...
stop [VAR] at	Stop at the current line.
stop [VAR] at LINE	Stop at a specified line.
stop [VAR] at LINE if EXP	Stop at a specified line only if the expression is true.

NOTE: if EXP is checked before VAR.

Example:

```
(dbx) stop at 58
[16] stop at "sam.c":58
(dbx) rerun
[16] stopped at [printline:58,0x2f8] pline->string);
(dbx)
```

the procedure name the line number the current program counter

Setting Breakpoints in Procedures (stop in)

The **stop in** command sets a breakpoint at the beginning or, conditionally, for the duration of a procedure.

Syntax:

Command	Select this command to...
stop in PROCEDURE	Stop at the beginning of the procedure.
stop VAR in PROCEDURE	Stop in the specified procedure when VAR changes.
stop in PROCEDURE if EXP*	Stop in the specified procedure if EXP is true.
stop VAR in PROCEDURE if EXP*	Stop in the specified procedure when VAR changes and EXP is true.
*EXP is checked before VAR.	

NOTE: Specifying both VAR and EXP causes stops *anywhere* in the procedure, not just at the beginning. Using this feature is expensive, because the

debugger must check the condition before and after each source line is executed.

Example:

```
(dbx) stop in printline
[15] stop in printline
(dbx) rerun
[15] stopped at [printline:58,02f8] pline->string);
(dbx)
```

Setting Conditional Breakpoints (stop if)

The **stop if** command causes DBX to stop program execution upon encountering a condition you specify. Because DBX must check after the execution of each line, this command is expensive and slows program execution markedly. Whenever possible, use **stop at** or **stop in** instead of **stop if**.

Syntax:

Command	Select this command to...
stop if EXP	Stop if EXP is true.
stop VAR if EXP	Stop if VAR changes <i>and</i> EXP is true.
EXP is checked before VAR.	

NOTE: EXP is checked before VAR.

Tracing Variables (trace)

The **trace** commands list the value of a variable during program execution as well as determine the scope for the variables that you're tracing.

Syntax:

Command	Select this command to...
trace VAR	List the specified variable after each source line is executed.
trace VAR at LINE	List the specified variable at the specified line.
trace VAR in PROCEDURE	List the specified variable in the specified procedure.
trace VAR at LINE if EXP	List the variable at the specified line when the expressions is true.
trace VAR in PROCEDURE if EXP	List the variable in the specified procedure when the expression is true.

NOTE: EXP is checked before VAR.

Example:

```
(dbx) trace i
[15] trace i in $block1
(dbx) rerun
[printline:58,0x2f8]:i=19
[23] [printline:58,0x2f8] pline->string);
    0 ( 19) #include<stdio.h>
[25] i changed before ["sam.c":41]:
        old value = 19;
        new value = 1;
[25] i changed before ["sam.c":41]:
        old value = 1;
        new value = 14;
[printline:58,0x2f8]: i=14
[23] [printline:58,0x2f8] pline->string);
    1. ( 14) struct line {
[25] i changed before ["sam.c":41]:
        old value = 14;
        new value = 22;
More (n if no)n
Escape from listing
(dbx)
```

Writing Conditional Code in DBX (when)

The **when** command is for use with conditional debugger code. It lets you specify a condition (for example, a line number, an expression, or a procedure name). When DBX encounters that condition, it executes the commands that you specified.

Syntax:

Command	Select this command to...
catch	Print a list of all signals that DBX will catch.
catch SIGNAL	Add a signal to the catch list.
ignore	Print a list of all signals that DBX will not catch.
ignore SIGNAL	Remove a signal from the catch list and add it to the ignore list.

Example:

```
(dbx) catch
INT QUIT ILL TRAP IOT EMT FPE BUS SEGV SYS PIPE TERM STOP TTIN
TTOU TINT SCPU XFSZ
(dbx) ignore
HUP KILL ALRM TSTP CONT CHLD
(dbx) catch kill
(dbx) catch
INT QUIT ILL TRAP IOT EMT FPE KILL BUS SEGV SYS PIPE TERM STOP
TTIN TTOU TINT XCPU XFSZ
(dbx) ignore
HUP ALRM TSTP CONT CHLD
(dbx)
```

↑
**removes KILL
from the ignore
list**

↑
**adds KILL
to the catch
list**

Examining Program State

This part of **Chapter 5** shows you how to do these things:

- print stack traces
- move up and down the activation levels of the stack
- print variable values
- print register values
- print information about the activation levels in the stack trace

Doing Stack Traces (where)

The **where** command does stack traces. Stack traces show the current activation levels (procedures) of a program. This command doesn't trace variables.

Syntax:

Command	Select this command to...
where	Do a stack trace.

Example:

If a breakpoint is set in *printline* in the sample program *sam.c.*, (see **Sample Program** at the end of this chapter), the program runs and stops in the procedure *main*. Then, by typing **cont** followed by **where**, a stack trace executes, providing the information shown below.

```
(dbx) stop in printline
[1] stop in printline
(dbx) where
>0 printline(pline = 0x7fff5b80) ["sam.c":58,0x2f7]
1 $block1["sam.c":47,0x2bb]
2 main(argc = 2, argv = 0x7fffeba0) ["sam.c":47,0x2bb]
(dbx)
```

the activation level number—the > shows that the user is examining this activation level

the procedure name

the current value of the argument pline

the source file name

the line number

the program counter

NOTE: In the example, *\$block1* has the same program counter as *main*. This indicates that *main* has a block with locally scoped variables in it, which doesn't appear to all of *main*.

Moving Up and Down the Stack (up, down)

The **up** and **down** commands move up and down the activation levels in the stack. These commands are useful when examining a call from one level to another. You can also move up and down the activation stack with the **func** command. For a definition of activation levels, see **What Are Activation Levels?**

Syntax:

Command	Select this command to...
up [EXP]	Move up the specified number of activation levels in the stack. The default is one level.
down [EXP]	Move down the specified number of activation levels in the stack. The default is one level.

Example:

```
(dbx) where
>0 printline(pline = 0x7fff5b80) ["sam.c":58,0x2f7]
  1 $block1["sam.c":47,0x2bb]
  2 main(argc = 2, argv = 0x7fffeba0) ["sam.c":47,0x2bb]
(dbx) down
$block1 ["sam.c":47,0x2bb]
(dbx) where
  0 printline(pline = 0x7fff5b80) ["sam.c":58,0x2f7]
>1 $block1["sam.c":47,0x2bb]
  2 main(argc = 2, argv = 0x7fffeba0) ["sam.c":47,0x2bb]
(dbx) up
printline(pline = 0x7fff5b80) ["sam.c":58,0x2f7]
(dbx) where
>0 printline(pline = 0x7fff5b80) ["sam.c":58,0x2f7]
  1 $block1["sam.c":47,0x2bb]
  2 main(argc = 2, argv = 0x7fffeba0) ["sam.:47,0x2bb]
(dbx)
```

Annotations in the example:

- Next to the second `$block1` line: **moves down one level** (with an arrow pointing to the line)
- Next to the first `printline` line in the second `where` block: **moves up one level** (with an arrow pointing to the line)
- Next to the first `printline` line in the third `where` block: **moves up one level** (with an arrow pointing to the line)

Printing (print and printf)

The **print** command lists the value of one or more expressions. You can also use **print** to display the program counter and the current value of registers; see the next section, **Printing Register Variables**, for details.

The **printf** command lists information in a format you specify and supports all formats of the **PRINTF** command except `%s`. For a full list of formats, see the **PRINTF(3S)** manual page in the *UNIX Programmer's Manual*. You might use **printf** when you want to see a variable's value in a different number base. The command alias list has some useful aliases for printing the value of variables in different bases—octal (**po**), decimal (**pd**), and hexadecimal (**px**). The default number base is decimal. See **Creating Command Aliases**.

Syntax:

Command	Select this command to...
<code>print EXP1, ..., EXPN</code>	Print the value of the specified expressions.
<code>printf "STRING", EXP1, ..., EXPN</code>	Print the value of the specified expressions in the format as specified by the string.

Note: If the expression has the same name as a dbx keyword, it must be enclosed within parentheses. For example, in order to print *output*, a keyword in the playback and record commands, you must specify:

```
print (output)
```

Example:

(dbx) print i		
14 ←	decimal	
(dbx) pd i		
14 ←	decimal	
(dbx) po i		
016 ←	octal	
(dbx) px i		
0xe ←	hexadecimal	
(dbx)		

Printing Register Values (printregs)

The `printregs` command prints register values, both the real machine register names and the software (from the include file `regdefs.h`) names. A prefix before the register number specifies the type of register; the prefixes used and their meanings are shown in the table below:

Prefix	Register Type
\$r	Machine register.
\$f	Floating point.
\$d	Double precision floating point.
\$pc	Program counter value..

You can also specify prefixed registers in the print command to display a register value or the program counter. For example,

```
print $r3
print $pc
```

print the values of machine register 3 and the program counter respectively.

You set the DBX variables \$hexints and \$hexouts to specify that the listing use hexadecimal.

Syntax:

Command	Select this command to...
printregs	Print the current values of all registers.

Example:

```
(dbx) printregs
r0/zero=0          r1/at=1           r2/v0=19          r3/v1=0
r4/a0=2147441472  r5/a1=34838       r6/a2=4096        r7/a3=80
r8/t0=19           r9/t1=34816       r10/t2=19         r11/t3=0
r12/t4=1           r13/t5=34820      r14/t6=0          r15/t7=1
r16/s0=2147441472 r17/s1=0          r18/s2=0          r19/s3=0
r20/s4=0           r21/s5=0          r22/s6=0          r23/s7=0
r24/t8=4086        r25/t9=255        r26/k0=0          r27/k1=0
r28/gp=50529       r29/s0=2147441400 r30/fp=2147442536 r31/ra=700
$f0= 0.0           $f1= 0.0          $f2= 0.0          $f3= 0.0
$f4= 0.0           $f5= 0.0          $f6= 0.0          $f7= 0.0
$f8= 0.0           $f9= 0.0          $f10=0.0          $f11=0.0
$f12=0.0           $f13=0.0          $f14=0.0          $f15=0.0
$f16=0.0           $f17=0.0          $f18=0.0          $f19=0.0
$f20=0.0           $f21=0.0          $f22=0.0          $f23=0.0
$f24=0.0           $f25=0.0          $f26=0.0          $f27=0.0
$f28=0.0           $f29=0.0          $f30=0.0          $f31=0.0
$d0= 0.0           $d2= 0.0          $d4= 0.0          $d6= 0.0
$d8= 0.0           $d10=0.0          $d12=0.0          $d14=0.0
$d16=0.0           $d18=0.0          $d20=0.0          $d22=0.0
$d24=0.0           $d26=0.0          $d28=0.0          $d30=0.0
$pc= 760
(dbx)
```

Printing Information about Activation Levels (dump)

The **dump** command prints information about activation levels. For example, this command prints values for all variables local to a specified activation level. To see what activation levels you have in your program, use the **where** command to do a stack trace.

Syntax:

Command	Select this command to...
dump	Print information about the current activation level.
dump .	Print information about all activation levels in your program.
dump PROCEDURE	Print information about the specified procedure (activation level).

Example:

```
(dbx) where
>0 printline (pline=0x7fff5b80) ["sam.c":58,0x2f7]
  1 $block1 ["sam.c":47,0x2bb]
(dbx) dump
printline (pline=0x7fff5b80) ["sam.c":58,0x2f7]
(dbx) dump .
> 0 printline (pline=0x7fff5b80) ["sam.c":58,0x2f7]
  1 $block1 ["sam.c":47,0x2bb]
  curlinenum = 1
  i=19
    2 main (argc=2,argv=0x7fffeba0) ["sam.c":47,0x2bb]
  fd = 0x4270
  line1=struct {
  string="#include<stdio.h>
  "
  linenum=0
  }
  in "";
(dbx) dump main
main (argc=2, argv=0x7fffeba0) ["saam.c":47,0x2bb]
fd=0x4270
line1=struct {
string="struct line {
length = 14
linenum = 1
}
(dbx)
```

Debugging at the Machine Level

DBX provides some commands specifically for people who need to debug assembly code. The commands described in this section let you do the following:

- set breakpoints
- execute single lines of code
- trace variables
- print the contents of memory addresses
- disassemble instructions

Setting Breakpoints in Machine Code (stopi)

The **stopi** commands set breakpoints in machine code. These commands work in the same way as the **stop at**, **stop in**, and **stop if** commands as described in the section **Setting Breakpoints**, except for the **stop at** command, where an address instead of a line number is specified.

Command	Select this command to...
<code>stopi [VAR] at</code>	Stop at the current line.
<code>stopi [VAR] at LINE</code>	Stop at a specified address.
<code>stopi [VAR] at LINE if EXP</code>	Stop at a specified address only if EXP is true.
<code>stopi if EXP</code>	Stop if EXP is true.
<code>stopi VAR if EXP</code>	Stop if VAR changes and EXP is true.
<code>stopi in PROCEDURE</code>	Stop at the beginning of the procedure.
<code>stopi VAR in PROCEDURE</code>	Stop in the specified procedure when VAR changes.
<code>stopi in PROCEDURE if EXP *</code>	Stop in the specified procedure if EXP is true.
<code>stopi VAR in PROCEDURE if EXP*</code>	Stop in the specified procedure when VAR changes and EXP is true.
* EXP is checked before VAR.	

Example:

```
(dbx) stopi at 0x2f8
[2] stopi at "sam.c":760
(dbx) rerun
[2] stopped at [printline:58,0x2f8] pline-> string);
(dbx)
```

Continuing after Breakpoints in Machine Code (conti)

The **conti** commands continue executing assembly code after a breakpoint.

Syntax:

Command	Select this command to...
<code>conti SIGNAL</code>	Send the specified signal and continue.
<code>conti to ADDRESS</code>	Continue until reaching the specified address.
<code>conti in PROCEDURE</code>	Continue until the beginning of the specified procedure.
<code>conti SIGNAL to ADDRESS</code>	Continue until reaching the specified address, then send the signal.
<code>conti SIGNAL in PROCEDURE</code>	Continue until reaching the beginning of the specified procedure, then send the signal.

Example:

```
(dbx) conti
0 (19)#include <stdio.h>
[2] stopped at [printline:58,0x2f8] pline->string0;
lw r2,32(sp)
(dbx)
```

Executing Single Lines of Machine Code (steppi and nexti)

The **steppi** and **nexti** commands execute a fixed number of machine instructions as specified by EXP. If you don't specify EXP for **steppi** and **nexti**, DBX executes one machine instruction. If you specify EXP, DBX executes the machine instructions as follows:

- For **steppi** and **nexti**, DBX does not take comment lines into consideration in interpreting EXP. The program executes EXP machine instructions, regardless of the number of comment lines interspersed among them.
- For **steppi**, DBX considers EXP to apply to *both* the current procedure *and* to procedure calls (**jal** and **jalr**). The program stops after executing EXP instructions in the current procedure and any called procedures.

- For **nexti**, DBX considers EXP to apply to *only* the current procedure. The program stops after executing EXP instructions in the current procedure, regardless of the number of instructions executed in any procedure calls.

Use **stepi** and **nexti** to execute source lines after a breakpoint.

Syntax:

Command	Select this command to...
<code>stepi [EXP] *</code>	Execute the specified number of lines of source code. EXP refers to the number of lines to be executed in <i>both</i> the current procedure <i>and</i> any procedure calls.
<code>nexti [EXP] *</code>	Execute the specified number of lines of source code. EXP refers to the number of lines to be executed in <i>only</i> the current procedure, regardless of any procedure calls.
*Default is 1.	

Example:

```
(dbx) rerun
[2] stopped at [printline:58,0x2f8] pline->string);
(dbx) stepi
[printline:58+0x4,0x2fc] pline->string);
lui r1,0x0
(dbx)
```

Tracing Variables in Machine Code (tracei)

The **tracei** commands track, one instruction at a time, changes to variables. The **tracei** commands work for machine instruction as the **trace** commands do for lines of source code.

Syntax:

Command	Select this command to...
<code>tracei VAR</code>	Print the value of the variable as it changes instruction by instruction.
<code>tracei VAR at ADDRESS</code>	Print the value of the variable when it changes instruction by instruction.
<code>tracei VAR in PROCEDURE</code>	Print the value of the variable when it changes in the specified procedure.
<code>tracei VAR at ADDRESS if EXP</code>	Print the value of the variable at the specified address when the expression is true.
<code>tracei VAR in PROCEDURE if EXP</code>	Print the value of the variable in the specified procedure when it traces instruction by instruction.

Printing the Contents of Memory

Entering values in the syntax shown below prints the contents of memory according to your specifications.

Syntax:

Command	Select this command to...
<code>ADDRESS /<COUNT><MODE></code>	Print the contents of the specified address for the specified count.

The parameter *address* is the address of the first item to be listed, *count* is the number of items to be listed, and *mode* is one of the characters shown in the table below specifying the item.

Mode	Select this mode to...
d	Print a short word in decimal.
D	Print a long word in decimal.
o	Print a short word in octal.
O	Print a long word in octal.
x	Print a short word in hexadecimal.
X	Print a long word in hexadecimal.
b	Print a byte in octal.
c	Print a byte as a character.
s	Print a string of characters that ends in a null byte.
f	Print a single precision real number.
g	Print a double precision real number.
i	Print machine instructions.

Example:

When you look at memory, you will see something like this:

```
(dbx) 0x2f8/10i
[printline:58,0x2f8] lw    r2,32(sp)
[printline:58,0x2fc] lui   r1,0x0
[printline:58,0x300] addiu r4,r1,16860
[printline:58,0x304] lui   r1,0x0
[printline:58,0x308] addiu r5,r1,16780
[printline:58,0x30c] lw    r6,260(r2)
[printline:58,0x310] lw    r7,256(r2)
[printline:58,0x314] jal   fprintf!!
[printline:58,0x318] sw    r2,16(sp)
[printline:59,0x31c] lui   r1,0x0
[printline:59,0x320] jal   fflush<!
[printline:59,0x324] addiu r4,r1,16960
(dbx) 0x2f8/10d
      000002f8: 32 3677 0 0 15361 1690 9252 0 15361
      00000308: 16780 9253
(dbx)
```

When you disassemble instructions, you will see something like this:

```
(dbx) &printline/10i
*[printline:58,0x2ec] addiu    sp, sp, -32
[printline:58,0x2f0] sw      f4, 32(sp)
[printline:58,0x2f4] sw      r31, 28(sp)
[printline:58,0x2f8] lw      r2, 32(sp)
[printline:58,0x2fc] lui     r1, 0x0
[printline:58,0x300] addiu   r4, r1, 16960
[printline:58,0x304] lui     r1, 0x0
[printline:58,0x308] addiu   r5, r1, 16780
[printline:58,0x30c] lw      r6, 260(r2)
[printline:58,0x310] lw      r7, 256(r2)
(dbx) &printline/10d
000002ec: 65504 10173 32 44963 23 15361
          4496428 44991 32 36770
000002fc:0
(dbx)
```

Debugger Command Summary

Table 5.3 lists all commands (except for command line editing commands) and gives each command's syntax. For a full description of each command, refer to the command's long description.

DBX Command Summary			
Command	Alias	Select this command to...	Syntax
/		Search ahead in the code for the specified string.	/REGEX
?		Search back in the code for the specified string.	?REGEX
!		Specify a command from your history list.	!STRING !INT !-INT
alias		List all existing aliases, or, if you specify an argument, to define a new alias.	alias [NAME (ARG1,... ARGN) "STRING"]
assign	a	Assign the specified expression to a specified program variable.	assign EXP1 = EXP2
catch		List all signals that DBX catches, or, if you specify an argument, to add a new signal to the catch list.	catch [signal]
cont	c	Continue executing a program after a breakpoint.	cont cont in PROCEDURE cont to LINE cont SIGNAL to LINE cont SIGNAL in PROCEDURE

Table 5.3 (1 of 7). Command Summary.

DBX Command Summary			
Command	Alias	Select this command to...	Syntax
conti		Continue executing assembly code after a breakpoint.	<code>conti SIGNAL</code> <code>conti to ADDRESS</code> <code>conti in PROCEDURE</code> <code>conti SIGNAL to ADDRESS</code> <code>conti SIGNAL in PROCEDURE</code>
delete	d	Delete the specified item from the status list.	<code>delete EXP1,...EXPN</code> <code>delete ALL</code>
down		Move down the specified number of activation levels in the stack. The default is one level.	<code>down [EXP]</code>
dump		Print variable information about the procedure. If you specify a dot (.), this command prints global variable information for all procedures.	<code>dump PROCEDURE</code> <code>dump .</code>
edit		Call an editor from DBX.	<code>edit [FILE]</code>
file	e	Print the name of the current file, or, if you specify a filename, this command changes the current file to the specified file.	<code>file [FILE]</code>
func	f	Move to the specified procedure (activation level) or print the current activation level.	<code>func</code> <code>func EXP</code> <code>func PROCEDURE</code>
goto	g	Go to the specified line.	<code>goto LINE</code>

Table 5.3 (2 of 7). Command Summary.

DBX Command Summary			
Command	Alias	Select this command to...	Syntax
help	?	Print a list of DBX commands. Uses UNIX system more .	<code>help</code>
history	h	Print a list of the previous commands that you've issued. The default is 20.	<code>history</code>
ignore		List all signals that DBX does not catch, or, if you specify an argument, add the specified signal to the ignore list.	<code>ignore [SIGNAL]</code>
list	l	List the specified lines. The default is 10 lines.	<code>list</code> <code>list [EXP:INT]</code> <code>list [EXP]</code>
next	n	Step over the specified number of lines. The default is one. This command does not step into procedures.	<code>next [INT]</code>
nexti	ni	Step over the specified number of machine instructions. The default is one. This command does not step into procedures.	<code>nexti [INT]</code>
playback input	pi	Replay commands that you saved with the record input command in a text file.	<code>playback input [FILE]</code>

Table 5.3 (3 of 7). Command Summary.

DBX Command Summary			
Command	Alias	Select command to...	Syntax
playback output	po	Replay debugger output that you saved with the record output command.	playback output [FILE]
print	p	Print the value of the specified expression.	print EXP1,...,EXPN
printf	pd	Print the value of the specified expression, using C string formatting.	printf "STRING", EXP1,...EXPN
printregs	pr	Print all register values.	printregs
quit	q	Exit DBX.	quit
record input	ri	Record all commands you type to DBX.	record input [FILE]
record output	ro	Record all DBX output.	record output [FILE]
return		Continue executing until the procedure returns. If you don't specify a procedure, DBX assumes the next procedure.	return [PROCEDURE]
run		Run your program.	run [ARG1 ... ARGN] [<FILE1] [>FILE2]
rerun	r	Run your program again, using the same arguments you specified to the run command.	rerun [ARG1 ... ARGN] [<FILE1] [>FILE2]

Table 5.3 (4 of 7). Command Summary.

DBX Command Summary			
Command	Alias	Select command to...	Syntax
set		See the list of existing debugger variables and their values, assign a value to a variable, or define a new variable and assign a value to it.	set set VAR = EXP
sh		Call a shell from DBX, or, execute a shell command.	sh [SHELL COMMAND]
source		Execute DBX commands from the specified file. If you don't specify a file name, DBX assumes that you want the file you created with the record inputcom- mand.	source [FILE]
status	j	Print a list of currently set breakpoints, record commands, and traces.	status
step	s	Step the specified number of lines. This command steps into procedures. The default is one line.	step [INT]
stepi	si	Step the specified number of machine instructions. This command steps into procedures. The default is one instruction.	stepi [INT]
stop	b bp	Set a breakpoint at the specified point.	stop [VAR] at stop [VAR] at LINE stop [VAR] in PROCEDURE stop [VAR] if EXP stop [VAR] at LINE if EXP stop [VAR] in PROCEDURE if EXP

Table 5.3 (5 of 7). Command Summary.

DBX Command Summary			
Command	Alias	Select command to...	Syntax
stopi		Set a breakpoint in machine code at the specified point.	<pre>stopi [VAR] at ADDRESS stopi [VAR] in PROCEDURE stopi [VAR] if EXP stopi [VAR] at ADDRESS if EXP stopi [VAR] in PROCEDURE if EXP</pre>
trace	tr	Trace the specified variable.	<pre>trace VAR trace VAR at LINE trace VAR in PROCEDURE trace VAR at LINE if EXP trace VAR in PROCEDURE if EXP</pre>
tracei		Trace the specified variable in the machine instruction.	<pre>tracei VAR tracei VAR at ADDRESS tracei VAR in PROCEDURE tracei VAR at ADDRESS if EXP tracei VAR IN PROCEDURE if EXP</pre>
unalias		Remove specified alias.	<pre>unalias ALIAS NAME</pre>
unset		Unset a debugger variable.	<pre>unset VAR</pre>
up		Move the specified number of activation levels up the stack. The default is one.	<pre>up [EXP]</pre>
use		Print a list of the source directories, or, if you specify a directory name, this command substitutes the new directories for the previous list.	<pre>use [DIR1 DIR2...DIRN]</pre>

Table 5.3 (6 of 7). Command Summary.

DBX Command Summary			
Command	Alias	Select command to...	Syntax
whatis		Print the type declaration for the specified name.	<code>whatis VAR</code>
when		Execute the specified DBX commands during execution.	<pre>when [VAR] [if EXP] {COMMAND_LIST} when [VAR] at LINE [if EXP]{COMMAND_ LIST} when [VAR] in PROCEDURE [if EXP]{COMMAND_ LIST}</pre>
where	t	Do a stack trace, which shows the current activation levels.	<code>where</code>
whereis		Print all qualifications of the specified variable name.	<code>whereis VAR</code>
which		Print the qualification of the variable name currently in use.	<code>which VAR</code>
examine address		Print the contents of the specified address or disassemble the code for the instruction at the specified address.	<code>ADDRESS/<COUNT><MODE></code>

Table 5.3 (7 of 7). Command Summary.

Sample Program

In this chapter, we use a sample program to illustrate debugger commands. The program *sam.c* counts non-blank lines in a program.

The sample C program, *sam.c*, consists of the code shown on the next page.

```

#include <Stdio.h>

struct line {
    char        string[256];
    int         length;
    int         linenumber;
};

typedef struct line LINETYPE;
void printline();
main (argc, argv)
int     argc;
char    **argv;
{
    LINETYPE    line1;
    FILE        *fd;
    extern FILE *fopen();
    extern char *fgets();
    if (argc < 2) {
        fprintf (stderr, "Usage sam filename\n");
        exit (1);
    } /* if */
    fd = fopen (argv[1], "r");
    if (fd == NULL) {
        fprintf (stderr, "cannot open %s\n", argv[1]);
        exit (1);
    } /* if */
    /* loop through the lines in a file and call a routine to print
       those that are not blank along with their lengths and linenumber.
    */
    while (fgets (line1.string, sizeof (line1.string), fd) != NULL) {
        int     i;
        static  curlinenumber = 0;
        i = strlen (line1.string);
        if (i == 1 && line1.string[0] == '\n')
            continue;          /* don't count blank lines */
        line1.length = i;
        line1.linenumber = curlinenumber++;
        printline (&line1);
    } /* while */
} /* main */

void printline (pline)
LINETYPE    *pline;
{ fprintf (stdout, "%3d. (%3d) %s",
    pline->linenumber pline->length,
    pline->string);
    fflush (stdout);
} /* printline */

```


Appendix A

C Implementation

The C language supported by the compiler is an implementation of the language defined in *The C Programming Language* by Kernighan and Ritchie (Prentice Hall, 1978). This appendix covers the following topics:

- Specifying vararg macros, a requirement for all functions that take a variable number of arguments.
- Deviations and extensions to the C language, as defined in *The C Programming Language* by Kernighan and Ritchie (Prentice-Hall).
- Translation limits.

Vararg.h Macros

If a function takes a variable number of arguments (for example, the C library functions **printf** and **scanf**), you must use the macros (defined in the `varargs.h` header file) shown below.

```
/*  @(#)varargs.h  1.2  */
typedef char *va_list;
#define va_dcl int va_alist;
#define va_start(list) list = (char *) &va_alist
#define va_end(list)
#define va_arg(list, mode) ((mode *) (list = \
    (char *) (sizeof(mode) > 4 ? (int)list + 2*8 - 1 & -8 \
    : (int)list + 2*4 - 1 & -4))) [-1]
*/
```

The **va_dcl** macro declares the formal parameters `va_alist`, which is either the format descriptor for the remaining parameters or a parameter itself.

The **va_start** must be called within the body of the function whose argument list is to be traversed. The function can then transverse the list or pass its `va_list` pointer to other functions to transverse the list. The **type** of the **va_start** argument is `va_list`; it is defined by the **typedef** statement in `varargs.h`.

The **va_arg** macro accesses the value of an argument rather than obtaining its address. The macro handles those type names that can be transformed into the appropriate pointer type by appending an asterisk (*), which handles most simple cases. The argument type in a variable argument list must never be an integer type smaller than **int**, and must never be **float**.

For more information on the `varargs.h` macros, see the `varargs` man page in the *UNIX Programmer's Manual*. Figure A.1 shows an example of the use of varargs macros; the expected output from the example is as follows:

```

load i 0 r
load i 4 4
add I
store I 0 4

```

```

#include <varargs.h>                #include <stdio.h>
#include <stdio.h>
enum operations {load, store, add, sub};
main () {
    void emit();
    emit(load, 'I', 0, 4);
    emit(load, 'I', 4, 4);
    emit(add, 'I');
    emit(store, 'I', 0, 4);
}
void
emit( op, va_alist )
/* emit takes a variable number of arguments and prints
/* them according to the operation format */
enum operations op;
va_dcl {
va_list arg_ptr;
register int length, offset;
register char type;
va_start(arg_ptr);
switch (op) {
    case add: /* print operation and length*/
        type = va_arg(arg_ptr, int);
        printf("add %c n", type);
        break;
    case sub: /* print operation and length*/
        type = va_arg(arg_ptr, int);
        printf("sub n", type);
        break;
    case load: /* print operation, offset and length */
        type = va_arg(arg_ptr, int);
        offset = va_arg(arg_ptr, int);
        length = va_arg(arg_ptr, int);
        printf("load %c %d %d n", type, offset, length);
        break;
    case store:
        type = va_arg(arg_ptr, int);
        offset = va_arg(arg_ptr, int);
        length = va_arg(arg_ptr, int);
        printf("store %c %d %dn", type, offset, length);
}
}

```

A.1. Passing a Variable Number of Arguments to a C Function.

Deviations

C does not support the **entry** keyword, which has no defined use. Additionally, C does not support the **asm** keyword, as implemented by some C compilers to allow for the inclusion of assembly language instructions.

Extensions

Extensions to C include the following:

- the **enumeration** type, a set of values represented by identifiers called enumeration constants; enumeration constants are specified when the type is defined. For information on the alignment, size, and value ranges of the **enumeration** type, see **Chapter 2**.
- the **void** type, which allows you to specify that no value be returned from a function.
- the **volatile** type modifier, which is used when programming I/O devices and the **signed** type. In addition, the **const** keyword has been reserved for future use. For more information on the **volatile** modifier, see **Chapter 2**.

Translation Limits

Table A.1 shows the maximum limits imposed on certain items by the C compiler.

<i>C Specification</i>	<i>Maximum</i>
Nesting levels	
Compound statements	≤ 30
Iterations	
Selections	
Conditional compilations	
Maximum number of type modifiers (arrays, pointers, function, volatile)	9
Case labels	500
Function call parameters.	150
Significant characters	≤ 32
External identifier	
Internal identifier	

Table A.1. C Compiler Limitations.

Appendix B

Pascal Implementation

The Pascal language supported by the compiler is an implementation of ANSI Standard Pascal (ANSI/IEEE770X3.97-1983). Pascal was originally designed as a language suitable for teaching programming concepts and one that would be efficient and reliable on available computers.

Pascal extends the standard definition to provide features for application development and to meet the following objectives:

- Provide extensions that make Pascal reasonable for a wide class of programs.
- Anticipate the requirements of customers.
- Be consistent with the direction of the emerging extended standard.
- Be consistent with the UNIX/C programming environment.

The sections that follow describe the extensions in detail.

Names

Pascal provides several extensions that apply to identifiers, the names that a Pascal programmer uses.

Use of Underscores

Many users find it helpful to use the underscore character in identifiers. The underscore can be used to make names that are composed of several words. The underscore is also used in the C programming language; including the feature in Pascal makes it possible to call to those C functions that require an underscore.

The compiler treats the underscore as an alphabetic character; it can be placed in any position of the identifier including the first, as shown below.

```
read_integer
_bits_per_word
```

Lowercase in Public Names

Pascal does not distinguish between uppercase and lowercase in respect to variable names. This causes a problem for those names meant to be linked with C functions, where case is significant.

Pascal converts all names that are used to refer to external variables, procedures, or functions into lowercase.

Alphabetic Labels

Pascal permits alphabetic labels in addition to numeric labels as specified by the Pascal standard.

Constants

Non-Decimal Number Constants

It is often useful to write integer constants in a radix other than base 10. This occurs in programs that use data structures defined by the system. Other than base 10, base 2, 8, and 16 are the most frequently used bases.

Pascal allows the use of any base from 2 through 36. You can specify a number in those bases by using the form:

base#number

The base is a decimal number in the range 2 to 36. The number after the base is a number in the specified radix using *a..z* (either case) to denote the values 10 through 25.

The number must specify a value in the range 0..maxint (2147483647). The following example shows the number 42 in several different bases.

```
2#101010
3#1120
4#222
8#52
42
10#42
11#39
16#2a
```

If the radix is a power of two (2, 4, 8, 16, or 32), the number may be negative if it specifies a 32-bit value. For example, 16#8000000 is -2147483648, which is the smallest integer contained in a 32-bit number.

String Padding

Standard Pascal requires that a literal character string have the same length as the variable with which it is used. Manually adding extra blanks invites errors, and changing a Pascal type definition would require manually changing all literal strings that are used with variables of that type.

Pascal pads a string constant with blanks on the right according to its use. That is, assigning a 3-character literal to a 6-character variable causes the string literal to be treated as being 6 characters long. Assigning a 6-character literal string to a variable containing 3 characters causes an error.

A string of length 1 is regarded as a character rather than a string. As a result, a string padding is not applied to a string of length 1. Add a blank space to the single character if you want padding to take place.

Non-Graphic Characters

Literal character strings can't contain ASCII characters that have no graphic representation. The most commonly used characters that have no graphic representation are control characters.

Pascal has a special form of a character string, enclosed in double quotation marks, in which such characters may be included. A backslash escape character is used to signal the use of a special character. For example, the following line rings (beeps) the bell on an ASCII terminal.

```
writeln(output, "\a");
```

The following table lists the escape character sequence used to encode special characters.

Character	Result
\a	alert (16#07)
\b	backspace (16#08)
\f	form feed (16#0c)
\n	newline (16#0a)
\r	carriage return (16#0d)
\t	horizontal tab (16#09)
\v	vertical feed (16#0b)
\\	backslash (16#5c)
\"	quotation mark (16#22)
\'	single quote (16#27)
\nnn	character with octal value of <i>nnn</i> .
\x nnn	character with hexadecimal value of <i>nnn</i> .

Constant Expressions

Using a constant expression in type definitions or constant definitions often makes programs easier to read and maintain. The following example shows the use of a constant expression—changing a single definition (*array_size*) changes both the size of the array and the definition of the index type used to access it.

```
const
  array_size = 100;
type
  array_index = 0..array_size-1;
var
  v : array[ array_index ] of integer;
```

Pascal permits a constant expression to be used where a single integer or scalar constant is ordinarily used. An expression can consist of any of the following operators and predefined functions:

Operator	Function
+	addition
-	subtraction and unary minus
*	multiplication
div	integer division
mod	modulo
=	equality relation
<>	inequality relation
<	less than
<=	less than or equal
>=	greater than or equal
>	greater than
()	parenthesis
bitand	bit-wise and
bitor	bit-wise or
bitxor	bit-wise exclusive or
lshift	logical left shift
rshift	logical right shift
lbound	low bound of an array
hbound	high bound of an array
first	lowest value of a scalar type
last	highest value of a scalar type
sizeof	the size (in bytes) of a data type
abs	absolute value
chr	inverse ordinal value for a character
ord	the ordinal value of a scalar value
pred	the predecessor to a scalar value
succ	the successor to a scalar value
type- functions	converts from one type to another

Pascal does not permit the use of a leading left parenthesis in constant expressions for the lower value of subrange types. That is,

```
subrange = (11+12)*13 .. 14+15;
```

mistakenly assumes the type is an enumeration instead of a subrange.

Statement Extensions

Ranges in Case Statement Constants

Pascal permits the specifications of value ranges in a **case** statement. The selectors in a case statement may be specified using the Pascal range notation to mean all values inclusive, as shown in the following example.

```
case i of
  1900..1999 : writeln('twentieth century');
  1800..1999 : writeln('nineteenth century');
  1700..1799 : writeln('eighteenth century');
  1600..1699 : writeln('seventeenth century');
otherwise :
  write(i div 100:1);
  writeln('th century');
end;
```

Otherwise Clause in Case Statement

Pascal extends the **case** statement to allow an **otherwise** clause, which is the default statement when no case clause equal to the case value exists. Any number of statements can be included between **otherwise** and **end**, as shown in the preceding example.

If no **otherwise** is specified, and no case clause satisfies the case selector values during execution, a case error warning message is printed.

Return Statement

Pascal provides a **return** statement that permits a procedure or function to return to the caller without branching to the end of the procedure or function. If used within a function, **return** may be coded with an optional expression. The last value assigned to the function name is returned unless this expression is specified. Using the **return** statement is equivalent to assigning the value in the expression to the function name and executing a **goto** to the end of the routine.

```
function factorial(n:integer):integer;
begin
  if n = 1 then return(1)
  else return(n*factorial(n-1))
end;
```

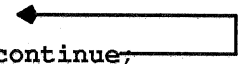
Continue Statement

The **continue** statement causes the flow of control to continue at the next iteration of the nearest enclosing loop statement (**for**, **while**, or **repeat**). If the statement is a **for** statement, the control variable is incremented and the termination test is performed.

```

for J := 1 to i do begin
    if Line[J] = ' ' then continue;
    write(output, Line:J);
end {for};

```



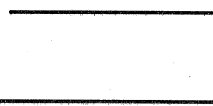
Break Statement

The **break** statement causes the flow of control to exit the nearest enclosing loop statement (**for**, **while**, or **repeat**). This feature permits you to end a loop without using the test specified on the loop statement.

```

while p <> nil do begin
    if p^.flag_ = 0 then break;
    p := p^.next;
end;
a := i;

```



Declaration Extensions

Separate Compilation

Pascal permits breaking a program into several compilation units, one that contains the main program, and the others containing procedures and functions called by the main program; the procedures and functions it calls need not be written in Pascal. Pascal also permits separately compiled compilation units to share data.

The compilation unit that contains the main program (the *program compilation unit*) follows standard Pascal syntax for a program. A compilation unit that contains separately compiled procedures and functions is called a *separate compilation unit*. Only one program compilation unit is permitted.

In a separate compilation unit, procedures, functions, and variables are placed sequentially without any program header or main program block.

Pascal allows Pascal declarations to be placed before the program header, whereas standard Pascal does not. All procedures, functions, and variables preceding the program header are given an external scope. This means that they

may be called (procedures and functions) or used (variables) by separate compilation units. The following example illustrates a program compilation unit:

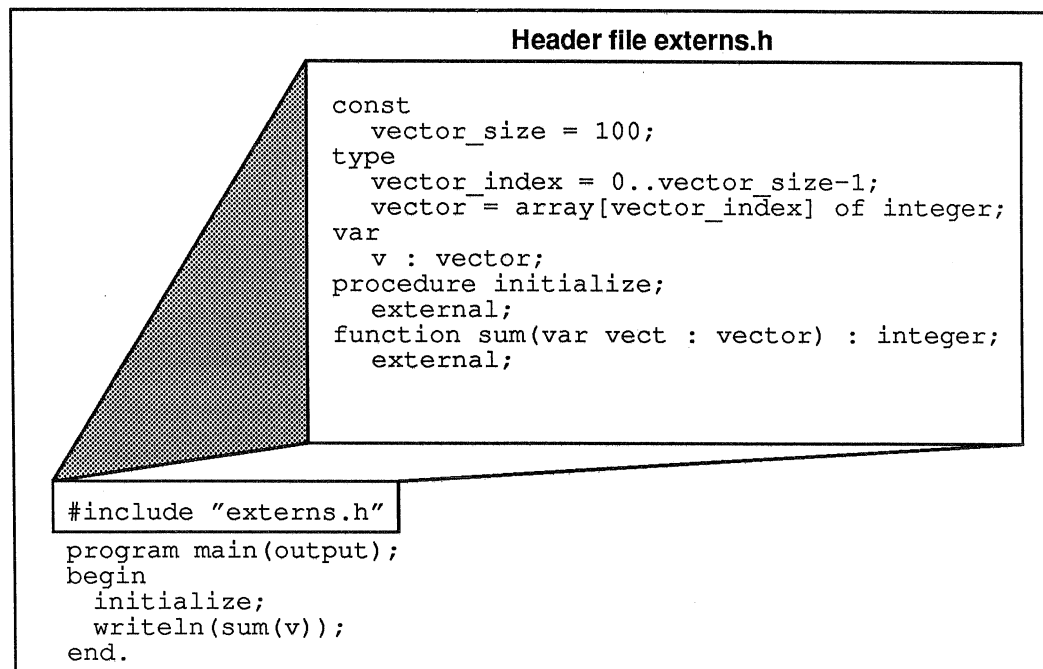


Figure B.1. External Declarations in a Program Compilation Unit.

In Figure B.1, **external** directives and other statements in the header file *externs.h* specify that *initialize* and *sum* (used by the main program) and their parameters are defined in a separate compilation unit. The external directive can be used to qualify only unnested routine names.

Initialize and *sum* must be defined when the main program is link edited. Consider the following example, where *mainone.p* contains the Pascal source code for the main program and *bodies.o* contain a previously compiled object module of *initialize* and *sum*:

```

pc -c mainone.p      ———  Compiles main program.
pc -o exec mainone.o bodies.o ———  Link edits the main program
                                with sum and initialize.

```

The external directive is similar to the Pascal **forward** directive, because it declares a routine name and its parameters without defining the body of the routine. The external directive can be used only to qualify unnested routine names.

All procedures, functions, and variables at the outermost level are given an external scope. Figure B.2 below shows the separate compilation unit that defines the routines shown in Figure B.1.

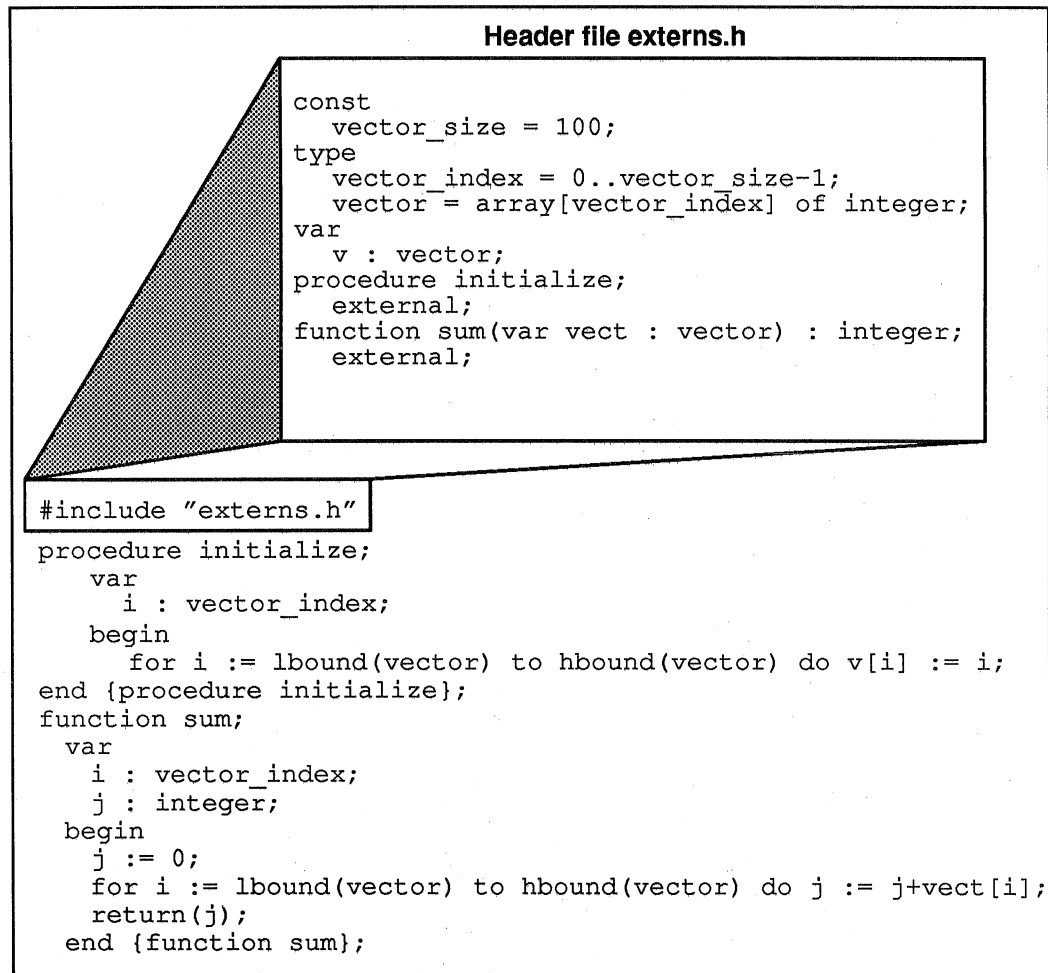


Figure B.2. External Declarations in a Separate Compilation Unit.

In Figure B.2, *initialize* and *sum* are defined with the **external** directive.

Note that, in this example, the external declarations were placed in the include file *externs.h*. This reduces the chance of errors due to inconsistent declarations; such includes are recommended for statements shared by multiple compilation units.

Shared Variables

All variables that are declared at the outermost nesting of a compilation unit have an external scope and can be used in different compilation units. The variable can have only one initializing clause and can be placed in one compilation unit. The previous example illustrates a variable, named *v*, that is accessed from both compilation units.

Initialization Clauses

Pascal permits an external variable to have a clause that initializes either a scalar, a set, an array, or a pointer. The BNF syntax of a variable declaration is extended to:

```

var-decl ::= identifier-list ":" type-denoter [ ":"=" in-
initial-clause ]

initial-clause ::= constant-expr |
                "[" initial-value-list "]"

initial-value ::= constant-expr [ ".." constant-expr ] |
                constant-expr ":" constant-expr |
                "otherwise" ":" constant-expr |
                "[" initial-value-list "]"

initial-value-list ::= initial-value |
                    initial-value-list "," initial-value

```

For scalar types, this initialization is very simple:

```

var
  a: integer := 5;
  letter: char := 'x';
  x1: real := 6.5;

```

An initial value may also be given to a structured type (array or set); every element of an array must be initialized or given a specified default value.

```

type
  color = (red, yellow, blue);
  hue = set of color;
  vector = array[1..100] of integer;
var
  orange : hue := [red, yellow];
  black : hue :=
    [first(color)..last(color)];
  name : packed array[1..32] of char :=
    'MIPS Computer Systems';
  vect : vector :=
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
     30 : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
     otherwise : 99];
  pointers : array[0..127] of ^vector :=
    [otherwise: nil];

```

The above clauses initialize two sets, a string and two arrays. All elements of the array *vect*, except the first ten elements and the ten elements starting at index 30, are initialized to 99. The array *pointers* is initialized entirely to the value *nil*.

Relax Declaration Ordering

The declaration clauses can be written in any order and may be repeated. The Pascal requirement that a declaration must precede any use must still be observed.

Predefined Procedures

Assert

```
assert(boolean-expr [, string])
```

The **assert** procedure evaluates a boolean expression and signals an execution time error (similar to a checking error) if the value is not true. If you specify the optional string, the string is written to standard error, otherwise the message

```
assertion error in Pascal program
```

is generated along with the line number and file name of the **assert** statement that causes the error.

Argv

```
argv(integer-expr, string-var)
```

The **argv** procedure returns the *i*-th program argument, placing the result in a string. The string can be blank, padded, or truncated, as required. The value of the first parameter must be in the range 0..*argc*-1. Argument 0 is the name of the program.

Date

```
date(string-var)
```

The **date** procedure returns the current date. The resulting string has the form *yy/mm/dd*, where *yy* is the last two digits of the year, *mm* is the number of the month (01 is January), and *dd* is the day of the month (01 is the first day). If the string is less than 8 characters long, data is truncated on the right; if the string is longer than 8 characters, the string is padded with blanks on the right.

Time

```
time(string-var)
```

The **time** procedure returns the current time. The resulting string has the form *hh:mm:ss*, where *hh* is the hour (24-hour clock), *mm* is the minutes (00 means on the hour), and *ss* is the seconds (00 means on the minute). If the string is less than 8 characters long, data is truncated on the right; if the string is longer than 8 characters, the string is padded with blanks on the right.

Predefined Functions

Type Functions

Standard Pascal offers two predefined functions, **ord** and **chr**, that convert predefined scalar types.

Pascal allows all scalar types to have a conversion function that converts an integer into that scalar type. As in standard Pascal, the **ord** function converts from the scalar type to integer. Pascal lets all type identifiers for a scalar type be used in an expression to convert its integer argument into the corresponding scalar value. Thus, if *color* is an enumerated data type, *color*(*i*) is a function that returns the *i*+1-th element of the enumeration (i.e., *color*(0) is first, *color*(1) is second, etc.).

```
boolean-var := boolean(integer-expr)
char-var := char(integer-expr)
color-var := color(integer-expr)
```

In Pascal, the **ord** function also operates on pointers, returning the machine address of the item referenced by the pointer. A data type identifier that represents a pointer data type can also be used to convert a cardinal number into a valid pointer of that type. This feature is highly machine dependent and should be used sparingly.

Min

```
scalar-var := min(scalar-expr [, scalar-expr] ... )
```

The **min** function returns the smallest of its scalar arguments. For example:

```
min(-6, 3, 5)
```

returns -6.

Max

```
scalar-var := max(scalar-expr [, scalar-expr] ... )
```

The **max** function returns the largest of its scalar arguments. For example:

```
max(-2, 3, 5)
```

returns 5.

Lbound

```
scalar-var := lbound(array-type [, index])
```

The **lbound** function returns the lower bound of the *array type* specified by the first argument. The array type must be specified by a type identifier or a variable whose type is an array. If the array is multi-dimensional, then an optional second argument specifies the dimension; 1, specifying the first or outermost dimension, is the default.

Hbound

```
scalar-var := hbound(array-type [, index])
```

The **hbound** function returns the high bound of the *array type* specified by the first argument. The *array type* must be specified by a type identifier or a variable whose type is an array. If the array is multi-dimensional, then an optional second argument specifies the dimension; 1 is the default which means the first (outermost) dimension.

First

```
scalar-var := first (type-identifier)
```

The **first** function returns the first (lowest) value of the named scalar type. For example, `first(integer)` returns `-2147483848`.

Last

```
scalar-var := last (type-identifier)
```

The **last** function returns the last (highest) value of the named scalar type. For example, `last(integer)` returns `2147483847` (`maxint`).

Sizeof

```
sizeof (type-id [, tagfield-value] ... )
```

The **sizeof** function returns the number of bytes occupied by the data type specified as an argument. If the argument specifies a record with a variant record part, then additional arguments specify the value of each tagfield.

Argc

```
integer-var := argc
```

The **argc** function returns the number of arguments passed to the program. The value of the function is 1 or greater.

Clock

```
integer-var := clock
```

The **clock** function returns the milliseconds of processor time used by the current process.

Bitand

```
integer-var := bitand (integer-expr, integer-expr)
```

The **bitand** function returns the bit-wise *and* of the two integer valued operands.

Bitor

```
integer-var := bitor (integer-expr, integer-expr)
```

The **bitor** function returns the bit-wise *or* of the two integer valued operands.

Bitxor

```
integer-var := bitxor (integer-expr, integer-expr)
```

The **bitxor** function returns the bit-wise *exclusive or* of the two integer valued operands.

Bitnot

```
integer-var := bitnot (integer-expr)
```

The **bitnot** function returns the bit-wise *not* of the integer valued operand.

Lshift

```
integer-var := lshift(integer-expr, integer-expr)
```

The **lshift** function returns the left shift of the first integer valued operand by the amount specified in the second argument. Zero bits are inserted on the right.

Rshift

```
integer-var := rshift(integer-expr, integer-expr)
```

The **rshift** function returns the right shift of the first integer valued operand by the amount specified in the second argument. Sign bits are inserted on the left if the operand is an integer type; zero bits are inserted if the operand is a cardinal type. You can force zero bits with the following construct:

```
rshift(cardinal(intexpr), shiftamount)
```

I/O Extensions

Specifying Radix in the Write Statement

Pascal permits the specification of operands in a **write** statement that write integer or cardinal numbers in any radix from 2 through 36. The following code writes a number in each radix:

```
for i:= 2 to 36 do
  writeln('x is ', x:1:i, ' in radix ', i:1);
```

A minus sign precedes any printed number if the type of the number is integer and the value is less than zero. If the type of the number is cardinal, then the compiler interprets the number as not having a sign and prints the sign bit as part of the number rather than with a negative sign.

Filename on Rewrite and Reset

Pascal accepts an optional string argument that specifies the path name of the file to be opened/created. Otherwise, it creates a file in a temporary area.

```
reset(input [, string])
rewrite(output [, string])
```

Reading Character Strings

Pascal allows you to read characters into a string array, while standard Pascal does not. A string is defined to be a packed array of char whose lower bound is 1 and whose upper bound is greater than 1. The array is padded with blanks when it is longer than the line, and the line is truncated if it is longer than the

array. In the following example, characters are read into the array one line at a time until the end-of-line (eoln) is reached.

```

program CountLines(input, output);
type
  string80 = packed array[1..80] of char;
var
  Line : string80;
begin
  .
  .
  while not eof(input) do begin
    readln(input, Line)
    i := i+1;
  end; {while}
  write ('The number of lines is ', I:1);

```

Reading and Writing Enumeration Types

Pascal provides an extension to permit any enumerated scalar type to be specified as the operand of a READ or a WRITE to a text file.

Reading an enumeration value interprets the programmer-defined name, preceded optionally by blank, tab, or new-line characters. The end of the name is delimited by any character that is not a valid character of an identifier. The delimiting character is skipped. Good choices for delimiting characters are blanks, tabs, commas, or new-lines.

Writing an enumerated value causes the programmer defined name to be written out to the text file. (Input and output is case sensitive.)

The following is an example of using enumerated values.

```

program testenum(input, output);
type
  color = (red, orange, yellow, green, blue, violet,
          black, white);
var
  vcolor : color;
begin
  repeat
    writeln('enter color');
    read(vcolor);
    writeln('The color is ', vcolor : 0);
  until eof;
end.

```

If any color other than one specified in the *color* enumeration is entered, the following message appears:

```
enumerated value string not within type
The color is red
```

where *string* represents the incorrect value entered. When an incorrect value is entered, the first value in the enumeration (*red* in the above example) is written to the screen.

Lazy I/O

Pascal provides an interpretation of the standard that simplifies terminal-oriented I/O. Standard Pascal defines the file pointer of a text file to point to the first character of a line after a **reset** or **readln** operation. This makes it difficult to issue a prompt message because the physical I/O operation for the next line occurs at the end of the **readln** procedure.

Pascal follows standard Pascal conventions, except that it doesn't perform physical I/O until the user actually uses the file pointer. In effect, it's *lazy* about performing the I/O operation. This allows the user to issue a prompt message after the **readln** (or **reset**) prior to the time when the user's terminal attempts to read the next line.

Standard Error

Pascal provides an additional predefined text file called **err** which is mapped to UNIX standard error. Also, the files **input** and **output** are mapped to the UNIX file standard input and standard output.

Predefined Data Type Extensions

Double

Pascal accepts a new predefined data type called **double** that represents a double precision floating point number. If either operand of an expression is double, then the compiler uses double precision.

Cardinal

Pascal accepts the **cardinal** data type that represents an unsigned integer in the range 0..4294967295 ($2^{32}-1$). If either operand of an expression is cardinal, then the compiler uses unsigned arithmetic.

Pointer

Pascal defines a new predefined data type called **pointer** that is compatible with any pointer type. This type can be thought of as the type of the Pascal value **nil**. Use **pointer** to write procedures and functions that accept an arbitrary pointer type as an operand. You cannot directly dereference a variable of this type because it is not bound to a base type. To dereference it, you must first assign the variable to a normally typed pointer.

Compiler Notes

Macro Preprocessor

Pascal invokes the C preprocessor (**cpp**) before each compilation allowing, you to use **cpp** syntax in the program. The **cpp** variables **LANGUAGE_PASCAL**,

LANGUAGE_C, LANGUAGE_FORTRAN, and LANGUAGE_ASSEMBLY are defined automatically, allowing you to build header files that can be used by different languages. The following example shows two conditional statements, one written in Pascal and the other in C.

```

#ifdef LANGUAGE_PASCAL
type
  pair =
    record
      high, low : integer;
    end {record};
#end
#ifdef LANGUAGE_C
typedef struct {
  int high, low;
} pair;
#end

```

You can also use the full conditional expression syntax of cpp, as well as C style comments (`/* ... */`), which are stripped during compilation.

Short Circuiting

Pascal always short circuits boolean expressions. Short circuiting is a technique where only a portion of a boolean expression is actually executed. For example, in

```
if (P <> nil) and (P^.Count > 0) then
```

the expression involving `P^.Count` isn't evaluated if the first expression is false. This extension is permitted by the Pascal standard; a program that relies on this feature, as does this example, would not be portable.

Translation Limits

The following table shows the maximum limits imposed on certain items by the Pascal compiler.

Pascal Specification	Maximum
Literal string length	288
Procedure nesting levels	20
Set size	481 – 512*
Significant characters	32
* Depends on actual limits of data item.	

What Is Byte Ordering?

A machine's byte ordering scheme (or whether a machine is big-endian or little-endian) affects memory organization and defines the relationship between address and byte position of data in memory. MIPS machines can be big-endian or little-endian.

Big-Endian Byte Ordering

Big-endian machines number the bytes of a word from 0 to 3. Byte 0 holds the sign and most significant bits. For halfwords, big-endian machines number the bytes from 0 to 1. Again, byte 0 holds the sign and most significant bits. Machines that use big-endian schemes include the IBM s/370 and Motorola MC68000.

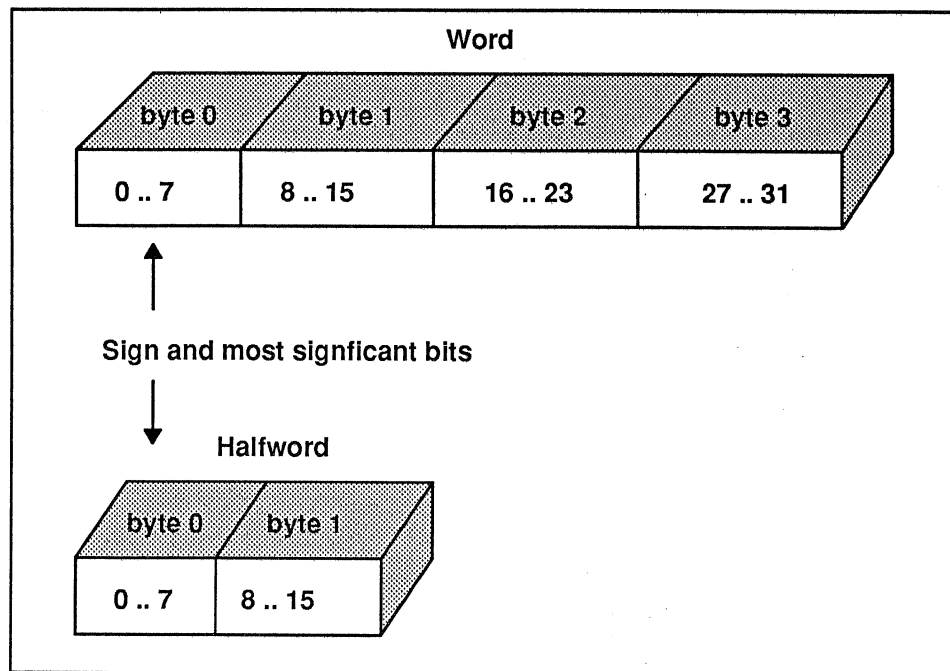


Figure C.1. Big-endian byte ordering.

Little-Endian Byte Ordering

Little-endian machines number the bytes of a word from 3 to 0. Byte 3 holds the sign and most significant bits. For halfwords, little-endian machines number the bytes from 1 to 0. Byte 1 holds the sign and most significant bits. Machines that use little-endian schemes include: DEC VAX & 11/780, Intel 80286, and National Semiconductor 32000.

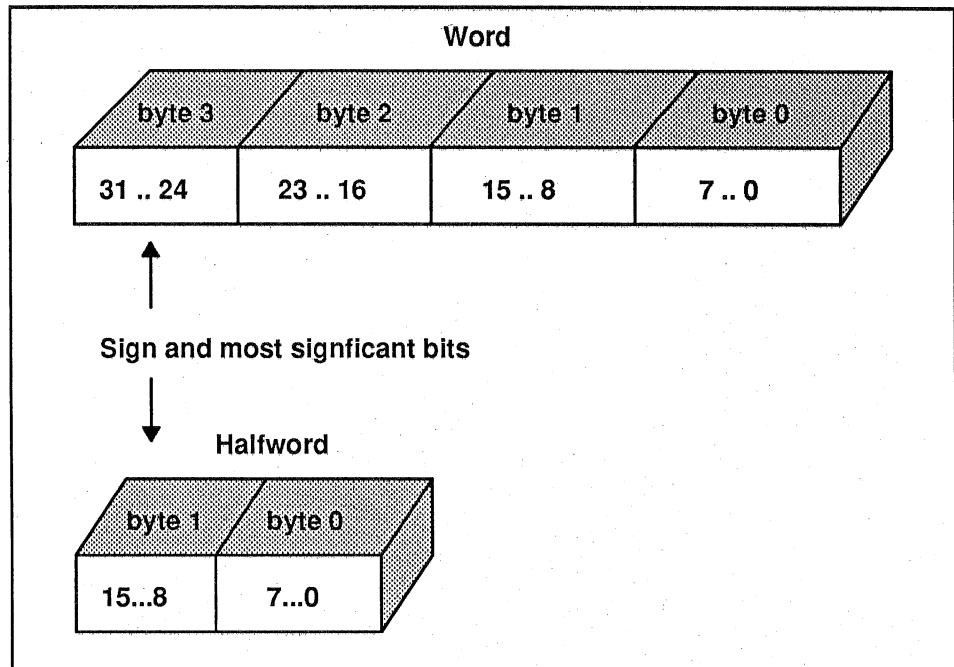


Figure C.1. Little-endian byte ordering.

Symbols

? command, in DBX, 5–34

/ command, in DBX, 5–34

Numbers

–5, compiler option, 1–8

A

–A link editor option, 1–16

a.out default, for –o option, 1–9

activation levels, 5–3

alert character, in Pascal, B–3

alias command, in DBX, 5–22

ar command, syntax, 1–33

archiver

ar command, 1–22

examples, 1–33

options, 1–34—1–37

purpose, 1–32

argc, Pascal predefined function, B–12

argv, Pascal predefined procedure, B–10

as driver command, 1–2

assembler, driver command, 1–2

assert, Pascal predefined procedure, B–10

assign command, in DBX, 5–40

auto (C storage class), 2–6

B

–B link editor option, 1–16

–B *num* link editor option, 1–16

BREAK statement, in Pascal, B–6

BSD memory map, producing, 1–18

–Bstring link editor option, 1–16

–b link editor option, 1–16

backslash character, in Pascal, B–3

backspace character, in Pascal, B–3

basic block counting, overview, 4–2

–bestGnum option

and –G option, 1–9

description, 1–17

examples, 4–32

big endian. *See* byte ordering

bitand, Pascal predefined function, B–12

bitnot, Pascal predefined function, B–12

bitor, Pascal predefined function, B–12

bitxor, Pascal predefined function, B–12

boolean expressions, short circuiting,, in Pascal, B–16

byte ordering, C–1

–EB (big endian) option, 1–10

–EL (little endian) option, 1–10

for C language structures, 2–2

Pascal arrays and records, 2–11—2–17

C

C language

arrays, 2–2

data types, 2–1—2–6

driver command, 1–2

extensions, A–3

interface to Pascal programs, 3–1—3–10

operators used by DBX, 5–10

storage classes, 2–6—2–7

structures, 2–2

translation limits, A–3

unions, 2–6

vararg.h macros, A–1

C macro preprocessor, option, 1–8

–C option

for C and assembler, 1–8

for Pascal and FORTRAN, 1–8

CASE statement ranges, in Pascal, B–5

CONTINUE statement, in Pascal, B–6

Cobol, driver command, 1–2

–c option

See also –o option

description, 1–8

cache conflicts, eliminating, 4–33

calls

C programs from Pascal, 3–7—3–10

Pascal programs from C, 3–4—3–7

cardinal, Pascal predefined data type, B-15
 carriage return character, in Pascal, B-3
 catch command, in DBX, 5-45
 cc driver command, 1-2
 characters, maximum significant, in Pascal, B-16
 chr, Pascal predefined function, B-10
 clock, Pascal predefined function, B-12
 cobol driver command, 1-2
 common files, 1-12
 compiler options

- byte-ordering, 1-10
- compiler development, 1-12
- debugging, 1-11, 5-5
- general, 1-7
- optimizer, 1-12

 compiling multi-language programs, 1-6
 constant expressions, in Pascal, B-3
 cont command, in DBX, 5-39
 conti command, in DBX, 5-53
 -cord

- compiler option
 - description, 1-8
 - example, 4-33-4-34
 - prof option, example, 4-33-4-34
 - reducing cache conflicts with, 4-33

 -count link editor option, 1-17
 -countall link editor option, 1-17
 cpp (C preprocessor), invoked by Pascal, B-15
 -cpp option, 1-8

D

-D *num* link editor option, 1-16
 -D option, 1-8
 data types

- C language
 - alignment, 2-1-2-6
 - size, 2-1-2-6
 - value ranges, 2-1-2-6
- Pascal
 - alignment, 2-8-2-17
 - size, 2-8
 - value ranges, 2-8-2-17

date, Pascal predefined procedure, B-10

DBX

command file, 5-5
 command summary, 5-57
 command syntax, 5-7
 commands, 5-12
 constants, 5-10
 data types, 5-10
 examining program state, 5-46
 example, 5-7
 expressions, 5-9
 guidelines to using, 5-3
 how to use, 5-5-5-8
 invocation syntax, 5-6
 machine level debugging, 5-51
 purpose, 5-2
 quitting, 5-7
 sample program, 5-64
 setting breakpoints, 5-41
 variables, predefined, 5-18-5-19
 debugging

- See also* DBX
- before optimization, 5-5
- g options, 1-11
- guidelines, 5-3
- how to use DBX, 5-5-5-8
- output, 5-4

 declarations, in Pascal, B-6
 delete command, in DBX, 5-30
 double, Pascal predefined data type, B-15
 down command, in DBX, 5-47
 driver, 1-1

- commands, 1-2

 dump command, in DBX, 5-50

E

-E option, 1-8
 -EB link editor option, 1-16
 -EL link editor option, 1-16
 -e epsym link editor option, 1-16
 edit command, in DBX, 5-34
 endian byte ordering. *See* byte ordering
 escape characters, B-3
 examples

- cord option, using to reduce cache conflicts, 4-33
- k option, 4-22-4-24

ar (archiver) command, 1-33
 -bestGnum option, 4-32-4-33
 C call
 to Pascal function, 3-6
 to Pascal procedure, 3-6
 C main routine, 3-4
 C routine passing strings to Pascal, 3-7
 compiling multi-language programs, 1-6
 excluding libraries, 4-32-4-33
 file command listing, 1-31
 -j option, 4-22-4-24
 link editor -l option, 1-14
 linking objects, 1-6
 nm symbol table listing, 1-27
 -nocount option, 4-32-4-33
 -O3 optimization, 4-20-4-21
 odump listings, 1-23-1-27
 optimizing with ulink, umerge, 4-20-4-21
 Pascal call
 passing arrays to C function, 3-10
 to C function, 3-9
 to C procedure, 3-9
 to C routine, 3-3
 Pascal main routine, 3-4
 pixie, using to reduce cache conflicts, 4-33
 prof, using to reduce cache conflicts, 4-33
 size command listing, 1-31
 ucode object library
 building, 4-24
 using, 4-24
 extern (C storage class), 2-6
 external directives, in Pascal, B-7

F

-F link editor option, 1-17
 FORTRAN, operators used by DBX, 5-10
 FORTRAN, driver command, 1-2
 -f fill link editor option, 1-16
 f77 driver command, 1-2
 -feedback
 compiler option
 description, 1-8
 example, 4-33-4-36
 prof option
 description, 4-16
 example, 4-33-4-36

file command, 1-31
 in DBX, 5-32
 file suffixes, 1-3
 first, Pascal predefined function, B-12
 form feed character, in Pascal, B-3
 formal directives, in Pascal, B-7
 func command, in DBX, 5-31

G

-G num link editor option, 1-17
 -G option, 1-9
 See also global data area
 -g debugging options, 1-11
 global data area
 controlling size of, 4-31
 determining optimal size, 4-32
 layout, 4-31
 purpose, 4-31
 goto command, in DBX, 5-39

H

hbound, Pascal predefined function, B-11
 header files, 1-12
 hexadecimal value character, in Pascal, B-3
 horizontal tab character, in Pascal, B-3

I

-I dirname option, 1-9
 -I option, 1-9
 -i option, 1-9
 ignore command, in DBX, 5-45
 including files, 1-12, 1-13
 invocation counting, overview, 4-2

J

-j option
 description, 1-9
 examples using, 4-22-4-24

- with `-i` option, 1-9
- `-jmopt` link editor option, description, 1-17
- `-jmpopt` link editor option, purpose, 4-34
- jump delay slot, and link editor, 4-34

K

- `-k` option, 1-9
 - examples using, 4-22—4-24
- `-ko` option, 1-9

L

- `-L` *dirname* link editor option, 1-18
- `-L` link editor option
 - and `L` *dirname* option, 1-18
 - description, 1-18
- language interfaces, 3-1
- languages supported, 1-2
- last, Pascal predefined function, B-12
- lazy I/O, in Pascal, B-15
- `lbound`, Pascal predefined function, B-11
- `ld` command, 1-14
- `/lib` link library, specifying, 1-18
- libraries, specifying, 1-14
- link editor
 - how to run, 1-14
 - purpose, 1-14
 - version number, determining with `-V` option, 1-19
- link library, specifying, 1-18
- list command, in DBX, 5-33
- literal string, maximum length, in Pascal, B-16
- little endian. *See* byte ordering
- `lshift`, Pascal predefined function, B-13
- `-lx` link editor option, 1-18

M

- `-M` link editor option, 1-18
- `-m` link editor option, 1-18
- max, Pascal predefined function, B-11

- memory map, producing
 - BSD format, 1-18
 - System V format, 1-18
- min, Pascal predefined function, B-11

N

- `-N` link editor option, 1-18
- NMAGIC file, creating, 1-18
- `-n` link editor option, 1-18
- newline character, in Pascal, B-3
- next command, in DBX, 5-37
- nexti command, in DBX, 5-53
- nm list utility, 1-27
- `-nocount` link editor option, 1-17
- `-nocpp` option, 1-9
 - See also* `-cpp` option
- `-nojmopt` link editor option
 - description, 1-17
 - purpose, 4-34

O

- `-O` optimizing options, 4-19—4-20
- `-O3` optimizer option, example, 4-20
- OMAGIC file, creating, 1-18
- OTHERWISE clause, in Pascal, B-5
- `-o` filename link editor option, 1-19
- `-o` filename option, 1-9
- object file (a.o)
 - producing with `-c` option, 1-8
 - tools, 1-20—1-38
- octal value character, in Pascal, B-3
- odump (object file utility)
 - example output, 1-23
 - options, 1-21
 - purpose, 1-20
 - syntax, 1-20
- `-Olimit` option
 - purpose, 4-21
 - using to improve performance, 4-21
- optimization
 - `-O` options, 1-12
 - benefits, 4-17

debugging before, 4-17
 effect of `-C` bounds checking options, 4-17
 full, examples, 4-20—4-21
 global, 4-16
 improving
 C and FORTRAN programs, 4-29
 C and Pascal programs, 4-25—4-28
 C programs, 4-29
 C, Pascal, and FORTRAN programs, 4-24, 4-30
 Pascal programs, 4-30
 loops, 4-17
`-O` options, 4-19—4-21
 of frequently used modules, 4-22—4-24
 of separate compilation units, 4-18—4-19
 register allocation, 4-18

options
 compilation, 1-8—1-12
 link editor, 1-15—1-20

ord, Pascal predefined function, B-10

P

`-P` option, 1-9

Pascal
 arrays, 2-11
 arrays and records, byte ordering, 2-11—2-17
 data types, 2-8—2-17
 driver command, 1-2
 interface to C programs, 3-1—3-10
 operators used by DBX, 5-10
 records, 2-12
 variant records, 2-15

`-p`, `-p1` option, 1-10, 4-26
See also profiling

Pascal language, B-1
 extensions, B-5
 I/O extensions, B-13
 constants, B-2
 predefined functions, B-10
 predefined procedures, B-10

`pc` driver command, 1-2

`pc-sampling`
 how to, 4-12
 overview, 4-2

performance, improving
 coding hints, 4-24—4-30
 limiting size of global data area, 4-31—4-33
 reducing cache conflicts, 4-33

striping symbol table info with `-s` option, 1-19
See also `-x` option
 using the optimizer (`-O`) options, 4-16
 using the profiler (`prof`), 4-1—4-13

`-pfile` link editor option, description, 1-19

`-pixie`, `prof` option, description, 4-14

`pixie`
 overview, 4-2
 reducing cache conflicts with, 4-33
 using to profile, 4-8

`pixie` program, example, 4-33—4-34

`PL/I`, driver command, 1-2

`pl1` driver command, 1-2

playback command, in DBX
 for input, 5-27—5-28
 for output, 5-28

pointer, Pascal predefined data type, B-15

`print` command, in DBX, 5-48

`printf` command, in DBX, 5-48

`printregs` command, in DBX, 5-49

procedures nesting levels, maximum, in Pascal, B-16

`prof`. *See* profiling

profiling, 4-10
 basic block counts, how to, 4-8—4-10
 examples, output listings, 4-3—4-8
 file names, 4-13
 overview, 4-1
`pc-sampling`, how to, 4-12—4-13
`prof` command
 format, 4-13—4-14
 options, 4-14—4-16
 reducing cache conflicts with, 4-33

Q

qualifying variable names (`dbx`), 5-9

quotation mark character, in Pascal, B-3

R

`READ`, in Pascal, B-14

`RESET` I/O extensions, in Pascal, B-13

`RETURN` statement, in Pascal, B-5

`REWRITE` I/O extensions, in Pascal, B-13

-r link editor option, 1-19
 record command, in DBX
 for input, 5-25-5-26
 for output, 5-26-5-27
 register (C storage class), 2-6
 rerun command, in DBX, 5-36
 return command, in DBX, 5-38
 rshift, Pascal predefined function, B-13
 run command, in DBX, 5-36

S

-S, link editor option, 1-19
 -S option. *See* -c option
 System V memory map, producing, 1-18
 .s file, producing with -S option, 1-10, 4-26
 -s link editor option, 1-19
 See also -x option
 set, DBX command, 5-16
 set size, maximum, in Pascal, B-16
 set sizes, 2-17
 sh command, in DBX, 5-29
 shared variables, in Pascal, B-8
 single quote character, in Pascal, B-3
 size command, 1-31
 source command, in DBX, 5-27
 source level debugger, 5-2
 source-level debugger. *See* DBX
 special character, Pascal, B-3
 static (C storage class), 2-6
 status command, in DBX, 5-29
 -std option, 1-10, 4-26
 step command, in DBX, 5-37
 stepi command, in DBX, 5-53
 stop at command, in DBX, 5-41
 stop if command, in DBX, 5-43
 stop in command, in DBX, 5-42
 stopi command, in DBX, 5-52
 storage classes (C language), 2-6-2-7

string array characters, reading, in Pascal, B-13
 symbol table
 dumping, 1-27
 stripping from object modules, with -x option, 1-19
 See also -s option

T

-T num link editor option, 1-19
 text segment, setting origin with -T num option, 1-19
 time, Pascal predefined procedure, B-10
 trace command, in DBX, 5-43
 tracei command, in DBX, 5-54
 type functions, Pascal, B-10

U

-U option, 1-10, 4-26
 -u symname link editor option, 1-19
 ucode object library, building, 4-24
 ulink, umerge
 examples, 4-20-4-21
 invoking for optimization, 4-19
 unalias command, in DBX, 5-22
 unset, DBX command, 5-17
 up command, in DBX, 5-47
 use command, in DBX, 5-31
 /usr/lib link library, specifying, 1-18
 /usr/local/lib link library, specifying, 1-18

V

-V
 compiler option, 1-10, 4-26
 link editor option, 1-19
 -VS num link editor option, 1-19
 -v
 compiler option, 1-10, 4-26
 link editor option, 1-19
 vararg.h macros, A-1
 variable number of arguments, A-1
 vertical feed character, in Pascal, B-3

volatile (C storage class), 2-6

W

WRITE I/O extensions, in Pascal, B-14

WRITE I/O extensions, in Pascal, B-13

-w option, 1-10, 4-26

whatis command, in DBX, 5-35

when command, in DBX, 5-44

where command, in DBX, 5-46

whereis command, in DBX, 5-35

which command, in DBX, 5-35

X

-x link editor option, 1-19

Z

ZMAGIC files, creating, 1-17

-z link editor option, 1-17

Customer Response Card

Your comments, which can assist us in improving our products and our publications, are welcome.

If you wish to reply, be sure to include your name and address, *and the name and part number that appears on the first page of this manual.*

Thank you for your cooperation.

No postage necessary if mailed in the U. S. A.

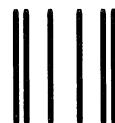
After writing comments, detach this page and then fold, seal, and mail.

Comments

Name of manual: _____

Part number: _____

MIPS may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED
STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1659 SUNNYVALE, CA

POSTAGE WILL BE PAID BY ADDRESSEE:



**MIPS Computer Systems
928 Arques Avenue
Sunnyvale, CA 94086-9756**

