The power of RISC is in the system.
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.
Summary of Changes

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 -jnopt, and -nojnopt 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
# Storage Mapping

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C Language</td>
<td>2-1</td>
</tr>
<tr>
<td>Alignment, Size, and Value Ranges</td>
<td>2-1</td>
</tr>
<tr>
<td>C Arrays, Structures, and Unions</td>
<td>2-2</td>
</tr>
<tr>
<td>Storage Classes</td>
<td>2-6</td>
</tr>
<tr>
<td>Pascal</td>
<td>2-8</td>
</tr>
<tr>
<td>Alignment, Size, and Value Ranges</td>
<td>2-8</td>
</tr>
<tr>
<td>Pascal Arrays, Records and Variant Records</td>
<td>2-11</td>
</tr>
<tr>
<td>Rules for Set Sizes</td>
<td>2-17</td>
</tr>
</tbody>
</table>

# Language Interfaces

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pascal/C Interface</td>
<td>3-1</td>
</tr>
<tr>
<td>General Considerations</td>
<td>3-1</td>
</tr>
<tr>
<td>Calling Pascal from C</td>
<td>3-4</td>
</tr>
<tr>
<td>Calling C from Pascal</td>
<td>3-7</td>
</tr>
<tr>
<td>Running C from Pascal</td>
<td>3-10</td>
</tr>
</tbody>
</table>

# Improving Program Performance

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>5-2</td>
</tr>
<tr>
<td>Why Use a Source-Level Debugger?</td>
<td>5-2</td>
</tr>
<tr>
<td>What Are Activation Levels?</td>
<td>5-3</td>
</tr>
<tr>
<td>Isolating Program Failures</td>
<td>5-3</td>
</tr>
<tr>
<td>Incorrect Output Results</td>
<td>5-4</td>
</tr>
<tr>
<td>Avoiding Some Pitfalls</td>
<td>5-4</td>
</tr>
<tr>
<td>Running DBX</td>
<td>5-5</td>
</tr>
<tr>
<td>Compiling Your Program for Debugging</td>
<td>5-5</td>
</tr>
<tr>
<td>Building a Command File</td>
<td>5-5</td>
</tr>
<tr>
<td>Invoking DBX (dbx)</td>
<td>5-6</td>
</tr>
<tr>
<td>Ending DBX (quit)</td>
<td>5-7</td>
</tr>
<tr>
<td>Using DBX Commands</td>
<td>5-7</td>
</tr>
<tr>
<td>DBX Command Syntax</td>
<td>5-7</td>
</tr>
<tr>
<td>Qualifying Variable Names</td>
<td>5-9</td>
</tr>
<tr>
<td>DBX Expressions and Precedence</td>
<td>5-9</td>
</tr>
<tr>
<td>DBX Data Types and Constants</td>
<td>5-10</td>
</tr>
<tr>
<td>Basic DBX Commands</td>
<td>5-12</td>
</tr>
<tr>
<td>Working with the DBX Monitor</td>
<td>5-12</td>
</tr>
<tr>
<td>Using History (history &amp; ! commands)</td>
<td>5-13</td>
</tr>
<tr>
<td>Editing on the DBX Command Line</td>
<td>5-13</td>
</tr>
<tr>
<td>Typing Multiple Commands</td>
<td>5-14</td>
</tr>
<tr>
<td>Completing Program Symbol Names</td>
<td>5-15</td>
</tr>
<tr>
<td>Controlling DBX</td>
<td>5-16</td>
</tr>
<tr>
<td>Setting DBX Variables (set)</td>
<td>5-16</td>
</tr>
<tr>
<td>Removing Variables (unset)</td>
<td>5-17</td>
</tr>
<tr>
<td>Predefined DBX Variables</td>
<td>5-18</td>
</tr>
<tr>
<td>Creating Command Aliases (alias)</td>
<td>5-22</td>
</tr>
<tr>
<td>Removing Command Aliases (unalias)</td>
<td>5-22</td>
</tr>
<tr>
<td>Predefined DBX Aliases</td>
<td>5-23</td>
</tr>
<tr>
<td>Recording Input (record input)</td>
<td>5-25</td>
</tr>
<tr>
<td>Recording Output (record output)</td>
<td>5-26</td>
</tr>
<tr>
<td>Playing Back the Input (source or playback input)</td>
<td>5-27</td>
</tr>
<tr>
<td>Playing Back the Output (playback output)</td>
<td>5-28</td>
</tr>
<tr>
<td>Invoking a Shell from DBX (sh)</td>
<td>5-29</td>
</tr>
<tr>
<td>Checking the Status (status)</td>
<td>5-29</td>
</tr>
<tr>
<td>Deleting Status Items (delete)</td>
<td>5-30</td>
</tr>
<tr>
<td>Examining Source Programs</td>
<td>5-31</td>
</tr>
<tr>
<td>Specifying Source Directories (use)</td>
<td>5-31</td>
</tr>
<tr>
<td>Moving to a Specified Procedure (func)</td>
<td>5-31</td>
</tr>
<tr>
<td>Specifying Source Files (file)</td>
<td>5-32</td>
</tr>
<tr>
<td>Listing Your Source Code (list)</td>
<td>5-33</td>
</tr>
<tr>
<td>Searching Through the Code (/ and ?)</td>
<td>5-34</td>
</tr>
<tr>
<td>Calling an Editor from DBX (edit)</td>
<td>5-34</td>
</tr>
<tr>
<td>Printing Symbolic Names (which and whereis)</td>
<td>5-35</td>
</tr>
<tr>
<td>Printing Type Declarations (whatis)</td>
<td>5-35</td>
</tr>
</tbody>
</table>
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
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement Extensions</td>
<td>B–5</td>
</tr>
<tr>
<td>Otherwise Clause in Case Statement</td>
<td>B–5</td>
</tr>
<tr>
<td>Return Statement</td>
<td>B–5</td>
</tr>
<tr>
<td>Continue Statement</td>
<td>B–6</td>
</tr>
<tr>
<td>Break Statement</td>
<td>B–6</td>
</tr>
<tr>
<td>Declaration Extensions</td>
<td>B–6</td>
</tr>
<tr>
<td>Separate Compilation</td>
<td>B–6</td>
</tr>
<tr>
<td>Shared Variables</td>
<td>B–8</td>
</tr>
<tr>
<td>Initialization Clauses</td>
<td>B–9</td>
</tr>
<tr>
<td>Relax Declaration Ordering</td>
<td>B–10</td>
</tr>
<tr>
<td>Predefined Procedures</td>
<td>B–10</td>
</tr>
<tr>
<td>Assert</td>
<td>B–10</td>
</tr>
<tr>
<td>Argv</td>
<td>B–10</td>
</tr>
<tr>
<td>Date</td>
<td>B–10</td>
</tr>
<tr>
<td>Time</td>
<td>B–10</td>
</tr>
<tr>
<td>Predefined Functions</td>
<td>B–10</td>
</tr>
<tr>
<td>Type Functions</td>
<td>B–10</td>
</tr>
<tr>
<td>Min</td>
<td>B–11</td>
</tr>
<tr>
<td>Max</td>
<td>B–11</td>
</tr>
<tr>
<td>Lbound</td>
<td>B–11</td>
</tr>
<tr>
<td>Hbound</td>
<td>B–11</td>
</tr>
<tr>
<td>First</td>
<td>B–12</td>
</tr>
<tr>
<td>Last</td>
<td>B–12</td>
</tr>
<tr>
<td>Argc</td>
<td>B–12</td>
</tr>
<tr>
<td>Clock</td>
<td>B–12</td>
</tr>
<tr>
<td>Bitand</td>
<td>B–12</td>
</tr>
<tr>
<td>Bitor</td>
<td>B–12</td>
</tr>
<tr>
<td>Bitxor</td>
<td>B–12</td>
</tr>
<tr>
<td>Bitnot</td>
<td>B–12</td>
</tr>
<tr>
<td>Lshift</td>
<td>B–13</td>
</tr>
<tr>
<td>Rshift</td>
<td>B–13</td>
</tr>
<tr>
<td>I/O Extensions</td>
<td>B–13</td>
</tr>
<tr>
<td>Specifying Radix in the Write Statement</td>
<td>B–13</td>
</tr>
<tr>
<td>Filename on Rewrite and Reset</td>
<td>B–13</td>
</tr>
<tr>
<td>Reading Character Strings</td>
<td>B–13</td>
</tr>
<tr>
<td>Reading and Writing Enumeration Types</td>
<td>B–14</td>
</tr>
<tr>
<td>Lazy I/O</td>
<td>B–15</td>
</tr>
<tr>
<td>Standard Error</td>
<td>B–15</td>
</tr>
<tr>
<td>Predefined Data Type Extensions</td>
<td>B–15</td>
</tr>
<tr>
<td>Double</td>
<td>B–15</td>
</tr>
<tr>
<td>Cardinal</td>
<td>B–15</td>
</tr>
<tr>
<td>Pointer</td>
<td>B–15</td>
</tr>
<tr>
<td>Compiler Notes</td>
<td>B–15</td>
</tr>
<tr>
<td>Macro Preprocessor</td>
<td>B–15</td>
</tr>
<tr>
<td>Short Circuiting</td>
<td>B–16</td>
</tr>
<tr>
<td>Translation Limits</td>
<td>B–16</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Task</th>
<th>Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write &amp; Edit Programs</td>
<td>UNIX System Editors</td>
</tr>
<tr>
<td>Compile, Link, &amp; Load Programs</td>
<td>The Link Editor &amp; the Compiler Drivers</td>
</tr>
<tr>
<td>Debug Programs</td>
<td>The Symbolic Debugger</td>
</tr>
<tr>
<td>Profile Programs</td>
<td>The Profiler</td>
</tr>
<tr>
<td>Optimize Programs</td>
<td>The Optimizer</td>
</tr>
<tr>
<td>Examine Object Files</td>
<td>The nm, file, size, &amp; dump Tools</td>
</tr>
<tr>
<td>Produce Necessary Libraries</td>
<td>The Archiver</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Language</th>
<th>Driver Name</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>cc</td>
<td>[compiler options]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[link editor options]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[source name list]</td>
</tr>
<tr>
<td>Pascal</td>
<td>pc</td>
<td>[compiler options]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[link editor options]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[source name list]</td>
</tr>
<tr>
<td>FORTRAN 77</td>
<td>f77</td>
<td>[compiler options]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[link editor options]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[source name list]</td>
</tr>
<tr>
<td>COBOL</td>
<td>cobol</td>
<td>[compiler options]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[link editor options]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[source name list]</td>
</tr>
<tr>
<td>PL/I</td>
<td>pl1</td>
<td>[compiler options]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[link editor options]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[source name list]</td>
</tr>
<tr>
<td>MIPS Assembly</td>
<td>as</td>
<td>[compiler options]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[source name list]</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.p</td>
<td>Pascal source code</td>
</tr>
<tr>
<td>.u</td>
<td>ucode object file</td>
</tr>
<tr>
<td>.a</td>
<td>object library</td>
</tr>
<tr>
<td>.b</td>
<td>ucode object library</td>
</tr>
<tr>
<td>.c</td>
<td>C source code</td>
</tr>
<tr>
<td>.cob</td>
<td>COBOL source code</td>
</tr>
<tr>
<td>.e</td>
<td>elf source</td>
</tr>
<tr>
<td>.f</td>
<td>Fortran 77 source</td>
</tr>
<tr>
<td>.i</td>
<td>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:</td>
</tr>
<tr>
<td></td>
<td>pc -c source.i</td>
</tr>
<tr>
<td></td>
<td>source.i is assumed to contain Pascal source statements.</td>
</tr>
<tr>
<td>.o</td>
<td>object file</td>
</tr>
<tr>
<td>pl1 or pli</td>
<td>PL/1 source code</td>
</tr>
<tr>
<td>.r</td>
<td>ratfor source code</td>
</tr>
<tr>
<td>.s</td>
<td>assembly source code</td>
</tr>
</tbody>
</table>

**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 elf(1), ratfor(1), and m4(1) manual pages in the User's Reference Manual.
Figure 1.1. The Compiler System Driver.
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:

\[
\begin{align*}
\textsc{cc} & \ 	ext{foo.c bar.c} & \text{runs the C compiler, creates object modules foo.o and bar.o, and the executable program a.out.}
\end{align*}
\]

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 \texttt{-c} option, which stops the driver immediately after the assembler phase. For example:

\[
\begin{align*}
\textsc{cc} & \ -c \ 	ext{main.c more.c} \\
\textsc{pc} & \ -c \ 	ext{rest.p}
\end{align*}
\]

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 \texttt{pc}, as shown below:

\[
\begin{align*}
\textsc{pc} & \ -o \ 	ext{all main.o more.o rest.o}
\end{align*}
\]
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 p11(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 p11(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.

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-S</td>
<td>Use the System V compatible include files and libraries instead of the 4.3 BSD default include files and libraries.</td>
</tr>
<tr>
<td>-c</td>
<td>Prevents the link editor from linking your program after compilation. This option forces the compiler to produce a .o file when you compile only one program.</td>
</tr>
<tr>
<td>-cord</td>
<td>Re-arranges the procedures in the link-edit object file to reduce cache conflicts in the executable object (a.out). At least one -feedback file must be specified. See cache conflicts and the -cord option profiling for more information.</td>
</tr>
<tr>
<td>-cpp</td>
<td>Run the C macro preprocessor on the source code before compiling. The default varies from driver to driver. Refer to the appropriate man page (cc, pc, as, etc) in the User’s Reference Manual for more information.</td>
</tr>
<tr>
<td>-C</td>
<td>C and assembler drivers only. Used with the -P and -E 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.</td>
</tr>
<tr>
<td>-C</td>
<td>Pascal and FORTRAN drivers only. Generates code that causes range checking for subscripts during program execution.</td>
</tr>
<tr>
<td>-feedback file</td>
<td>Produces (together with the -cord option) an object with procedures rearranged so as to reduce cache conflicts; file is the output produce when running -prof with the -prof -feedback option specified. See cache conflicts and the -cord option for details.</td>
</tr>
<tr>
<td>-E</td>
<td>Runs only the C macro preprocessor and sends results to the standard output. Specify also -C to retain comments. Use -E when you suspect the preprocessor isn’t emitting the intended code.</td>
</tr>
<tr>
<td>-D name or -D name=def</td>
<td>Defines a macro name if you specified a #define in your program. Unless you specify a definition name=def, the compiler defines the name to be &quot;1&quot;.</td>
</tr>
<tr>
<td>Option Name</td>
<td>Purpose</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td>-G num</td>
<td><em>num</em> 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 <em>num</em> to control the number of data items placed in these sections. See Limiting the Size of Global Pointer Data in Chapter 4 for examples of using –bestGnum and its related options.</td>
</tr>
<tr>
<td>-I dirname</td>
<td>Compiler searches the current directory <em>dirname</em>, and the default directory, /usr/include, in that order for the include file.</td>
</tr>
<tr>
<td>-I</td>
<td>When specified in addition to the –I dirname, the compiler searches only <em>dirname</em> and does not search the default directory.</td>
</tr>
<tr>
<td>-j</td>
<td>Creates a file suffixed with a .u 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.</td>
</tr>
<tr>
<td>-k option</td>
<td><em>option</em> 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 <em>option</em>.</td>
</tr>
<tr>
<td>-ko filename</td>
<td><em>filename</em> is the name of the output file to be created by the ucode loader.</td>
</tr>
<tr>
<td>-nocpp</td>
<td>Do not run the C macro preprocessor on C and assembly source files before processing. See also the –cpp option.</td>
</tr>
<tr>
<td>-o filename</td>
<td>Assigns the name <em>filename</em> to the program object. When used with the –e option, tells where to leave .o file. The default filename is a.out.</td>
</tr>
<tr>
<td>-P</td>
<td>Same as –E options, except puts results in a .i file. Specify both –P and –C to retain comments.</td>
</tr>
</tbody>
</table>

*General Compiler Options (2 of 3).*
### General Compiler Options

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-p1</code> or <code>-p</code></td>
<td>Permits program counter (pc) sampling. This option provides operational statistics for use in improving program performance. See Chapter 4 for more details. <strong>Note:</strong> 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.</td>
</tr>
<tr>
<td><code>-S</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>-std</code></td>
<td>Issues a warning message when the compiler finds a non-standard feature in the programming language of your source program.</td>
</tr>
<tr>
<td><code>-U name</code></td>
<td>Overrides a definition of a macro name that you specified with the <code>-D</code> option, or that is defined automatically by the driver.</td>
</tr>
<tr>
<td><code>-v</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>-V</code></td>
<td>Prints the version number of the driver and its phases. When reporting a suspected compiler problem, you must include this number.</td>
</tr>
<tr>
<td><code>-w</code></td>
<td>Suppresses warning messages.</td>
</tr>
</tbody>
</table>

*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:

<table>
<thead>
<tr>
<th>Byte Ordering Options</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Option Name</strong></td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>-EB</td>
</tr>
<tr>
<td>-EL</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Debugging Options</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Option Name</strong></td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>-g0*</td>
</tr>
<tr>
<td>-g1</td>
</tr>
<tr>
<td>-g or -g2</td>
</tr>
<tr>
<td>-g3</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
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.

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

*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 `p11(1)` in the *User’s Reference Manual*. 

Languages Programmer’s Guide
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.
<table>
<thead>
<tr>
<th>Option Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-A file</td>
<td>This option produces an object that may be read into an already existing program. The argument, file, 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.</td>
</tr>
<tr>
<td>-b</td>
<td>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.</td>
</tr>
<tr>
<td>-B num</td>
<td>Sets the starting address of the uninitialized data segment (bss) to the hexadecimal address num. This option is valid only when you've also specified the -N link editor option described later in this table.</td>
</tr>
<tr>
<td>-Bstring</td>
<td>Append string to the library name created by the -lx or -klx option. The library is searched both with and without string.</td>
</tr>
<tr>
<td>-bestGnum</td>
<td>See -G on the next page.</td>
</tr>
<tr>
<td>-D num</td>
<td>Sets the starting address of the data segment (data) to the hexadecimal address num. This option is valid only when you've also specified the -N link editor option.</td>
</tr>
<tr>
<td>-e epsym</td>
<td>Sets the default entry point address for the output file to the specified symbol epsym.</td>
</tr>
<tr>
<td>-EB</td>
<td>Uses big-endian byte ordering when writing out header and symbol table entries.</td>
</tr>
<tr>
<td>-EL</td>
<td>Uses little-endian byte ordering when writing out header and symbol table entries.</td>
</tr>
<tr>
<td>-f fill</td>
<td>Sets the fill pattern for &quot;holes&quot; within an output section of an object file; fill is four-byte hexadecimal constant that defines the fill pattern.</td>
</tr>
</tbody>
</table>

Table 1.1 (1 of 4). Link Editor Options.
<table>
<thead>
<tr>
<th>Option Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-F or -z</td>
<td>Creates a ZMAGIC file (an object file that loads on demand) This is the default. See Chapter 10 of the Assembly Language Guide for more information on ZMAGIC files.</td>
</tr>
<tr>
<td>-G num</td>
<td>Specifies the maximum size (in decimal bytes) of a <code>.comm</code> item that should be allocated in the small uninitialized data (<code>sbrs</code>) section for reference by the global pointers. The default is 8 bytes.</td>
</tr>
<tr>
<td>-bestGnum</td>
<td>Prints the optimum value to be specified as the <code>num</code> value for <code>-G</code>.</td>
</tr>
<tr>
<td>-count</td>
<td>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 <code>-bestGnum</code> option. For example, you would exclude any object for which you did not have the source code for recompilation. The options are explained below.</td>
</tr>
<tr>
<td>-nocount</td>
<td>Objects that follow on the command line cannot be recompiled.</td>
</tr>
<tr>
<td>-countall</td>
<td>Overides any <code>-nocount</code> option appearing after it on the command line.</td>
</tr>
<tr>
<td>-jmpopt</td>
<td>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 <code>-g1</code>, <code>-g2</code> or <code>-g</code> flag is present. When enabled, this option can cause an out-of-memory condition in the link editor.</td>
</tr>
</tbody>
</table>

See Limiting the Size of Global Pointer Data in Chapter 4 for examples of using the `-bestGnum` and related count options.

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

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Purpose</th>
</tr>
</thead>
</table>
| `-lx`       | Specifies the name of a link library, where `x` is the library name. The link editor searches for `libx.a` in `/lib`, `/usr/lib`, and `/usr/local/lib` directories respectively. For example, if you specify `curses`, the library pathnames can be:  
/`lib/curses.a`
/`usr/lib/curses.a`
/`usr/local/lib/curses.a`  
If a library relies on procedures or data from another library, specify that library's name first.  
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.  
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 `Id(1)` manual page in the *User's Reference Manual* for more information. |
| `-L dirname` | Searches `dirname` for libraries specified in the `-lx` option before searching directories `/lib`, `/usr/lib`, and `/usr/local/lib`.  
This option must precede the `-lx` option. |
| `-L`        | If the link editor doesn’t find the library in `dirname`, then `/lib`, `/usr/lib`, and `/usr/local/lib` are NOT searched.  
A `-L dirname` option must be specified with `-L`. |
| `-m`        | Produces a link editor memory map in System V format. |
| `-M`        | Produces a link editor memory map in BSD format. |
| `-n`        | Creates an NMAGIC* file. The text segment is read-only and shareable by all users of the file. |
| `-N`        | Creates an OMAGIC* file. The text segment isn’t readable and shareable by other users. The data segment follows immediately; after the text segment. |

*See Chapter 10 of the Assembly Language Programmer's Guide for more information on NMAGIC and OMAGIC files.*

---

*Table 1.1 (3 of 4). Link Editor Options.*
<table>
<thead>
<tr>
<th>Option Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-o filename</td>
<td>Specifies a name for your object file. If you don’t specify a name, the link editor uses a.out as the default.</td>
</tr>
<tr>
<td>-p file</td>
<td>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 Optimizing Frequently Used Modules in Chapter 4 for an example.</td>
</tr>
<tr>
<td>-r</td>
<td>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.</td>
</tr>
<tr>
<td>-s</td>
<td>Strips symbol table information from the program object, reducing its size.</td>
</tr>
<tr>
<td>-S</td>
<td>Suppresses non–fatal error reporting.</td>
</tr>
<tr>
<td>-T num</td>
<td>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 MIPS Assembly Language Programmer’s Guide.</td>
</tr>
<tr>
<td>-u symname</td>
<td>Makes symname undefined so that library components that define symname are loaded.</td>
</tr>
<tr>
<td>-v</td>
<td>Prints the name of each file as it is processed by the link editor.</td>
</tr>
<tr>
<td>-V</td>
<td>Prints the link editor version number. You might need this number, for example, when reporting a suspected bug in the link editor.</td>
</tr>
<tr>
<td>-VS num</td>
<td>Puts the specified decimal version stamp num in the object file that the link editor produces.</td>
</tr>
<tr>
<td>-x</td>
<td>Retains external and static symbols in the symbol table to allow some debugging facilities. Doesn’t retain local (non–global) symbols.</td>
</tr>
</tbody>
</table>

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

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

*Table 1.2 (1 of 2). Odump Options.*
<table>
<thead>
<tr>
<th>Option Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>−d number</td>
<td>Dumps the section number, or a range of section numbers, that starts at the specified number and that ends with the last section number or the number you specify with the +d auxiliary option.</td>
</tr>
<tr>
<td>+d number</td>
<td>Dumps the sections in a range that starts with the first section or with the section you specify with the −d option.</td>
</tr>
<tr>
<td>−n name</td>
<td>Dumps information only for the named entry name. Use this option with the −h, −s, −r, −l and −t options.</td>
</tr>
<tr>
<td>−p</td>
<td>Suppresses the printing of headers.</td>
</tr>
<tr>
<td>−t index</td>
<td>Dumps only the indexed symbol table entry. You can specify a range of table entries by using the −t option with the +t option.</td>
</tr>
<tr>
<td>+t index</td>
<td>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.</td>
</tr>
<tr>
<td>−v</td>
<td>Dumps information in symbolic rather than numeric representation (for example, in Static rather than 0X02). Use this option with all dump options except −s.</td>
</tr>
<tr>
<td>−z name,number</td>
<td>Dumps the line number entry or a range of entries that start at the specified number for the named function.</td>
</tr>
<tr>
<td>+z number</td>
<td>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.</td>
</tr>
</tbody>
</table>

Table 1.2 (2 of 2). Odump Options
***STRING TABLE INFORMATION***

```plaintext
[Offset] Name
sam.o:
[ 1]   sam.c
[ 2]   line
[ 3]   string
[ 4]   length
[ 5]   linenum
[ 6]   LINETYPE
[ 7]   main
[ 8]   argc
[ 9]   argv
[10]   linel
[12]   i
[13]   curlinenumber
[14]   printline
[15]   pline
[16]   i
[17]   /usr/local/mips/include/stdio.h
[18]   _iobuf
[19]   _cnt
[20]   _ptr
[21]   _base
[22]   _bufsiz
[23]   _flag
[24]   _file
[25]   _name
```

% odump -c sam.o

---

***FILE HEADER***

```plaintext
<table>
<thead>
<tr>
<th>Magic</th>
<th>Ncons</th>
<th>Time/Date</th>
<th>Symptr</th>
<th>Nsyms</th>
<th>Opt hdr</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000540</td>
<td>2</td>
<td>0x1f22b375</td>
<td>0x00000344</td>
<td>96</td>
<td>0x0038</td>
<td>0x0000</td>
</tr>
</tbody>
</table>
```

% odump -F sam.o

***FILE DESCRIPTOR TABLE***

```plaintext
<table>
<thead>
<tr>
<th>Filename</th>
<th>lnOffset</th>
<th>iBase/count</th>
<th>merge</th>
<th>sex</th>
<th>address</th>
<th>cbLine</th>
<th>sym</th>
<th>line</th>
<th>pd</th>
<th>aux</th>
<th>xfd</th>
<th>language</th>
</tr>
</thead>
<tbody>
<tr>
<td>sam.o:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sam.c</td>
<td>0x00000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>---</td>
<td>C</td>
<td></td>
<td></td>
<td>e1</td>
</tr>
<tr>
<td>ps/include/stdio.h</td>
<td>0x00000000</td>
<td>23</td>
<td>27</td>
<td>103</td>
<td>2</td>
<td>40</td>
<td>0</td>
<td>merge</td>
<td>C</td>
<td></td>
<td></td>
<td>e1</td>
</tr>
</tbody>
</table>
```

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

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

<table>
<thead>
<tr>
<th>Vaddr</th>
<th>Symndx</th>
<th>Type</th>
<th>Extern</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000034</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0x00000038</td>
<td>1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>0x00000040</td>
<td>0</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>0x00000044</td>
<td>1</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>0x00000050</td>
<td>0</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>0x00000058</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0x00000078</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0x0000007c</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>0x00000088</td>
<td>0</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>0x00000084</td>
<td>1</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>0x0000008c</td>
<td>1</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>0x00000094</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>0x000000ac</td>
<td>1</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>0x000000ec</td>
<td>1</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>0x000000fc</td>
<td>0</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>0x00000100</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0x00000110</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>0x0000015c</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0x00000160</td>
<td>1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>0x00000168</td>
<td>0</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>0x00000170</td>
<td>1</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>0x00000178</td>
<td>1</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>0x00000180</td>
<td>1</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>0x0000017c</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

***RELATIVE FILE INDEX TABLE***

```
% odump -r sam.o
```

```
sam.o:  [0 for 0]
/usr/local/mips/include/stdio.h[0 for 0]
```

***SECTION DATA in HEX***

```
% odump -s sam.o
```

```
27BD  FED0  AFBF  0014  AFA4  0130  AFA5  0134  AF00  0010  8FA4  0130
0000  0000  AFA5  0020  8FA5  0020  0000  0000  2951  0002  1020  0007
0000  0000  3C01  0000  2424  0000  0C00  0000  2785  0010  0C00  0000
2004  0001  8E89  0134  2785  8024  8F04  0004  0C00  0000  0000  0000
AFA2  0024  8F99  0024  0000  0000  1720  0009  0000  0000  0000  8FA8  0134
3C01  0000  2424  0000  8D06  0004  0C00  0000  2785  8026  0C00  0000
2004  0001  27A4  0024  8FA6  0024  0C00  0000  2005  0100  1060  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

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>Value</th>
<th>Sclass</th>
<th>Syntype</th>
<th>Ref</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sam.c</td>
<td>0x00000000</td>
<td>0x01</td>
<td>0x0b</td>
<td>0x001b</td>
</tr>
<tr>
<td>1</td>
<td>line</td>
<td>0x00000010</td>
<td>0x0b</td>
<td>0x07</td>
<td>0x0006</td>
</tr>
<tr>
<td>2</td>
<td>string</td>
<td>0x00000000</td>
<td>0x0b</td>
<td>0x09</td>
<td>0x0000e</td>
</tr>
<tr>
<td>3</td>
<td>length</td>
<td>0x00000000</td>
<td>0x0b</td>
<td>0x09</td>
<td>0x0004</td>
</tr>
<tr>
<td>4</td>
<td>linenum</td>
<td>0x00000000</td>
<td>0x0b</td>
<td>0x09</td>
<td>0x0004</td>
</tr>
<tr>
<td>5</td>
<td>LINTYPE</td>
<td>0x00000000</td>
<td>0x0b</td>
<td>0x08</td>
<td>0x0001</td>
</tr>
<tr>
<td>6</td>
<td>main</td>
<td>0x00000000</td>
<td>0x01</td>
<td>0x06</td>
<td>0x0017</td>
</tr>
<tr>
<td>7</td>
<td>argc</td>
<td>0x00000000</td>
<td>0x05</td>
<td>0x03</td>
<td>0x0004</td>
</tr>
<tr>
<td>8</td>
<td>argv</td>
<td>0x00000000</td>
<td>0x05</td>
<td>0x03</td>
<td>0x0019</td>
</tr>
<tr>
<td>9</td>
<td>linel</td>
<td>0x00000000</td>
<td>0x01</td>
<td>0x07</td>
<td>0x0013</td>
</tr>
<tr>
<td>10</td>
<td>fd</td>
<td>0x00000000</td>
<td>0x05</td>
<td>0x04</td>
<td>0x001a</td>
</tr>
<tr>
<td>11</td>
<td>l</td>
<td>0x00000000</td>
<td>0x05</td>
<td>0x04</td>
<td>0x0004</td>
</tr>
<tr>
<td>12</td>
<td>main</td>
<td>0x00000000</td>
<td>0x01</td>
<td>0x07</td>
<td>0x0012</td>
</tr>
<tr>
<td>13</td>
<td>main</td>
<td>0x00000000</td>
<td>0x01</td>
<td>0x07</td>
<td>0x000a</td>
</tr>
<tr>
<td>14</td>
<td>main</td>
<td>0x00000000</td>
<td>0x01</td>
<td>0x07</td>
<td>0x0007</td>
</tr>
<tr>
<td>15</td>
<td>main</td>
<td>0x00000000</td>
<td>0x01</td>
<td>0x07</td>
<td>0x0015</td>
</tr>
<tr>
<td>16</td>
<td>main</td>
<td>0x00000000</td>
<td>0x01</td>
<td>0x06</td>
<td>0x0015</td>
</tr>
</tbody>
</table>

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 errno
0000cd34 T cleanup
00004ae8 D ctype
0001fa0 T dprint
0000de4 T exit
0001878 T filbuf
0000990 T filbuf
000c560 N gp
0004228 D lcb
0004598 G lastbuf
001f44 t lowdigit
```

```
<table>
<thead>
<tr>
<th>value field</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td></td>
</tr>
<tr>
<td>(see Table 1.3)</td>
<td></td>
</tr>
</tbody>
</table>
```

*Figure 1.4. Symbol Table in BSD Format (option –B)*
The meanings of the character keys shown in an `nm` listing are described below.

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

*Table 1.3. nm Character Key Meanings.*
Symbols from sam.o:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value 1</th>
<th>Class 1</th>
<th>Type</th>
<th>Size</th>
<th>Index</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>sam.c</td>
<td>000000000</td>
<td>File</td>
<td>ref=27</td>
<td></td>
<td>0</td>
<td>Text</td>
</tr>
<tr>
<td>line</td>
<td>000000264</td>
<td>Block</td>
<td>ref=6</td>
<td></td>
<td>1</td>
<td>Info</td>
</tr>
<tr>
<td>string</td>
<td>000000200</td>
<td>Member</td>
<td>unsigned char [256]</td>
<td></td>
<td>2</td>
<td>Info</td>
</tr>
<tr>
<td>length</td>
<td>00002048</td>
<td>Member</td>
<td>int</td>
<td></td>
<td>3</td>
<td>Info</td>
</tr>
<tr>
<td>linenumber</td>
<td>00002080</td>
<td>Member</td>
<td>int</td>
<td></td>
<td>4</td>
<td>Info</td>
</tr>
<tr>
<td>LINETYPE</td>
<td>00000000</td>
<td>End</td>
<td>ref=1</td>
<td></td>
<td>5</td>
<td>Info</td>
</tr>
<tr>
<td>main</td>
<td>000000000</td>
<td>Proc</td>
<td>end=20 int</td>
<td></td>
<td>7</td>
<td>Text</td>
</tr>
<tr>
<td>argc</td>
<td>000000000</td>
<td>Param</td>
<td>int</td>
<td></td>
<td>8</td>
<td>Abs</td>
</tr>
<tr>
<td>argv</td>
<td>000000000</td>
<td>Param</td>
<td>unsigned char **</td>
<td></td>
<td>9</td>
<td>Abs</td>
</tr>
<tr>
<td>linel</td>
<td>-0000264</td>
<td>Local</td>
<td>struct line</td>
<td></td>
<td>10</td>
<td>Text</td>
</tr>
<tr>
<td>fd</td>
<td>-0000268</td>
<td>Local</td>
<td>struct _iobuf</td>
<td></td>
<td>11</td>
<td>Abs</td>
</tr>
<tr>
<td>i</td>
<td>-0000272</td>
<td>Local</td>
<td>int</td>
<td></td>
<td>12</td>
<td>Abs</td>
</tr>
<tr>
<td>i</td>
<td>-0000280</td>
<td>Local</td>
<td>int</td>
<td></td>
<td>13</td>
<td>Abs</td>
</tr>
<tr>
<td>curlinenumber</td>
<td>000000456</td>
<td>Static</td>
<td>int</td>
<td></td>
<td>14</td>
<td>Text</td>
</tr>
<tr>
<td>main</td>
<td>000000264</td>
<td>End</td>
<td>ref=14</td>
<td></td>
<td>15</td>
<td>Abs</td>
</tr>
<tr>
<td>getline</td>
<td>000000828</td>
<td>End</td>
<td>ref=10</td>
<td></td>
<td>16</td>
<td>Text</td>
</tr>
<tr>
<td>main</td>
<td>00000312</td>
<td>End</td>
<td>ref=7</td>
<td></td>
<td>17</td>
<td>Text</td>
</tr>
<tr>
<td>printline</td>
<td>00000312</td>
<td>Proc</td>
<td>end=26 btNil</td>
<td></td>
<td>18</td>
<td>Text</td>
</tr>
<tr>
<td>pline</td>
<td>000000000</td>
<td>Param</td>
<td>struct line</td>
<td></td>
<td>19</td>
<td>Abs</td>
</tr>
<tr>
<td>i</td>
<td>-0000004</td>
<td>Local</td>
<td>int</td>
<td></td>
<td>20</td>
<td>Abs</td>
</tr>
<tr>
<td>printline</td>
<td>00000100</td>
<td>End</td>
<td>ref=20</td>
<td></td>
<td>21</td>
<td>Abs</td>
</tr>
<tr>
<td>sam.c</td>
<td>000000000</td>
<td>End</td>
<td>ref=0</td>
<td></td>
<td>22</td>
<td>Abs</td>
</tr>
<tr>
<td>/usr/local/mips/include</td>
<td>000000000</td>
<td>File</td>
<td>ref=38</td>
<td></td>
<td>23</td>
<td>Abs</td>
</tr>
<tr>
<td>_iobuf</td>
<td>0000024</td>
<td>Block</td>
<td>ref=37</td>
<td></td>
<td>24</td>
<td>Abs</td>
</tr>
<tr>
<td>_cnt</td>
<td>000000000</td>
<td>Member</td>
<td>int</td>
<td></td>
<td>25</td>
<td>Abs</td>
</tr>
<tr>
<td>_ptr</td>
<td>00000032</td>
<td>Member</td>
<td>unsigned char *</td>
<td></td>
<td>26</td>
<td>Abs</td>
</tr>
<tr>
<td>_base</td>
<td>00000064</td>
<td>Member</td>
<td>unsigned char *</td>
<td></td>
<td>27</td>
<td>Abs</td>
</tr>
<tr>
<td>_buksz</td>
<td>00000096</td>
<td>Member</td>
<td>int</td>
<td></td>
<td>28</td>
<td>Abs</td>
</tr>
<tr>
<td>_flag</td>
<td>00000128</td>
<td>Member</td>
<td>short</td>
<td></td>
<td>29</td>
<td>Abs</td>
</tr>
<tr>
<td>_file</td>
<td>00000144</td>
<td>Member</td>
<td>unsigned char</td>
<td></td>
<td>30</td>
<td>Abs</td>
</tr>
<tr>
<td>_name</td>
<td>00000160</td>
<td>Member</td>
<td>unsigned char *</td>
<td></td>
<td>31</td>
<td>Abs</td>
</tr>
</tbody>
</table>

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).
<table>
<thead>
<tr>
<th>Option Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-A</code></td>
<td>Prints the listing in System V format. The default format is that of your operating system.</td>
</tr>
<tr>
<td><code>-B</code></td>
<td>Prints the listing in BSD format. The default format is that of your operating system.</td>
</tr>
<tr>
<td><code>-a</code></td>
<td>Prints debugging information (turns BSD output into System V format).</td>
</tr>
<tr>
<td><code>-b</code></td>
<td>Prints the value field in octal.</td>
</tr>
<tr>
<td><code>-d</code></td>
<td>Prints the value field in decimal (the default for System V output).</td>
</tr>
<tr>
<td><code>-e</code></td>
<td>Prints only external and static variables.</td>
</tr>
<tr>
<td><code>-h</code></td>
<td>Suppresses printing of headers.</td>
</tr>
<tr>
<td><code>-n</code></td>
<td>Sorts external symbols by name for System V format. Sorts all symbols by value for Berkeley format (by name is the BSD default output).</td>
</tr>
<tr>
<td><code>-o</code></td>
<td>Prints value field in octal (System V output). Prints the filename immediately before each symbol name (BSD output).</td>
</tr>
<tr>
<td><code>-p</code></td>
<td>Lists symbols in the order they appear in the Symbol table.</td>
</tr>
<tr>
<td><code>-r</code></td>
<td>Reverses the sort that you specified for external symbols with the <code>-n</code> and <code>-v</code> options.</td>
</tr>
<tr>
<td><code>-T</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>-u</code></td>
<td>Prints only undefined symbols.</td>
</tr>
<tr>
<td><code>-v</code></td>
<td>Sorts external symbols by value (default for Berkeley format).</td>
</tr>
<tr>
<td><code>-V</code></td>
<td>Prints the version number of <code>nm</code>.</td>
</tr>
<tr>
<td><code>-x</code></td>
<td>Prints the value field in hexadecimal.</td>
</tr>
</tbody>
</table>

*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 4096164437 184917 2d255
```
### Size Options

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-A</td>
<td>Prints data section headers in System V format. The default format is determined by the UNIX version running at your installation.</td>
</tr>
<tr>
<td>-B</td>
<td>Prints data section headers in Berkeley format. The default format is determined by the UNIX version running at your installation.</td>
</tr>
<tr>
<td>-d</td>
<td>Prints the section sizes in decimal.</td>
</tr>
<tr>
<td>-o</td>
<td>Prints the section sizes in octal.</td>
</tr>
<tr>
<td>-V</td>
<td>Prints the version of size that you are using.</td>
</tr>
<tr>
<td>-x</td>
<td>Prints the section sizes in hexadecimal.</td>
</tr>
</tbody>
</table>

*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.
<table>
<thead>
<tr>
<th>Option Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Suppresses the warning message that the archiver issues when it discovers that the archive you specified doesn't already exist.</td>
</tr>
<tr>
<td>d</td>
<td>Deletes the specified objects from the archive.</td>
</tr>
<tr>
<td>l</td>
<td>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.</td>
</tr>
<tr>
<td>m</td>
<td>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 posObject parameter.</td>
</tr>
<tr>
<td>p</td>
<td>Prints the specified object(s) in the archive on the standard output device (usually the terminal screen).</td>
</tr>
<tr>
<td>q</td>
<td>Adds the specified object files to the end of the archive. An existing object file with the same name is not 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.</td>
</tr>
<tr>
<td>r</td>
<td>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 posObject parameter. See Example 4 in the preceding section for an example of using the posObject parameter. See also the u suboption. Use the r option when updating existing libraries.</td>
</tr>
</tbody>
</table>

*Table 1.6 (1 of 2). Archiver Options.*
### Archiver Options (Part 2)

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

*Table 1.6 (2 of 2). Archiver Options.*
The archiver has these suboptions:

<table>
<thead>
<tr>
<th>Suboption Name</th>
<th>Use with...</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>m or r</td>
<td>Specifies that the object file follow the posObject file you specify in the ar statement.¹</td>
</tr>
<tr>
<td>b</td>
<td>m or r</td>
<td>Specifies that the object file precede the posObject file you specify in the ar statement.¹</td>
</tr>
<tr>
<td>i</td>
<td>m or r</td>
<td>Same as -b²</td>
</tr>
<tr>
<td>o</td>
<td>x</td>
<td>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.</td>
</tr>
<tr>
<td>u</td>
<td>r</td>
<td>The archiver replaces the existing object file when the last modified data is earlier (precedes) that of the new object file.</td>
</tr>
</tbody>
</table>

See example 4 in the Examples section preceding this section for an example of the posObject 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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Alignment</th>
<th>Signed</th>
<th>Unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>32 bits</td>
<td>Word (^1)</td>
<td>(-2^{31}\text{ to }2^{31} - 1)</td>
<td>0 to (2^{32} - 1)</td>
</tr>
<tr>
<td>long</td>
<td>32 bits</td>
<td>Word (^1)</td>
<td>(-2^{31}\text{ to }2^{31} - 1)</td>
<td>0 to (2^{32} - 1)</td>
</tr>
<tr>
<td>enum</td>
<td>32 bits</td>
<td>Word (^1)</td>
<td>(-2^{31}\text{ to }2^{31} - 1)</td>
<td>0 to (2^{32} - 1)</td>
</tr>
<tr>
<td>short</td>
<td>16 bits</td>
<td>Halfword (^2)</td>
<td>(-32,768..32,767)</td>
<td>0..65,535</td>
</tr>
<tr>
<td>char(^4)</td>
<td>8 bits</td>
<td>Byte</td>
<td>(-128..127)</td>
<td>0..255</td>
</tr>
<tr>
<td>float(^5)</td>
<td>32 bits</td>
<td>Word (^1)</td>
<td>See note.</td>
<td>See note.</td>
</tr>
<tr>
<td>double(^6)</td>
<td>64 bits</td>
<td>Doubleword (^3)</td>
<td>See note.</td>
<td>See note.</td>
</tr>
<tr>
<td>pointer</td>
<td>32 bit</td>
<td>Word (^1)</td>
<td></td>
<td>0 to (2^{32} - 1)</td>
</tr>
</tbody>
</table>

\(^1\) 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:
<table>
<thead>
<tr>
<th>Maximum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
</tr>
<tr>
<td>3.40282356*10^{38}</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>1.7976931348623158*10^{38}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Minimum Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Denormalized</td>
</tr>
<tr>
<td>Normalized</td>
</tr>
<tr>
<td>float</td>
</tr>
<tr>
<td>1.40129846*10^{-46}</td>
</tr>
<tr>
<td>1.17549429*10^{-38}</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>4.9406564584124654*10^{-324}</td>
</tr>
<tr>
<td>2.2250738585072012*10^{308}</td>
</tr>
</tbody>
</table>

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:

```c
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:

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

the size of the resulting array is 48 (2*3*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:

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

is mapped out in storage as follows:

### Big Endian

<table>
<thead>
<tr>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>n0</th>
<th>n1</th>
<th>n2</th>
<th>n3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n4</th>
<th>n5</th>
<th>n6</th>
<th>n7</th>
<th>n8</th>
<th>n9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>

### Little Endian

<table>
<thead>
<tr>
<th>n9</th>
<th>n8</th>
<th>n7</th>
<th>n6</th>
<th>n5</th>
<th>n4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n3</th>
<th>n2</th>
<th>n1</th>
<th>n0</th>
<th>v</th>
<th>v</th>
<th>v</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

- Padded bytes

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:

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

the compiler maps the structures as follows:

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>n0</td>
<td>n1</td>
<td>n2</td>
<td>n3</td>
<td>n4</td>
<td>n5</td>
<td>n6</td>
<td>n7</td>
<td>n8</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

**Big Endian**

<table>
<thead>
<tr>
<th>Byte</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td></td>
<td>n9</td>
<td>n8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Little Endian**

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:

```c
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:

```c
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:

```c
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.

<table>
<thead>
<tr>
<th>Scalar Type</th>
<th>Value Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>0 or 1</td>
</tr>
<tr>
<td>char</td>
<td>0..127</td>
</tr>
<tr>
<td>integer</td>
<td>$2^{31}$..$2^{31}$-1</td>
</tr>
<tr>
<td>cardinal</td>
<td>0..$2^{30}$-1</td>
</tr>
<tr>
<td>real</td>
<td>See note 1.</td>
</tr>
<tr>
<td>double</td>
<td>See note 1.</td>
</tr>
</tbody>
</table>

*Table 2.2. Pascal Value Ranges.*

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

<table>
<thead>
<tr>
<th>Maximum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
</tr>
<tr>
<td>double</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Minimum Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denormalized</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>real</td>
</tr>
<tr>
<td>double</td>
</tr>
</tbody>
</table>

**NOTE 2:** Enumerated types with *n* elements are treated the same as the integer subrange 0..*n*-1.
<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>8</td>
<td>byte</td>
</tr>
<tr>
<td>char</td>
<td>8</td>
<td>byte</td>
</tr>
<tr>
<td>integer</td>
<td>32</td>
<td>word</td>
</tr>
<tr>
<td>cardinal</td>
<td>32</td>
<td>word</td>
</tr>
<tr>
<td>pointer</td>
<td>32</td>
<td>word</td>
</tr>
<tr>
<td>file</td>
<td>32</td>
<td>word</td>
</tr>
<tr>
<td>real</td>
<td>32</td>
<td>word</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>doubleword</td>
</tr>
<tr>
<td>subrange of</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0..255 or -128..127</td>
<td>8</td>
<td>byte</td>
</tr>
<tr>
<td>0..65535 or -32768..32767</td>
<td>16</td>
<td>halfword</td>
</tr>
<tr>
<td>0..2^32 -1</td>
<td>32</td>
<td>word</td>
</tr>
<tr>
<td>-2^31 .. -2^31 -1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>set of char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>set of char subrange</td>
<td>128</td>
<td>word</td>
</tr>
<tr>
<td>set of a..b</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>see NOTE</td>
<td>word</td>
</tr>
</tbody>
</table>

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

**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:

\[
\begin{align*}
& \text{if } b - \lfloor a/32 \rfloor \cdot 32 + 1 \leq 32 \text{ then} \\
& \quad \text{size} = b - \lfloor a/32 \rfloor \cdot 32 + 1 \text{ bits} \\
& \text{else} \\
& \quad \text{size} = (b/32) - \lfloor a/32 \rfloor + 1 \text{ bits}
\end{align*}
\]

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

<table>
<thead>
<tr>
<th>Packed Records</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar Type</td>
</tr>
<tr>
<td>boolean</td>
</tr>
<tr>
<td>char</td>
</tr>
<tr>
<td>integer</td>
</tr>
<tr>
<td>cardinal</td>
</tr>
<tr>
<td>pointer</td>
</tr>
<tr>
<td>file</td>
</tr>
<tr>
<td>real</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>subrange of</td>
</tr>
</tbody>
</table>

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 \( a..b \):

\[
\begin{align*}
& \text{If } a \geq 0 \text{ then } \text{size} = \left\lceil \log \left( \frac{b+1}{2} \right) \right\rceil \text{ bits} \\
& \text{If } a < 0 \text{ then } \text{size} = \max\left( \left\lceil \log \left( \frac{b+1}{2} \right) \right\rceil \log \left( \frac{-a}{2} \right) \right) + 1 \text{ bits}
\end{align*}
\]

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 speci-
fied for element of the array. The size of an array is the size of the data type mul-
tiplied by the number of elements. For example, for the following declaration:

\[
\text{var } x: \text{ array [1..2, 1..3] of double;}
\]

the size of the resulting array is 48 bytes (2*3*8, where 8 is the size of the dou-
ble 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:

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

is mapped out in storage as follows:

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>Byte</td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>15 14 13 12 11 10 9 8</td>
</tr>
<tr>
<td>n4 n5 n6 n7 n8 n9</td>
<td>n9 n8 n7 n6 n5 n4</td>
</tr>
<tr>
<td>8 9 10 11 12 13 14 15</td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>

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 divi-
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 $v: \text{integer}$ 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:

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

the compiler maps the structures as follows:

```
<table>
<thead>
<tr>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>15 14 13 12 11 10 9 8</td>
</tr>
<tr>
<td>n0 n1 n2 n3 n4 n5 n6 n7</td>
<td>v   v   v   v   n9 n8</td>
</tr>
<tr>
<td>Byte</td>
<td>Byte</td>
</tr>
<tr>
<td>8 9 10 11 12 13 14 15</td>
<td>n7   n6   n5   n4   n3   n2</td>
</tr>
<tr>
<td></td>
<td>n1   n0</td>
</tr>
</tbody>
</table>
```

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:

### Big Endian

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>page</td>
</tr>
<tr>
<td>Bit 31</td>
<td>19</td>
</tr>
<tr>
<td>Bit 30</td>
<td></td>
</tr>
</tbody>
</table>

### Little Endian

<table>
<thead>
<tr>
<th>Byte 1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>segment</td>
<td>page</td>
</tr>
<tr>
<td>Bit 31</td>
<td>22</td>
</tr>
<tr>
<td>Bit 30</td>
<td></td>
</tr>
</tbody>
</table>

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:

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

is mapped out as follows:

<table>
<thead>
<tr>
<th>BigEndian</th>
<th>31</th>
<th>28</th>
<th>23</th>
<th>16</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LittleEndian</th>
<th>31</th>
<th>15</th>
<th>7</th>
<th>3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td></td>
<td>b</td>
<td></td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

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 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:

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

is mapped out as follows:

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>Little Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>Padded bits.</td>
<td></td>
</tr>
</tbody>
</table>
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:

\[ \text{set of } a..b \]

\( a \) 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:

\[ \text{set of } 5..31 \]

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 \( \text{set of } a..b \) to be a valid set in Pascal, the following condition must be met:

\[ \text{size} = (b - 32 \lfloor a / 32 \rfloor + 1) \leq 512 \]

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

<table>
<thead>
<tr>
<th>Specification</th>
<th>Lower</th>
<th>Upper</th>
<th>Set size</th>
<th>Valid Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>set of 1..511</td>
<td>0*</td>
<td>511</td>
<td>512</td>
<td>Yes</td>
</tr>
<tr>
<td>set of 0..511</td>
<td>0</td>
<td>511</td>
<td>512</td>
<td>Yes</td>
</tr>
<tr>
<td>set of 1..512</td>
<td>0*</td>
<td>512</td>
<td>513</td>
<td>No</td>
</tr>
<tr>
<td>set of 31..512</td>
<td>0*</td>
<td>512</td>
<td>513</td>
<td>No</td>
</tr>
<tr>
<td>set of 32..512</td>
<td>32</td>
<td>512</td>
<td>481</td>
<td>Yes</td>
</tr>
<tr>
<td>set of 32..543</td>
<td>32</td>
<td>543</td>
<td>512</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*As padded down to by Pascal.
Language Interfaces

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:

```c
vars: 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 (\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.

```
{ This predefined Pascal function
  is a MIPS extension }
{ 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. }
Checks for null character.
```

For more information on `atoi`, see the `atof(3)` (for BSD) or `strtol(3c)` (for Sys-
```

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. }
  Checks for null character.
  for i := lbound(s) to hbound(s) do
    begin
      if s[i] = ' ' then
        begin
          s[i] := chr(0);
          break;
        end;
      writeln(output, atoi(s));
    end.
```

```

Variable number of arguments. C functions can be defined that take a vari-
able 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 sup-
pressed). You could not expect a C program to detect similar errors when you
pass data to it from a Pascal program.

Languages Programmer’s Guide

3–3
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:

<table>
<thead>
<tr>
<th>Pascal</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>program p(input,output);</td>
<td>main() {</td>
</tr>
<tr>
<td>begin</td>
<td>printf(&quot;hi\n!&quot;);</td>
</tr>
<tr>
<td>writeln(&quot;hi!&quot;);</td>
<td>}</td>
</tr>
<tr>
<td>end.</td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>If Pascal function returns:</th>
<th>Declare C function as:</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>int</td>
</tr>
<tr>
<td>cardinal</td>
<td>unsigned int</td>
</tr>
<tr>
<td>char</td>
<td>char</td>
</tr>
<tr>
<td>boolean</td>
<td>char</td>
</tr>
<tr>
<td>enumeration</td>
<td>unsigned, or corresponding enum</td>
</tr>
<tr>
<td>real</td>
<td>enum (C's enum are signed)</td>
</tr>
<tr>
<td>double</td>
<td>none</td>
</tr>
<tr>
<td>pointer type</td>
<td>double</td>
</tr>
<tr>
<td>record type</td>
<td>corresponding pointer type</td>
</tr>
<tr>
<td>array type</td>
<td>union type</td>
</tr>
<tr>
<td></td>
<td>corresponding array type</td>
</tr>
</tbody>
</table>

1 Applies also to subrange with lower bound <0.
2 Applies also to subrange with lower bound >=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.
<table>
<thead>
<tr>
<th>If Pascal expects:</th>
<th>C argument should be:</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>integer or char value $-2^{31}..2^{31}-1$</td>
</tr>
<tr>
<td>cardinal</td>
<td>integer or char value $0..2^{32}-1$</td>
</tr>
<tr>
<td>subrange</td>
<td>integer or char value in subrange</td>
</tr>
<tr>
<td>char</td>
<td>integer or char (0..255)</td>
</tr>
<tr>
<td>boolean</td>
<td>integer or char (0 or 1 only)</td>
</tr>
<tr>
<td>enumeration</td>
<td>integer or char (0..N-1)</td>
</tr>
<tr>
<td>real</td>
<td>none</td>
</tr>
<tr>
<td>double</td>
<td>float or double</td>
</tr>
<tr>
<td>procedure</td>
<td>struct {void *p(); int *l}</td>
</tr>
<tr>
<td>function</td>
<td>struct {function-&gt;type *f(); int *l}</td>
</tr>
<tr>
<td>pointer types&lt;sup&gt;1&lt;/sup&gt;</td>
<td>pointer type</td>
</tr>
<tr>
<td>reference parameter</td>
<td>und &lt;0. := lbound(s)</td>
</tr>
<tr>
<td>record types</td>
<td>pointer to the appropriate type</td>
</tr>
<tr>
<td>by–reference array parameters</td>
<td>structure or union type</td>
</tr>
<tr>
<td>by–reference text</td>
<td>corresponding array type</td>
</tr>
<tr>
<td>by–value array parameters</td>
<td>FILE&lt;sup&gt;**&lt;/sup&gt;</td>
</tr>
<tr>
<td></td>
<td>structure containing the corresponding array</td>
</tr>
</tbody>
</table>

<sup>1</sup>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.

```c
/* Send the last command-line argument to the Pascal routine */
#include <stdio.h>
main(argc, argv)
   int argc; char **argv;
{ 
   FILE *temp = stdout;
   if (argc != 0)
      p_routine(&temp, argv[argc - 1]);
}
```

```pascal
{ We assume the string passed to us by the C program will not exceed 100 bytes in length. type
   astring = packed array [1 .. 100] of char;
procedure p_routine(var f: text; var c: astring);
var
   i: integer;
begin
   i := lbound(c);
   while (i < hbond(c)) and (c[i] <> chr(0)) do
      begin
         write(f, c[i])
         i := i + 1;
      end;
writeln(f);
end;
```

![Figure 3.3. Passing Strings to a Pascal Procedure from C.](image)

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.

<table>
<thead>
<tr>
<th>If C expects:</th>
<th>Pascal parameter should be:</th>
</tr>
</thead>
<tbody>
<tr>
<td>int ¹</td>
<td>integer</td>
</tr>
<tr>
<td>unsigned int²</td>
<td>cardinal</td>
</tr>
<tr>
<td>short ³</td>
<td>integer (or –32768..32767)</td>
</tr>
<tr>
<td>unsigned short</td>
<td>cardinal (or 0..65535)</td>
</tr>
<tr>
<td>char ⁴</td>
<td>char</td>
</tr>
<tr>
<td>signed char</td>
<td>integer (or –128..127)</td>
</tr>
<tr>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>enum type</td>
<td>corresponding enumeration type</td>
</tr>
<tr>
<td>string (char *)</td>
<td>packed character array passed by reference (VAR)</td>
</tr>
<tr>
<td>pointer to function</td>
<td>none</td>
</tr>
<tr>
<td>FILE *</td>
<td>none</td>
</tr>
<tr>
<td>FILE **</td>
<td>text, passed by reference (VAR)</td>
</tr>
<tr>
<td>pointer type</td>
<td>corresponding pointer type</td>
</tr>
<tr>
<td></td>
<td>or corresponding type passed by reference (VAR)</td>
</tr>
<tr>
<td>struct type</td>
<td>corresponding record type</td>
</tr>
<tr>
<td>union type</td>
<td>corresponding record type</td>
</tr>
<tr>
<td>array type</td>
<td>corresponding array type passed by reference (VAR)</td>
</tr>
</tbody>
</table>

¹Same as types signed int, long, signed long, signed
²Same as types unsigned, unsigned long
³Same as type signed short
⁴Same 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.
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 created 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.
Figures 4.2 through 4.6 show listings from raw data produced by pixie. The prof option used is given at the top of each figure.
Figure 4.2. Profiler Listing for Procedure Invocations.
### Procedures — Basic Block Counts

**Profiler option:--pixied--procedures**

Table showing cycle counts:

<table>
<thead>
<tr>
<th>File</th>
<th>#cycles</th>
<th>%cycles</th>
<th>cum %</th>
<th>bytes</th>
<th>procedure (file)</th>
</tr>
</thead>
<tbody>
<tr>
<td>textoutput.c</td>
<td>48071708</td>
<td>32.45</td>
<td>32.45</td>
<td>34</td>
<td>write_char (/..textoutput.c)</td>
</tr>
<tr>
<td>textinput.c</td>
<td>26452936</td>
<td>19.86</td>
<td>52.31</td>
<td>30</td>
<td>eoln (/..textoutput.c)</td>
</tr>
<tr>
<td>textinput.c</td>
<td>20662526</td>
<td>13.95</td>
<td>66.26</td>
<td>23</td>
<td>read_char (/..textoutput.c)</td>
</tr>
<tr>
<td>textinput.c</td>
<td>4307332</td>
<td>2.91</td>
<td>69.17</td>
<td>62</td>
<td>write_char (/..textoutput.c)</td>
</tr>
<tr>
<td>textoutput.c</td>
<td>3678408</td>
<td>2.48</td>
<td>71.65</td>
<td>133</td>
<td>write_chars (/..textoutput.c)</td>
</tr>
<tr>
<td>textoutput.c</td>
<td>1573858</td>
<td>1.06</td>
<td>72.71</td>
<td>28</td>
<td>write_string (/..textoutput.c)</td>
</tr>
<tr>
<td>textinput.c</td>
<td>362700</td>
<td></td>
<td></td>
<td></td>
<td>writeln (/..textoutput.c)</td>
</tr>
<tr>
<td>textinput.c</td>
<td>279500</td>
<td></td>
<td></td>
<td></td>
<td>rof (/..textoutput.c)</td>
</tr>
<tr>
<td>textinput.c</td>
<td>250152</td>
<td></td>
<td></td>
<td></td>
<td>flsbuf (/..flsbuf.c)</td>
</tr>
<tr>
<td></td>
<td>30783</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1781</td>
<td>0.01</td>
<td>100.00</td>
<td>60</td>
<td>refill (/..refill.c)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.3. Profiler Listing for Procedures Based on Basic Blocks Counts.**

- **Cumulative total, i.e., 92.91% of all cycles used by write_char, main, eoln, and read_char.**
- **Cycles used by write_char in units and as a percentage of total program cycles.**
- **Statistics describe calls to write_char compiled from source file textoutput.c.**
- **fopen used an average of 82 cycles per call and 13 bytes per line.**
Procedures — Basic Block Counts (with clock time)
Profiler option:—pixie—procedure —clock

148137751 cycles (18.5172 seconds at 8.00 megahertz)

<table>
<thead>
<tr>
<th>cycles</th>
<th>%cycles</th>
<th>cum %</th>
<th>seconds</th>
<th>cycles</th>
<th>bytes</th>
<th>procedure (file)</th>
</tr>
</thead>
<tbody>
<tr>
<td>48071708</td>
<td>32.45</td>
<td>32.45</td>
<td>6.0090</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42443503</td>
<td>28.65</td>
<td>61.11</td>
<td>5.3054</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26457936</td>
<td>17.86</td>
<td>78.96</td>
<td>3.3072</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20662326</td>
<td>13.95</td>
<td>92.81</td>
<td>2.5828</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4307932</td>
<td>2.91</td>
<td>95.82</td>
<td>0.5385</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3678408</td>
<td>2.48</td>
<td>98.30</td>
<td>0.4598</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1573858</td>
<td>1.06</td>
<td>99.36</td>
<td>0.1967</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>362700</td>
<td>0.24</td>
<td>99.61</td>
<td>0.0453</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>279002</td>
<td>0.19</td>
<td>99.80</td>
<td>0.0349</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>251152</td>
<td>0.17</td>
<td>99.97</td>
<td>0.0314</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30283</td>
<td>0.02</td>
<td>99.99</td>
<td>0.0038</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13391</td>
<td>0.01</td>
<td>100.00</td>
<td>0.0017</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2923</td>
<td>0.00</td>
<td>100.00</td>
<td>0.0004</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1356</td>
<td>0.00</td>
<td>100.00</td>
<td>0.0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>735</td>
<td>0.00</td>
<td>100.00</td>
<td>0.0001</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>0.00</td>
<td>100.00</td>
<td>0.0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>0.00</td>
<td>100.00</td>
<td>0.0000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>0.00</td>
<td>100.00</td>
<td>0.0000</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The profiler computes the time in seconds based on the megahertz machine speed specified in the —clock option.

This listing contains the same information as the listing shown in Figure 4.3, except that this listing also contains the number of seconds spent in each procedure.

6 0.00 100.00 0.0000 4
5 0.00 100.00 0.0000 5
5 0.00 100.00 0.0000 5

Figure 4.4. Profler Listing for Procedures Based on Basic Blocks Counts (with clock times).
## Heavy — Basic Block Counts

**Profiler option**: —pixie — heavy

* -h[heavy] using basic-block counts;
* sorted in descending order by the number of cycles executed in each *
* line; unexecuted lines are excluded

<table>
<thead>
<tr>
<th>procedure (file)</th>
<th>line</th>
<th>bytes</th>
<th>cycles</th>
<th>%</th>
<th>cum %</th>
</tr>
</thead>
<tbody>
<tr>
<td>write_char (...)</td>
<td>120</td>
<td>88</td>
<td>28276478</td>
<td>19.09</td>
<td>19.09</td>
</tr>
<tr>
<td>eoln (...)</td>
<td>31</td>
<td>116</td>
<td>22808688</td>
<td>15.40</td>
<td>34.49</td>
</tr>
<tr>
<td>main (fixfont.p)</td>
<td>42</td>
<td>92</td>
<td>19069136</td>
<td>12.87</td>
<td>47.36</td>
</tr>
<tr>
<td>read_char (...)</td>
<td>59</td>
<td>56</td>
<td>9881982</td>
<td>6.67</td>
<td>54.06</td>
</tr>
<tr>
<td>main (fixfont.p)</td>
<td>43</td>
<td>40</td>
<td>4583512</td>
<td>3.48</td>
<td>57.54</td>
</tr>
<tr>
<td>write_char (...)</td>
<td>105</td>
<td>20</td>
<td>7024225</td>
<td>4.77</td>
<td>62.31</td>
</tr>
<tr>
<td>eoln (...)</td>
<td>60</td>
<td>28</td>
<td>5390172</td>
<td>3.64</td>
<td>65.95</td>
</tr>
<tr>
<td>main (fixfont.p)</td>
<td>37</td>
<td>20</td>
<td>449868</td>
<td>0.31</td>
<td>66.26</td>
</tr>
<tr>
<td>write_char (...)</td>
<td>115</td>
<td>12</td>
<td>514465</td>
<td>0.34</td>
<td>66.60</td>
</tr>
<tr>
<td>write_char (...)</td>
<td>34</td>
<td>40</td>
<td>34344</td>
<td>0.23</td>
<td>66.83</td>
</tr>
<tr>
<td>eoln (...)</td>
<td>61</td>
<td>16</td>
<td>2736936</td>
<td>1.86</td>
<td>88.48</td>
</tr>
<tr>
<td>read_char (...)</td>
<td>89</td>
<td>8</td>
<td>1795072</td>
<td>1.21</td>
<td>89.69</td>
</tr>
<tr>
<td>main (fixfont.p)</td>
<td>36</td>
<td>8</td>
<td>200582</td>
<td>0.13</td>
<td>90.82</td>
</tr>
<tr>
<td>write_chars (...)</td>
<td>47</td>
<td>12</td>
<td>170355</td>
<td>1.16</td>
<td>88.05</td>
</tr>
<tr>
<td>write_char (...)</td>
<td>106</td>
<td>4</td>
<td>141395</td>
<td>0.95</td>
<td>89.01</td>
</tr>
<tr>
<td>write_char (...)</td>
<td>112</td>
<td>16</td>
<td>141395</td>
<td>0.95</td>
<td>90.36</td>
</tr>
<tr>
<td>write_integer (...)</td>
<td>197</td>
<td>32</td>
<td>136429</td>
<td>0.92</td>
<td>91.28</td>
</tr>
<tr>
<td>main (fixfont.p)</td>
<td>45</td>
<td>9</td>
<td>17000</td>
<td>0.12</td>
<td>91.40</td>
</tr>
<tr>
<td>write_integer (...)</td>
<td>198</td>
<td>20</td>
<td>17000</td>
<td>0.12</td>
<td>91.52</td>
</tr>
<tr>
<td>main (fixfont.p)</td>
<td>39</td>
<td>19</td>
<td>141395</td>
<td>0.95</td>
<td>92.47</td>
</tr>
<tr>
<td>eoln (...)</td>
<td>28</td>
<td>18</td>
<td>141395</td>
<td>0.95</td>
<td>93.42</td>
</tr>
<tr>
<td>main (fixfont.p)</td>
<td>36</td>
<td>20</td>
<td>141395</td>
<td>0.95</td>
<td>93.77</td>
</tr>
<tr>
<td>write_string (...)</td>
<td>150</td>
<td>44</td>
<td>571050</td>
<td>0.39</td>
<td>94.16</td>
</tr>
<tr>
<td>write_chars (...)</td>
<td>48</td>
<td>4</td>
<td>567855</td>
<td>0.38</td>
<td>94.54</td>
</tr>
<tr>
<td>write_chars (...)</td>
<td>47</td>
<td>4</td>
<td>567855</td>
<td>0.38</td>
<td>94.92</td>
</tr>
<tr>
<td>write_chars (...)</td>
<td>49</td>
<td>28</td>
<td>487387</td>
<td>0.33</td>
<td>95.25</td>
</tr>
<tr>
<td>write_chars (...)</td>
<td>18</td>
<td>20</td>
<td>348150</td>
<td>0.24</td>
<td>95.49</td>
</tr>
<tr>
<td>main (fixfont.p)</td>
<td>31</td>
<td>100</td>
<td>348000</td>
<td>0.23</td>
<td>95.72</td>
</tr>
</tbody>
</table>

The most heavily used line is line 120, located in procedure write_char, compiled from source file textoutput.c.

Lines 120, 31, and 42 executed 47.36% of total program cycles.

Line 36 of textoutput.c has 8 bytes of code, used 1,795,872 cycles, or 1.21% of total program cycles.

Figure 4.5. Profiler Listing for Heavy Line Usage.
### Lines — Basic Block Counts

**Profiler option:** —pixie —lines

<table>
<thead>
<tr>
<th>procedure (file)</th>
<th>line</th>
<th>bytes</th>
<th>cycles</th>
<th>%cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>write_char (./textoutput.c)</td>
<td>105</td>
<td>20</td>
<td>7069725</td>
<td>4.77</td>
</tr>
<tr>
<td></td>
<td>106</td>
<td>8</td>
<td>2827890</td>
<td>1.91</td>
</tr>
<tr>
<td></td>
<td>111</td>
<td>8</td>
<td>2827930</td>
<td>1.91</td>
</tr>
<tr>
<td></td>
<td>106</td>
<td>4</td>
<td>3253945</td>
<td>0.95</td>
</tr>
<tr>
<td></td>
<td>112</td>
<td>16</td>
<td>413945</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>113</td>
<td>72</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>115</td>
<td>17</td>
<td>4241835</td>
<td>2.86</td>
</tr>
<tr>
<td></td>
<td>116</td>
<td>28</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>117</td>
<td>28</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>110</td>
<td>88</td>
<td>28276478</td>
<td>19.09</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>60</td>
<td>15</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>32</td>
<td>9</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>24</td>
<td>6</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>24</td>
<td>6</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>4</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>40</td>
<td>8490</td>
<td>0.01</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>18</td>
<td>166</td>
<td>0.00</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>48</td>
<td>12</td>
<td>0.00</td>
</tr>
<tr>
<td>eoln (./textinput.c)</td>
<td>27</td>
<td>12</td>
<td>2736936</td>
<td>1.85</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>4</td>
<td>912312</td>
<td>0.62</td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>116</td>
<td>2208688</td>
<td>15.40</td>
</tr>
<tr>
<td></td>
<td>58</td>
<td>8</td>
<td>1796724</td>
<td>1.21</td>
</tr>
<tr>
<td></td>
<td>59</td>
<td>56</td>
<td>9881982</td>
<td>6.67</td>
</tr>
<tr>
<td></td>
<td>60</td>
<td>28</td>
<td>5390172</td>
<td>3.64</td>
</tr>
<tr>
<td></td>
<td>61</td>
<td>16</td>
<td>3593448</td>
<td>2.43</td>
</tr>
<tr>
<td>read_char (./textinput.c)</td>
<td>18</td>
<td>20</td>
<td>348150</td>
<td>0.24</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>8</td>
<td>139260</td>
<td>0.09</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>12</td>
<td>208890</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>4</td>
<td>139</td>
<td>0.00</td>
</tr>
</tbody>
</table>

The statistics to the right describe lines of code in procedure write_char compiled from the source file textoutput.c.

- **Line 105 in write_char contains 20 bytes of code; it executed 7,069,725 times using 4.77% of the total program cycles.**

- **Line 117 in write_char contains 28 bytes of code; no cycles were recorded for execution.**

---

**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*) 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:

```plaintext
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* and *myprog.Counts*, and prints it in an easily readable format. For example:

```plaintext
prof --pixie myprog myprog.Addr myprog.Counts
```

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

```plaintext
program_name.Addr 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).
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 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 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.

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` and `pname.Counts` and pro-
  vides 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-p[rocedures]</code></td>
<td>Lists the time spent in each procedure.</td>
</tr>
<tr>
<td></td>
<td>See Figures 4.3 for a sample output listing.</td>
</tr>
<tr>
<td><code>-pixie</code></td>
<td><code>Basic block counting.</code> Indicates that information is to be generated on basic block counting, and that the <code>Addr</code> and <code>Counts</code> file produced by <code>pixie</code> are to be used by default.</td>
</tr>
<tr>
<td></td>
<td>See Figure 2.3 through 2.6 for examples of sample output.</td>
</tr>
<tr>
<td><code>-i[nvocations]</code></td>
<td><code>Basic block counting.</code> 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.</td>
</tr>
<tr>
<td></td>
<td>See Figure 4.2 for sample output.</td>
</tr>
<tr>
<td><code>-l[ines]</code></td>
<td><code>Basic block counting.</code> List statistics for each line of source code.</td>
</tr>
<tr>
<td></td>
<td>See Figure 4.6 for sample output.</td>
</tr>
<tr>
<td><code>-o[nly] proc_name</code></td>
<td>Reports information on only the procedure specified by <code>procedure_name</code>, rather than on the entire program. You may specify more than one <code>--o</code> option. If you specify uppercase <code>-O</code>, <code>prof</code> uses only the name procedure(s), rather than the entire program, as the base upon which it calculates percentages.</td>
</tr>
<tr>
<td></td>
<td>Excludes information on the procedure(s) (and their descendants) specified by <code>procedure_name</code>. If you specify uppercase <code>-E</code> for Exclude, <code>prof</code> also omits that</td>
</tr>
<tr>
<td><code>-e[xclude] procedure_name</code></td>
<td><code>Basic block counting.</code> Prints a list of procedures that are never invoked.</td>
</tr>
<tr>
<td><code>-z[ero]</code></td>
<td><code>Basic block counting.</code> Prints a list of procedures that are never invoked.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-q[n]</td>
<td>Allows you to condense output listings by truncating unwanted lines. You can truncate by specify n in three different ways:</td>
</tr>
<tr>
<td>n</td>
<td>n is an integer. All Lines after n line are truncated.</td>
</tr>
<tr>
<td>n%</td>
<td>n is an integer followed by the percentage sign. All lines after the line containing n% calls in the %calls column are truncated.</td>
</tr>
<tr>
<td>n cum%</td>
<td>n is an integer followed by the characters <em>cum</em> (for cumulative) and a percentage sign. All lines after the line containing ncum% calls in the cum% column are truncated.</td>
</tr>
</tbody>
</table>

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.

```
-prof -q 4
-prof -q 13%
-prof -q 92cum%
```

<table>
<thead>
<tr>
<th>calls</th>
<th>%calls</th>
<th>cum%</th>
</tr>
</thead>
<tbody>
<tr>
<td>48071708</td>
<td>32.45</td>
<td>6.0090</td>
</tr>
<tr>
<td>42443503</td>
<td>28.65</td>
<td>5.3054</td>
</tr>
<tr>
<td>26457936</td>
<td>17.86</td>
<td>3.3072</td>
</tr>
<tr>
<td>20662326</td>
<td>13.95</td>
<td>2.5828</td>
</tr>
<tr>
<td>6307432</td>
<td>2.01</td>
<td>0.3385</td>
</tr>
<tr>
<td>3678406</td>
<td>2.43</td>
<td>0.4598</td>
</tr>
<tr>
<td>3773618</td>
<td>1.06</td>
<td>0.1947</td>
</tr>
<tr>
<td>582700</td>
<td>0.24</td>
<td>0.0452</td>
</tr>
<tr>
<td>2739002</td>
<td>0.19</td>
<td>0.0399</td>
</tr>
<tr>
<td>2511152</td>
<td>0.17</td>
<td>0.0314</td>
</tr>
<tr>
<td>202952</td>
<td>0.02</td>
<td>0.0038</td>
</tr>
<tr>
<td>13390</td>
<td>0.01</td>
<td>0.0017</td>
</tr>
<tr>
<td>2223</td>
<td>0.00</td>
<td>0.0004</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-h[eavy]</code></td>
<td>Basic block counting. Same as the <code>--lines</code> option, but sorts the lines by their frequency of use. See Figure 4.5 for a sample output listing.</td>
</tr>
<tr>
<td><code>-c[lock]</code></td>
<td>Basic block counting. Lists the number of seconds spent in each routine, based on the CPU clock frequency n, expressed in megahertz; n defaults to 8.0 of omitted. Never use the default if the next argument <code>program_name</code> or <code>profile_name</code> begins with a digit. See Figure 4.4 for a sample output listing.</td>
</tr>
<tr>
<td><code>-t[estcoverage]</code></td>
<td>Basic block counting. Lists line numbers that contain code that is never executed.</td>
</tr>
<tr>
<td><code>-m[erge]filename</code></td>
<td>This option is useful when multiple input files of profile data (normally in <code>mon.out</code>) 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.</td>
</tr>
<tr>
<td><code>-f[eedback] filename</code></td>
<td>Produces a file used by the driver <code>--cord</code> option to maximize cache efficiency. See Reducing Cache Conflicts in this chapter for details.</td>
</tr>
</tbody>
</table>

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. RISCComputers 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:

```c
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.

```assembly
# 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.

```assembly
# 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 nexta[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-\textit{filename}ones.

The \textit{upload} and \textit{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 \textit{upload} and \textit{umerge} phases of the compiler system overcomes this deficiency. The \textit{upload} phase links multi-compilation units into a single compilation unit. Then, \textit{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 –\textit{On} option determines the execution sequence. The table belows summarizes the functions of each of the –\textit{O} options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>–O3</td>
<td>The \textit{ulink} and \textit{umerge} 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. The \textit{ulink} phase combines all the ucode files and symbol tables, and passes control to \textit{umerge}. \textit{Umerge} reorders the ucode for optimal processing by uopt. Upon completion, \textit{umerge} passes control to uopt, which performs global optimizations on the program.</td>
</tr>
<tr>
<td>–O2</td>
<td>\textit{Ulink} and \textit{umerge} are bypassed, and only the global optimizer (uopt) phase executes. It performs optimization only within the bounds of individual compilation units.</td>
</tr>
<tr>
<td>–O1</td>
<td>\textit{Ulink}, \textit{umerge}, and uopt are bypassed. However, the code generator and the assembler perform basic optimizations in a more limited scope.</td>
</tr>
<tr>
<td>–O0</td>
<td>\textit{Ulink}, \textit{umerge}, and uopt are bypassed, and the assembler bypasses certain optimizations it normally performs.</td>
</tr>
</tbody>
</table>

**NOTE:** The –O3 options is not available for the \textit{cobol} driver when the edition of this manual was printed. You should refer to the \textit{cc(1)} \textit{f77(1)} \textit{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 a.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 following statement to perform optimization and complete the compilation process.

```bash
$ 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 \texttt{--Olimit} option at compilation.

Because compilation time increases by the square of the program size, the RIS/\texttt{j} 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**.

   ```
   b.c
   foo()
   {
   ...
   }
   bar()
   {
   ...
   }
   zot()
   {
   ...
   }
   struct
   {
   ...
   }
   work;
   
   c.c
   x()
   {
   ...
   }
   help()
   {
   ...
   }
   struct
   {
   ...
   }
   ddata;
   y()
   {
   ...
   }
   ```

   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).

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

   \% cc \( -c -03 -kp \) 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.

![Diagram](image)

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

   \% cc \( -j\) future.c
   \% cc \( -03\) 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.
<table>
<thead>
<tr>
<th>Option Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-S</td>
<td>Similar to -c, except produces assembly code in a .s file instead of object code in a .o file.</td>
</tr>
<tr>
<td>-std</td>
<td>Issues a warning message when the compiler finds a non-standard feature in the programming language of your source program.</td>
</tr>
<tr>
<td>-U name</td>
<td>Overrides a definition of a macro name that you specified with the -D option, or that is defined automatically by the driver.</td>
</tr>
<tr>
<td>-v</td>
<td>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.</td>
</tr>
<tr>
<td>-V</td>
<td>Prints the version number of the driver and its phases. When reporting a suspected compiler problem, you must include this number.</td>
</tr>
<tr>
<td>-w</td>
<td>Suppresses warning messages.</td>
</tr>
<tr>
<td>-p1 or -p</td>
<td>Permits program counter (pc) sampling. This option provides operational statistics for use in improving program performance. See Chapter 4 for more details. <strong>Note:</strong> 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.</td>
</tr>
</tbody>
</table>
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:

```c
int len = 10;
char a[10];

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

Generated Assembly Code:

```assembly
# 8 for (p = a; p != a + len; ) *p++ = 0;      # p = a
move $2, $4
lw $3, len
addu $24, $4, 33          # a + len != a
beq $24, $4, 33
$32:
sb $0, 0($2)               # *p = 0
addu $2, $2, 1             # p++
lw $25, len
addu $8, $4, $25
bne $8, $2, 32
$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 unchang-
ing values.

**Using subscripts instead of pointers.** The use of subscripting in the procedure
`zero` 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.

<table>
<thead>
<tr>
<th>Source Code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>void azero()</td>
</tr>
<tr>
<td>int i;</td>
</tr>
<tr>
<td>for (i = 0; i != len; i++) a[i] = 0;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generated Assembly Code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i = 0; i != len; i++) a[i] = 0;</td>
</tr>
<tr>
<td>move $2, $0            # i = 0</td>
</tr>
<tr>
<td>beq $4, 0, $37        # len != 0</td>
</tr>
<tr>
<td>la $5, a</td>
</tr>
<tr>
<td>$36:</td>
</tr>
<tr>
<td>sb $0, 0($5)          # *a = 0</td>
</tr>
<tr>
<td>addu $2, $2, 1       # i++</td>
</tr>
<tr>
<td>addu $5, $5, 1       # a++</td>
</tr>
<tr>
<td>beq $2, $4, $36 # i != len</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Source Code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>char a[10];</td>
</tr>
<tr>
<td>void lzero(len);</td>
</tr>
<tr>
<td>int len;</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>char *p;</td>
</tr>
<tr>
<td>for (p = a; p != a + len; ) *p++ = 0;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Generated Assembly Code:</th>
</tr>
</thead>
<tbody>
<tr>
<td># for (p = a; p != a + len; ) *p++ = 0;</td>
</tr>
<tr>
<td>move $2, $6            # p = a</td>
</tr>
<tr>
<td>addu $5, $6, $4</td>
</tr>
<tr>
<td>beq $5, $6, $33       # a + len != a</td>
</tr>
<tr>
<td>$32:</td>
</tr>
<tr>
<td>sb $0, 0($2)          # *p = 0</td>
</tr>
<tr>
<td>addu $2, $2, 1       # p++</td>
</tr>
<tr>
<td>bne $5, $2, $32       # a + len != p</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Bad</th>
<th>Good</th>
</tr>
</thead>
<tbody>
<tr>
<td>`while (n--){}</td>
<td>`while (n != 0) {</td>
</tr>
<tr>
<td>...</td>
<td>n--;</td>
</tr>
<tr>
<td>}</td>
<td>...</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>OK</th>
<th>More Efficient</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if ( i == 1) c = &quot;1&quot;;</code></td>
<td><code>c = &quot;01&quot;[i];</code></td>
</tr>
<tr>
<td><code>else c = &quot;0&quot;;</code></td>
<td></td>
</tr>
</tbody>
</table>

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:

```c
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 03fe0000 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`.

4-30 Languages Programmer's Guide
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 `.data` 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 -nount libieee.a
    -count liblaser.a

In the above example, the link editor assumes that libieee.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

RISC Computer hardware provides two high-speed caches—one for program data
and the other for instructions—that temporarily hold data or instructions fre-
quently used by the processor. During execution, instruction or data from speci-
fied memory locations are placed in the cache. Because the cache is much
smaller than memory, a single cache location is shared by many distinct mem-
ory 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 re-
peatedly, 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 re-
turn, 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 us-
ing 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)`, `p11(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 `--jimpopt` enables the compiler to fill additional delay slots at the cost of requiring more memory by the link editor. The default is `nojimpopt`. 
For programs requiring the ultimate in performance, you should specify the
—jmport. 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 println(pline = 0x7ffff5b80)["sam.c":58,0x2f7]

   println 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 = 0x7ffebea0)["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.

3. Do a stack trace using the where command.

4. 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 playback 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:**

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbx [options] [objfile corefile]</code></td>
<td>Invoke DBX from the shell command line.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Select this option to...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-I dirname</code></td>
<td>Tell DBX to look in the specified directory for source files. To specify multiple directories, you must use a separate <code>-I</code> 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.</td>
</tr>
<tr>
<td><code>-c filename</code></td>
<td>Select a command file other than your <code>dbxinit</code> file.</td>
</tr>
<tr>
<td><code>-i</code></td>
<td>Use interactive mode. This option does not treat <code>#</code>s 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.</td>
</tr>
<tr>
<td><code>-r</code></td>
<td>Run your program immediately upon entering DBX.</td>
</tr>
<tr>
<td><code>-k</code></td>
<td>Turn on kernel debugging.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>quit</td>
<td>End your debugging session.</td>
</tr>
<tr>
<td>q</td>
<td></td>
</tr>
</tbody>
</table>

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 fget\(s(1)\) in the \textit{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 \texttt{stop in} command illustrates the syntax conventions just described:

\begin{verbatim}
  stop VAR in PROCEDURE if EXP
\end{verbatim}

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

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

\textit{Table 5.1 (1 of 2). Keywords Used in Command Syntax Descriptions.}
<table>
<thead>
<tr>
<th>Syntax</th>
<th>You specify:</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING</td>
<td>Any ASCII string.</td>
</tr>
<tr>
<td>DIR</td>
<td>A directory name.</td>
</tr>
<tr>
<td>ARGS</td>
<td>Program arguments (maximum allowed by DBX is 1000; however, system limits may also apply).</td>
</tr>
<tr>
<td>VAR</td>
<td>Valid program variable or DBX predefined variable (see Table 5.2). For machine-level debugging, VAR can also be an address.</td>
</tr>
<tr>
<td></td>
<td>You must qualify program variables with duplicate names as described in the section <strong>Qualifying Variable Names</strong>.</td>
</tr>
<tr>
<td>NAME</td>
<td>DBX command name.</td>
</tr>
<tr>
<td>COMMAND_LIST</td>
<td>One or more commands, each separated by semicolons.</td>
</tr>
</tbody>
</table>

*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:**

```plaintext
sam.main.i
current file
procedure
________variable
```

**DBX Expressions and Precedence**

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>(&quot;FILE&quot; #EXP)</td>
<td>Uses the specified line number (#EXP) in that file.</td>
</tr>
<tr>
<td></td>
<td>(PROCEDURE #EXP)</td>
<td>Uses the specified line number (#EXP) in that procedure.</td>
</tr>
<tr>
<td></td>
<td>(#EXP)</td>
<td>Takes line number (#EXP) and returns the address for that line.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$address</td>
<td>pointer</td>
</tr>
<tr>
<td>$unsigned</td>
<td>unsigned integer</td>
</tr>
<tr>
<td>$char</td>
<td>character</td>
</tr>
<tr>
<td>$boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>$real</td>
<td>double precision real</td>
</tr>
<tr>
<td>$integer</td>
<td>signed integer</td>
</tr>
<tr>
<td>$float</td>
<td>single precision real</td>
</tr>
<tr>
<td>$double</td>
<td>double precision real</td>
</tr>
<tr>
<td>$uchar</td>
<td>unsigned character</td>
</tr>
<tr>
<td>$short</td>
<td>16–bit integer</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>0</td>
</tr>
<tr>
<td>true</td>
<td>nonzero</td>
</tr>
<tr>
<td>nil</td>
<td>0</td>
</tr>
<tr>
<td>0x number</td>
<td>hexadecimal</td>
</tr>
<tr>
<td>0&lt;number</td>
<td>decimal</td>
</tr>
<tr>
<td>0&lt;number</td>
<td>octal</td>
</tr>
<tr>
<td>number</td>
<td>decimal</td>
</tr>
<tr>
<td>number..[number] [eE][+-EXP]</td>
<td>float</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>default</td>
<td>decimal</td>
</tr>
</tbody>
</table>
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.

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

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:**

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

**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:

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

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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMMAND; COMMAND</td>
<td>Type multiple commands on the same line.</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING^Z</td>
<td>Complete a symbol name or to see what symbol names contain the specified string.</td>
</tr>
</tbody>
</table>

Example:

```
(dbx) i^z
ioctl.ioctl .ioctl 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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>Display a list of DBX predefined variables.</td>
</tr>
<tr>
<td>set VAR = EXP</td>
<td>Assign a new value to a variable or to define a new variable.</td>
</tr>
</tbody>
</table>
Example:

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

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>unset VAR</td>
<td>Unset the value of a DBX variable.</td>
</tr>
</tbody>
</table>
Example:

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

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.

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>integer</td>
</tr>
<tr>
<td>B</td>
<td>boolean</td>
</tr>
<tr>
<td>S</td>
<td>ASCII character string</td>
</tr>
<tr>
<td>R</td>
<td>reset exclusively and periodically by the debugger</td>
</tr>
<tr>
<td>Key</td>
<td>Variable</td>
</tr>
<tr>
<td>-----</td>
<td>------------</td>
</tr>
<tr>
<td>S</td>
<td>$addrfmt</td>
</tr>
<tr>
<td>B</td>
<td>$casesense</td>
</tr>
<tr>
<td>I R</td>
<td>$curevent</td>
</tr>
<tr>
<td>I R</td>
<td>$curline</td>
</tr>
<tr>
<td>I R</td>
<td>$cursrcline</td>
</tr>
<tr>
<td></td>
<td>$curpc</td>
</tr>
<tr>
<td>B</td>
<td>$datacache</td>
</tr>
<tr>
<td></td>
<td>$debugflag</td>
</tr>
<tr>
<td>S R</td>
<td>$defaultout</td>
</tr>
<tr>
<td>S R</td>
<td>$defaultin</td>
</tr>
<tr>
<td></td>
<td>$defin</td>
</tr>
<tr>
<td></td>
<td>$defout</td>
</tr>
<tr>
<td></td>
<td>$dispix</td>
</tr>
</tbody>
</table>

*Table 5.2 (1 of 3). Predefined DBX Variables.*
<table>
<thead>
<tr>
<th>Key</th>
<th>Variable</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>$\text{hexchars}$</td>
<td>0</td>
<td>Displays hexadecimal when set to a non-zero value. Hexadecimal overrides octal.</td>
</tr>
<tr>
<td>B</td>
<td>$\text{hexin}$</td>
<td>0</td>
<td>Changes the default input constants to hexadecimal when set to a non-zero value.</td>
</tr>
<tr>
<td>B</td>
<td>$\text{hexints}$</td>
<td>0</td>
<td>Changes the default output constants to hexadecimal when set to a non-zero value. Hexadecimal overrides octal.</td>
</tr>
<tr>
<td>B</td>
<td>$\text{hexstrings}$</td>
<td>0</td>
<td>Specifies whether you want to see all strings printed in hexadecimal. A 1 means use hexadecimal; a 0 means use characters.</td>
</tr>
<tr>
<td>I R</td>
<td>$\text{historyevent}$</td>
<td>none</td>
<td>Shows the current history number.</td>
</tr>
<tr>
<td>I</td>
<td>$\text{lines}$</td>
<td>20</td>
<td>Specifies the size of your DBX history list.</td>
</tr>
<tr>
<td>I</td>
<td>$\text{listwindow}$</td>
<td>10</td>
<td>Specifies how many lines the list command lists.</td>
</tr>
<tr>
<td>S</td>
<td>$\text{main}$</td>
<td>&quot;main&quot;</td>
<td>Specifies the name of the procedure where execution is to begin. DBX by default starts at the procedure $\text{main}$ unless you specify otherwise.</td>
</tr>
<tr>
<td>I</td>
<td>$\text{maxstrlen}$</td>
<td>128</td>
<td>Specifies how many characters of a string that DBX will print for pointers to strings.</td>
</tr>
<tr>
<td>B</td>
<td>$\text{octints}$</td>
<td>0</td>
<td>Changes the default output constants to octal when set to a non-zero value. Hexadecimal overrides octal.</td>
</tr>
<tr>
<td>B</td>
<td>$\text{octin}$</td>
<td>0</td>
<td>Changes the default input constants to octal when set to a non-zero value. Hexadecimal overrides octal.</td>
</tr>
<tr>
<td>B</td>
<td>$\text{page}$</td>
<td>1</td>
<td>Specifies whether to page long information. A nonzero value turns on paging; a 0 turns it off.</td>
</tr>
</tbody>
</table>

Table 5.2 (2 of 3). Predefined DBX Variables.
<table>
<thead>
<tr>
<th>Key</th>
<th>Variable</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>$pagewindow</td>
<td>22</td>
<td>Specifies how many lines you see when you look at information that runs longer than one screen. Change this variable to match the number of lines on your terminal. If you set this variable to 0, it assumes a minimum of one line.</td>
</tr>
<tr>
<td>B</td>
<td>$pinmode</td>
<td>0</td>
<td>Prints input when you use the playback input command.</td>
</tr>
<tr>
<td>B</td>
<td>$printdata</td>
<td>0</td>
<td>Specifies whether to print the value of the registers where you disassemble instructions. A nonzero value means print the register value; a 0 means don't print the value.</td>
</tr>
<tr>
<td>B</td>
<td>$printwide</td>
<td>0</td>
<td>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.</td>
</tr>
<tr>
<td>S</td>
<td>$prompt</td>
<td>&quot;(dbx)&quot;</td>
<td>Sets the prompt for DBX.</td>
</tr>
<tr>
<td>B</td>
<td>$regstyle</td>
<td>1</td>
<td>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 regdefs.h. This variable does not affect coprocessor register names.</td>
</tr>
<tr>
<td>B</td>
<td>$repeatmode</td>
<td>1</td>
<td>Specifies whether you want DBX to repeat the last command when you press the carriage return. A nonzero value means repeat the last command; a 0 means don't repeat the last command.</td>
</tr>
<tr>
<td>B</td>
<td>$romode</td>
<td>0</td>
<td>Records input when you use the record output command.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>alias</td>
<td>See a list of all existing aliases.</td>
</tr>
<tr>
<td>alias NAME1 [ARG1..ARGN] &quot;NAME2&quot;</td>
<td>Define a new alias. NAME1 specifies the alias. NAME2 specifies the command. ARG1...ARGN let you specify arguments to the alias.</td>
</tr>
</tbody>
</table>

Example:

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

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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>unalias &quot;name&quot;</td>
<td>Remove an alias from a command, where name is the alias name.</td>
</tr>
</tbody>
</table>
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)
```

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

<table>
<thead>
<tr>
<th>Alias</th>
<th>Command</th>
<th>Select this alias to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi</td>
<td>playback input</td>
<td>Replay DBX commands that you saved with the record input command.</td>
</tr>
<tr>
<td>po</td>
<td>printf &quot;%0\n&quot;</td>
<td>Print the value of the specified expression or variable in octal.</td>
</tr>
<tr>
<td>pr</td>
<td>printregs</td>
<td>Print values for all registers.</td>
</tr>
<tr>
<td>px</td>
<td>printf &quot;%x\n&quot;</td>
<td>Print the value of the specified variable or expression in hexadecimal.</td>
</tr>
<tr>
<td>q</td>
<td>quit</td>
<td>End your debugging session.</td>
</tr>
<tr>
<td>r</td>
<td>rerun</td>
<td>Run your program again with the same arguments that you specified with the run command.</td>
</tr>
<tr>
<td>ri</td>
<td>record input</td>
<td>Record every command you type in a file.</td>
</tr>
<tr>
<td>ro</td>
<td>record output</td>
<td>Record all debugger output in the specified file.</td>
</tr>
<tr>
<td>s</td>
<td>step</td>
<td>Step the next number of specified lines.</td>
</tr>
<tr>
<td>si</td>
<td>stepi</td>
<td>Step the next number of specified lines of assembly code instructions.</td>
</tr>
<tr>
<td>t</td>
<td>where</td>
<td>Do a stack trace.</td>
</tr>
<tr>
<td>u</td>
<td>list $c\line–15:10</td>
<td>List the previous 10 lines.</td>
</tr>
<tr>
<td>w</td>
<td>list $c\line–5:10</td>
<td>List the 5 lines preceding and following the current line.</td>
</tr>
<tr>
<td>W</td>
<td>list $c\line–10:20</td>
<td>List the 10 lines preceding and following the current line.</td>
</tr>
<tr>
<td>wi</td>
<td></td>
<td>List the 5 machine instructions preceding and following the machine instruction.</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>record input [filename]</td>
<td>Record everything you type to DBX in a file.</td>
</tr>
</tbody>
</table>

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 $trimode. Use the playback output command to look at the recorded information, or use any UNIX system editor.
Syntax:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>record output [filename]</td>
<td>Record all DBX output in a file.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>playback input [filename]</td>
<td>Execute the commands from the specified file.</td>
</tr>
<tr>
<td>source [FILE]</td>
<td></td>
</tr>
</tbody>
</table>

Example:

```sh
(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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>playback output [filename]</td>
<td>Print the commands from the specified file.</td>
</tr>
</tbody>
</table>

Example:

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

Languages Programmer's Guide
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:**

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>sh</td>
<td>Call a shell from DBX.</td>
</tr>
<tr>
<td>sh [SHELL COMMAND]</td>
<td>Execute the shell command.</td>
</tr>
</tbody>
</table>

**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)
```

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:**

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>Check the status of commands.</td>
</tr>
</tbody>
</table>
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:

```
<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete EXP1,...EXPN</td>
<td>Delete the specified status item (EXP) from the status list.</td>
</tr>
<tr>
<td>delete all</td>
<td>Delete all status items.</td>
</tr>
</tbody>
</table>
```

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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>use</td>
<td>List the current directories.</td>
</tr>
<tr>
<td>use DIR1 ... DIRN</td>
<td>Specify different directories.</td>
</tr>
</tbody>
</table>

Example:

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

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:**

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

**Example:**

```
(dbx) where
> 0 printline [pline = 0x7fff5b80) ["sam.c":58,0x2f7]
 1 $block1 ["sam.c":47, 0x2bb]
 2 main [argc=2, argv=0x7fff6b0]["sam.c":47,0x2bb]
(dbx) func 2
main 47       printline(&linel)
(dbx) func main
(dbx)        the procedure name
the procedure's arguments
the source file name
the current program counter
the current line
the activation level
```

**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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>Print the name of the file you’re currently using.</td>
</tr>
<tr>
<td>file FILE</td>
<td>Change the current file to the specified file.</td>
</tr>
</tbody>
</table>

Example:

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

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:

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

For example:

```
(dbx) list 53:2 ← the user specified a list starting at line 53 for two lines
    53
    54  LINETYPE *pline;
(dbx)
```
If you use the predefined alias w, (see Predefined DBX Aliases), you get something like this:

```c
(dbx) w
  53
  54 LINE_TYPE  *pline;
  55
  56 {
  57  fprintf(stdout, #53d. (%d)%s", pline->line_number
  58  pline->string;   current line
  59  ff; isj(stdout);
  60  } /* println */
(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:**

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>/REGEX</td>
<td>Search ahead in the code for the specified regular expression.</td>
</tr>
<tr>
<td>?REGEX</td>
<td>Search back in the code for the specified regular expression.</td>
</tr>
</tbody>
</table>

**Example:**

```c
(dbx) /lines continue; /* don't count blank lines */
(dbx) /lines linel.length=1
(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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>edit</td>
<td>Call an editor from DBX on the current file.</td>
</tr>
<tr>
<td>edit [filename]</td>
<td>Call an editor and edit the specified file.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>which VAR</td>
<td>Print the default version of the variable.</td>
</tr>
<tr>
<td>whereis VAR</td>
<td>Print all versions of the specified variable.</td>
</tr>
</tbody>
</table>

**Example:**

```
(dbx) which i
sam.main.i
(dbx) whereis i
sam.printline.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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>whatis VAR</td>
<td>Print the type declaration for the specified variable or procedure in your program.</td>
</tr>
</tbody>
</table>

Languages Programmer’s Guide
Example:

```c
(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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>run [ARG1,...,ARGN] [&lt;FILE1] [&gt;FILE2]</code></td>
<td>Run your program with the specified arguments.</td>
</tr>
<tr>
<td><code>rerun [ARG1,...,ARGN] [&lt;FILE1] [&gt;FILE2]</code></td>
<td>Rerun your program with the arguments you specified to the <code>run</code> command or with new arguments.</td>
</tr>
</tbody>
</table>

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:**

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>step [EXP] *</code></td>
<td>Execute the specified number of lines of source code. EXP refers to the number of lines to be executed in both the current procedure and any called procedures.</td>
</tr>
<tr>
<td><code>next [EXP] *</code></td>
<td>Execute the specified number of lines of source code. EXP refers to the number of lines to be executed in only the current procedure, regardless of any called procedures executed.</td>
</tr>
</tbody>
</table>

*Default is 1.*

**Example:**

```plaintext
(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(linel.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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>Execute the current procedure and return to the next sequential line in the calling procedure.</td>
</tr>
<tr>
<td>return PROCEDURE</td>
<td>Execute the program until DBX returns to the specified procedure.</td>
</tr>
</tbody>
</table>

Example:

```
(dbx) rerun
[6] stopped at [printline: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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>goto LINE</td>
<td>Go to a specified line and continue execution.</td>
</tr>
</tbody>
</table>

Example:

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

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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>cont</td>
<td>Continue from the current line.</td>
</tr>
<tr>
<td>cont to LINE</td>
<td>Continue until the specified line.</td>
</tr>
<tr>
<td>cont in PROCEDURE</td>
<td>Continue until the specified procedure.</td>
</tr>
<tr>
<td>cont SIGNAL</td>
<td>Continue from the current line and send the signal.</td>
</tr>
<tr>
<td>cont SIGNAL to LINE</td>
<td>Continue until reaching the specified line and send the signal.</td>
</tr>
<tr>
<td>cont SIGNAL in PROCEDURE</td>
<td>Continue until reaching the specified procedure and send the signal.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign EXP1 = EXP2</td>
<td>Assign a new value to a program variable.</td>
</tr>
</tbody>
</table>

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 address to be an integer and assign a 1 to it
1
(dbx)
```
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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>stop [VAR] at LINE</td>
<td>Stop at the current line.</td>
</tr>
<tr>
<td>stop [VAR] at LINE</td>
<td>Stop at a specified line.</td>
</tr>
<tr>
<td>stop [VAR] at LINE if EXP</td>
<td>Stop at a specified line only if the expression is true.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>stop in PROCEDURE</td>
<td>Stop at the beginning of the procedure.</td>
</tr>
<tr>
<td>stop VAR in PROCEDURE</td>
<td>Stop in the specified procedure when VAR changes.</td>
</tr>
<tr>
<td>stop in PROCEDURE if EXP*</td>
<td>Stop in the specified procedure if EXP is true.</td>
</tr>
<tr>
<td>stop VAR in PROCEDURE if EXP*</td>
<td>Stop in the specified procedure when VAR changes and EXP is true.</td>
</tr>
</tbody>
</table>

*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 println
[15] stop in println
(dbx) rerun
[15] stopped at [println:58,02f8] pline->string);
```

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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>stop if EXP</td>
<td>Stop if EXP is true.</td>
</tr>
<tr>
<td>stop VAR if EXP</td>
<td>Stop if VAR changes and EXP is true.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace VAR</td>
<td>List the specified variable after each source line is executed.</td>
</tr>
<tr>
<td>trace VAR at LINE</td>
<td>List the specified variable at the specified line.</td>
</tr>
<tr>
<td>trace VAR in PROCEDURE</td>
<td>List the specified variable in the specified procedure.</td>
</tr>
<tr>
<td>trace VAR at LINE if EXP</td>
<td>List the variable at the specified line when the expressions is true.</td>
</tr>
<tr>
<td>trace VAR in PROCEDURE if EXP</td>
<td>List the variable in the specified procedure when the expression is true.</td>
</tr>
</tbody>
</table>

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);
[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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>when VAR [if EXP] {COMMAND_LIST}</td>
<td>Execute the command list when VAR changes.</td>
</tr>
<tr>
<td>when [VAR] at LINE [if EXP] {COMMAND_LIST}</td>
<td>Execute the command list when VAR changes and the debugger encounters line.</td>
</tr>
<tr>
<td>when in PROCEDURE {COMMAND_LIST}</td>
<td>Execute the command list upon entering PROCEDURE.</td>
</tr>
<tr>
<td>when [VAR] in PROCEDURE [if EXP] {COMMAND_LIST}</td>
<td>Execute the specified commands on each line of PROCEDURE when EXP is true and VAR changes.</td>
</tr>
</tbody>
</table>

NOTE: EXP is checked before VAR.

Example:

```
(dbx) when in println {print i}
[14] print i in println
(dbx) rerun
[14] stopped at [println:58,0x2f8] pline->string);
(dbx) cont
0. (19) #include <stdio.h>
14 value of i
[14] stopped at [println:58,0x2f8] pline->string);
(dbx) cont
1. (14) struct line {
22 value of i
[14] stopped at [println:58,0x2f8] pline->string);
(dbx) when in println {stop]
[15] stop in println
(dbx) rerun
[15] stopped at [println:58, 0x2f8] pline->string);
(dbx) DBX stops in the procedure println
```

Stopping at Signals (catch and ignore)

The catch command lists the signals that DBX catches or specifies a signal for DBX to catch. If a child in the program encounters a specified signal, DBX stops the process and gives you control.
Syntax:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>catch</td>
<td>Print a list of all signals that DBX will catch.</td>
</tr>
<tr>
<td>catch SIGNAL</td>
<td>Add a signal to the catch list.</td>
</tr>
<tr>
<td>ignore</td>
<td>Print a list of all signals that DBX will not catch.</td>
</tr>
<tr>
<td>ignore SIGNAL</td>
<td>Remove a signal from the catch list and add it to the ignore list.</td>
</tr>
</tbody>
</table>

Example:

```
(dbx) catch
INT QUIT ILL TRAP IOT EMT FPE BUS SEGV SYS PIPE TERM STOP TTIN
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 SCPU XFSZ
(dbx) ignore
HUP ALRM TSTP CONT CHLD
(dbx)                                adds KILL
                                        from the ignore
                                        list
                                        removes 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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>where</td>
<td>Do a stack trace.</td>
</tr>
</tbody>
</table>

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]
> $block1 ["sam.c":47,0x2bb]
> 2 main [argv = 2, argv = 0x7fffefa0] ["sam.c":47,0x2bb]
```

```
(dbx)     the procedure name
>0         the current value of the argument pline
> $block1   the program counter
> 2 main    the line number

the activation level number—\nthe > shows that the user is examining this activation level
```

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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>up [EXP]</td>
<td>Move up the specified number of activation levels in the stack. The default is one level.</td>
</tr>
<tr>
<td>down [EXP]</td>
<td>Move down the specified number of activation levels in the stack. The default is one level.</td>
</tr>
</tbody>
</table>

Example:

```
(dbx) where
>0 println(pline = 0x7fff5b80) ["sam.c":58,0x2f7]
  1 $block1 ["sam.c":47,0x2bb]
  2 main(argc = 2, argv = 0x7ffffe8a0) ["sam.c":47,0x2bb]
(dbx) down $block1 ["sam.c":47,0x2bb] moves down one level
(dbx) where
  0 println(pline = 0x7fff5b80) ["sam.c":58,0x2f7]
  >1 $block1 ["sam.c":47,0x2bb] moves up one level
>1 println(pline = 0x7fff5b80) ["sam.c":58,0x2f7]
(dbx) where
  0 println(pline = 0x7fff5b80) ["sam.c":58,0x2f7]
  1 $block1 ["sam.c":47,0x2bb]
  2 main(argc = 2, argv = 0x7ffffe8a0) ["sam.c":47,0x2bb]
(dbx)
```

Printing (print and printf)

The `print` commands 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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to…</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>print EXP1,…,EXPN</code></td>
<td>Print the value of the specified expressions.</td>
</tr>
<tr>
<td><code>printf “STRING”, EXP1,…,EXPN</code></td>
<td>Print the value of the specified expressions in the format as specified by the string.</td>
</tr>
</tbody>
</table>

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
(dbx) pd i
14
(dbx) po i
016
(dbx) px i
0xe
(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:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Register Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r</td>
<td>Machine register.</td>
</tr>
<tr>
<td>$f</td>
<td>Floating point.</td>
</tr>
<tr>
<td>$d</td>
<td>Double precision floating point.</td>
</tr>
<tr>
<td>$pc</td>
<td>Program counter value.</td>
</tr>
</tbody>
</table>

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:**

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>printregs</td>
<td>Print the current values of all registers.</td>
</tr>
</tbody>
</table>

**Example:**

```plaintext
(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/b0=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/x0=0  r27/x1=0
r28/gp=50529  r29/s0=2147441400  r30/fp=2147442536  r31/ra=700
$0f= 0.0  $f1= 0.0  $f2= 0.0  $f3= 0.0
$ff4= 0.0  $f5= 0.0  $f6= 0.0  $f7= 0.0
$ff8= 0.0  $f9= 0.0  $f10= 0.0  $f11= 0.0
$ff12= 0.0  $f13= 0.0  $f14= 0.0  $f15= 0.0
$ff16= 0.0  $f17= 0.0  $f18= 0.0  $f19= 0.0
$ff20= 0.0  $f21= 0.0  $f22= 0.0  $f23= 0.0
$ff24= 0.0  $f25= 0.0  $f26= 0.0  $f27= 0.0
$ff28= 0.0  $f29= 0.0  $f30= 0.0  $f31= 0.0
$dd0= 0.0  $d2= 0.0  $d4= 0.0  $d6= 0.0
$dd8= 0.0  $d10= 0.0  $d12= 0.0  $d14= 0.0
$dd16= 0.0  $d18= 0.0  $d20= 0.0  $d22= 0.0
$ddd4= 0.0  $d26= 0.0  $d28= 0.0  $d30= 0.0

(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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>dump</td>
<td>Print information about the current activation level.</td>
</tr>
<tr>
<td>dump .</td>
<td>Print information about all activation levels in your program.</td>
</tr>
<tr>
<td>dump PROCEDURE</td>
<td>Print information about the specified procedure (activation level).</td>
</tr>
</tbody>
</table>

Example:

```
(dbx) where
>0 println (pline=0x7fff5b80) [“sam.c”:58,0x2f7]
  1 $block1 [“sam.c”:47,0x2bb]
(dbx) dump
println (pline=0x7fff5b80) [“sam.c”:58,0x2f7]
(dbx) dump .
> 0 println(pline=0x7fff5b80) [“sam.c”:58,0x2f7]
  1 $block1 [“sam.c”:47,0x2bb]
  curlinenumber = 1
i=19
  2 main (argc=2, argv=0x7fffeba0) [“sam.c”:47,0x2bb]
    fd = 0x4270
    line1=struct {
      string=“#include<stdio.h>"
    }
    linenumber=0
  }
in “”;  
(dbx) dump main
main (argc=2, argv=0x7fffeba0) [“saam.c”:47,0x2bb]
  fd=0x4270
  line1=struct {
    string=“struct line {
      length = 14
      linenumber = 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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stopi [VAR] at</code></td>
<td>Stop at the current line.</td>
</tr>
<tr>
<td><code>stopi [VAR] at LINE</code></td>
<td>Stop at a specified address.</td>
</tr>
<tr>
<td><code>stopi [VAR] at LINE if EXP</code></td>
<td>Stop at a specified address only if EXP is true.</td>
</tr>
<tr>
<td><code>stopi if EXP</code></td>
<td>Stop if EXP is true.</td>
</tr>
<tr>
<td><code>stopi VAR if EXP</code></td>
<td>Stop if VAR changes and EXP is true.</td>
</tr>
<tr>
<td><code>stopi in PROCEDURE</code></td>
<td>Stop at the beginning of the procedure.</td>
</tr>
<tr>
<td><code>stopi VAR in PROCEDURE</code></td>
<td>Stop in the specified procedure when VAR changes.</td>
</tr>
<tr>
<td><code>stopi in PROCEDURE if EXP *</code></td>
<td>Stop in the specified procedure if EXP is true.</td>
</tr>
<tr>
<td><code>stopi VAR in PROCEDURE if EXP *</code></td>
<td>Stop in the specified procedure when VAR changes and EXP is true.</td>
</tr>
</tbody>
</table>

*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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>conti SIGNAL</td>
<td>Send the specified signal and continue.</td>
</tr>
<tr>
<td>conti to ADDRESS</td>
<td>Continue until reaching the specified address.</td>
</tr>
<tr>
<td>conti in PROCEDURE</td>
<td>Continue until the beginning of the specified procedure.</td>
</tr>
<tr>
<td>conti SIGNAL to ADDRESS</td>
<td>Continue until reaching the specified address, then send the signal.</td>
</tr>
<tr>
<td>conti SIGNAL in PROCEDURE</td>
<td>Continue until reaching the beginning of the specified procedure, then send the signal.</td>
</tr>
</tbody>
</table>

Example:

```
(dbx) conti
0 (19)#include <stdio.h>
[2] stopped at [printfline:58,0x2f8] pline->string0;
lw r2,32(sp)
(dbx)
```

Executing Single Lines of Machine Code (stepi and nexti)

The stepi and nexti commands execute a fixed number of machine instructions as specified by EXP. If you don’t specify EXP for stepi and nexti, DBX executes one machine instruction. If you specify EXP, DBX executes the machine instructions as follows:

- For stepi 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 stepi, 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.
Chapter 5

- 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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>stepi [EXP]*</td>
<td>Execute the specified number of lines of source code. EXP refers to the number of lines to be executed in both the current procedure and any procedure calls.</td>
</tr>
<tr>
<td>nexti [EXP]*</td>
<td>Execute the specified number of lines of source code. EXP refers to the number of lines to be executed in only the current procedure, regardless of any procedure calls.</td>
</tr>
</tbody>
</table>

*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:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>tracei VAR</td>
<td>Print the value of the variable as it changes instruction by instruction.</td>
</tr>
<tr>
<td>tracei VAR at ADDRESS</td>
<td>Print the value of the variable when it changes instruction by instruction.</td>
</tr>
<tr>
<td>tracei VAR in PROCEDURE</td>
<td>Print the value of the variable when it changes in the specified procedure.</td>
</tr>
<tr>
<td>tracei VAR at ADDRESS if EXP</td>
<td>Print the value of the variable at the specified address when the expression is true.</td>
</tr>
<tr>
<td>tracei VAR in PROCEDURE if EXP</td>
<td>Print the value of the variable in the specified procedure when it traces instruction by instruction.</td>
</tr>
</tbody>
</table>

Printing the Contents of Memory

Entering values in the syntax shown below prints the contents of memory according to your specifications.

Syntax:

<table>
<thead>
<tr>
<th>Command</th>
<th>Select this command to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDRESS/&lt;COUNT&gt;&lt;MODE&gt;</td>
<td>Print the contents of the specified address for the specified count.</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Mode</th>
<th>Select this mode to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Print a short word in decimal.</td>
</tr>
<tr>
<td>D</td>
<td>Print a long word in decimal.</td>
</tr>
<tr>
<td>o</td>
<td>Print a short word in octal.</td>
</tr>
<tr>
<td>O</td>
<td>Print a long word in octal.</td>
</tr>
<tr>
<td>x</td>
<td>Print a short word in hexadecimal.</td>
</tr>
<tr>
<td>X</td>
<td>Print a long word in hexadecimal.</td>
</tr>
<tr>
<td>b</td>
<td>Print a byte in octal.</td>
</tr>
<tr>
<td>c</td>
<td>Print a byte as a character.</td>
</tr>
<tr>
<td>s</td>
<td>Print a string of characters that ends in a null byte.</td>
</tr>
<tr>
<td>f</td>
<td>Print a single precision real number.</td>
</tr>
<tr>
<td>g</td>
<td>Print a double precision real number.</td>
</tr>
<tr>
<td>i</td>
<td>Print machine instructions.</td>
</tr>
</tbody>
</table>

**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:58,0x31c] lui r1,0x0
[printline:58,0x320] jal fflush
[printline:58,0x324] addiu r4,r1,16960
(dbx) 0x2f8/10d
 000002f8: 32 3677 0 0 15361 1690 9252 0 15361
 00000308: 16780 9253
```

*Languages Programmer's Guide*
When you disassemble instructions, you will see something like this:

```
( dbx ) &print line/10i
* [ print line: 58, 0x2ec ] addiu sp, sp, -32
[ print line: 58, 0x2f0 ] sw f4, 32(sp)
[ print line: 58, 0x2f4 ] sw r31, 28(sp)
[ print line: 58, 0x2f8 ] lw r2, 32(sp)
[ print line: 58, 0x2fc ] lui r1, 0x0
[ print line: 58, 0x300 ] addiu r4, r1, 16960
[ print line: 58, 0x304 ] lui r1, 0x0
[ print line: 58, 0x308 ] addiu r5, r1, 16780
[ print line: 58, 0x30c ] lw r6, 260(r2)
[ print line: 58, 0x310 ] lw r7, 256(r2)
( dbx ) &print line/10d
000002ec: 65504 10173 32 44963 23 15361
           449628 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.
<table>
<thead>
<tr>
<th>Command</th>
<th>Alias</th>
<th>Select this command to...</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td></td>
<td>Search ahead in the code for the specified string.</td>
<td>/REGEX</td>
</tr>
<tr>
<td>?</td>
<td></td>
<td>Search back in the code for the specified string.</td>
<td>?REGEX</td>
</tr>
</tbody>
</table>
| !       |       | 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.*
<table>
<thead>
<tr>
<th>Command</th>
<th>Alias</th>
<th>Select this command to...</th>
<th>Syntax</th>
</tr>
</thead>
</table>
| conti   |       | Continue executing assembly code after a breakpoint. | conti SIGNAL  
conti to ADDRESS  
conti in PROCEDURE  
conti SIGNAL to ADDRESS  
conti SIGNAL in PROCEDURE |
| delete  | d     | Delete the specified item from the status list. | delete EXP1,...,EXPn  
delete ALL |
| down    |       | Move down the specified number of activation levels in the stack. The default is one level. | down [EXP] |
| dump    |       | Print variable information about the procedure. If you specify a dot (.), this command prints global variable information for all procedures. | dump PROCEDURE  
dump . |
| edit    |       | Call an editor from DBX. | edit [FILE] |
| 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. | file [FILE] |
| func    | f     | Move to the specified procedure (activation level) or print the current activation level. | func  
func EXP  
func PROCEDURE |
| goto    | g     | Go to the specified line. | goto LINE |

*Table 5.3 (2 of 7). Command Summary.*
<table>
<thead>
<tr>
<th>Command</th>
<th>Alias</th>
<th>Select this command to...</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>help</td>
<td>?</td>
<td>Print a list of DBX commands. Uses UNIX system more.</td>
<td>help</td>
</tr>
<tr>
<td>history</td>
<td>h</td>
<td>Print a list of the previous commands that you've issued. The default is 20.</td>
<td>history</td>
</tr>
<tr>
<td>ignore</td>
<td></td>
<td>List all signals that DBX does not catch, or, if you specify an argument, add the specified signal to the ignore list.</td>
<td>ignore [SIGNAL]</td>
</tr>
<tr>
<td>list</td>
<td>l</td>
<td>List the specified lines. The default is 10 lines.</td>
<td>list [EXP:INT], list [EXP]</td>
</tr>
<tr>
<td>next</td>
<td>n</td>
<td>Step over the specified number of lines. The default is one. This command does not step into procedures.</td>
<td>next [INT]</td>
</tr>
<tr>
<td>nexti</td>
<td>ni</td>
<td>Step over the specified number of machine instructions. The default is one. This command does not step into procedures.</td>
<td>nexti [INT]</td>
</tr>
<tr>
<td>playback input</td>
<td>pi</td>
<td>Replay commands that you saved with the record input command in a text file.</td>
<td>playback input [FILE]</td>
</tr>
</tbody>
</table>

Table 5.3 (3 of 7). Command Summary.
<table>
<thead>
<tr>
<th>Command</th>
<th>Alias</th>
<th>Select command to...</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>playback</td>
<td>po</td>
<td>Replay debugger output that you saved with the record output command.</td>
<td>playback output [FILE]</td>
</tr>
<tr>
<td>output</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>print</td>
<td>p</td>
<td>Print the value of the specified expression.</td>
<td>print EXP1,...,EXPN</td>
</tr>
<tr>
<td>printf</td>
<td>pd</td>
<td>Print the value of the specified expression, using C string formatting.</td>
<td>printf &quot;STRING&quot;, EXP1,...,EXPN</td>
</tr>
<tr>
<td>printregs</td>
<td>pr</td>
<td>Print all register values.</td>
<td>printregs</td>
</tr>
<tr>
<td>quit</td>
<td>q</td>
<td>Exit DBX.</td>
<td>quit</td>
</tr>
<tr>
<td>record input</td>
<td>ri</td>
<td>Record all commands you type to DBX.</td>
<td>record input [FILE]</td>
</tr>
<tr>
<td>record output</td>
<td>ro</td>
<td>Record all DBX output.</td>
<td>record output [FILE]</td>
</tr>
<tr>
<td>return</td>
<td></td>
<td>Continue executing until the procedure returns. If you don't specify a procedure, DBX assumes the next procedure.</td>
<td>return [PROCEDURE]</td>
</tr>
<tr>
<td>run</td>
<td></td>
<td>Run your program.</td>
<td>run [ARG1 ... ARGN] [&lt;FILE1&gt;]</td>
</tr>
<tr>
<td>rerun</td>
<td>r</td>
<td>Run your program again, using the same arguments you specified to the run command.</td>
<td>rerun [ARG1 ... ARGN] [&lt;FILE1&gt;]</td>
</tr>
</tbody>
</table>

Table 5.3 (4 of 7). Command Summary.
<table>
<thead>
<tr>
<th>Command</th>
<th>Alias</th>
<th>Select command to...</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td></td>
<td>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.</td>
<td>set&lt;br&gt;set VAR = EXP</td>
</tr>
<tr>
<td>sh</td>
<td></td>
<td>Call a shell from DBX, or, execute a shell command.</td>
<td>sh [SHELL COMMAND]</td>
</tr>
<tr>
<td>source</td>
<td></td>
<td>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 input command.</td>
<td>source [FILE]</td>
</tr>
<tr>
<td>status</td>
<td>j</td>
<td>Print a list of currently set breakpoints, record commands, and traces.</td>
<td>status</td>
</tr>
<tr>
<td>step</td>
<td>s</td>
<td>Step the specified number of lines. This command steps into procedures. The default is one line.</td>
<td>step [INT]</td>
</tr>
<tr>
<td>stepl</td>
<td>si</td>
<td>Step the specified number of machine instructions. This command steps into procedures. The default is one instruction.</td>
<td>stepl [INT]</td>
</tr>
<tr>
<td>stop</td>
<td>b</td>
<td>Set a breakpoint at the specified point.</td>
<td>stop [VAR] at&lt;br&gt;stop [VAR] at LINE&lt;br&gt;stop [VAR] in PROCEDURE&lt;br&gt;stop [VAR] if EXP&lt;br&gt;stop [VAR] at LINE if EXP&lt;br&gt;stop [VAR] in PROCEDURE if EXP</td>
</tr>
<tr>
<td></td>
<td>bp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.3 (5 of 7). Command Summary.
<table>
<thead>
<tr>
<th>Command</th>
<th>Alias</th>
<th>Select command to...</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>stopi</td>
<td></td>
<td>Set a breakpoint in machine code at the specified point.</td>
<td>stopi [VAR] at ADDRESS&lt;br&gt;stopi [VAR] in PROCEDURE&lt;br&gt;stopi [VAR] if EXP&lt;br&gt;stopi [VAR] at ADDRESS if EXP&lt;br&gt;stopi [VAR] in PROCEDURE if EXP</td>
</tr>
<tr>
<td>trace</td>
<td>tr</td>
<td>Trace the specified variable.</td>
<td>trace VAR&lt;br&gt;trace VAR at LINE&lt;br&gt;trace VAR in PROCEDURE&lt;br&gt;trace VAR at LINE if EXP&lt;br&gt;trace VAR in PROCEDURE if EXP</td>
</tr>
<tr>
<td>tracei</td>
<td></td>
<td>Trace the specified variable in the machine instruction.</td>
<td>tracei VAR&lt;br&gt;tracei VAR at ADDRESS&lt;br&gt;tracei VAR in PROCEDURE&lt;br&gt;tracei VAR at ADDRESS if EXP&lt;br&gt;tracei VAR in PROCEDURE if EXP</td>
</tr>
<tr>
<td>unalias</td>
<td></td>
<td>Remove specified alias.</td>
<td>unalias ALIAS NAME</td>
</tr>
<tr>
<td>unset</td>
<td></td>
<td>Unset a debugger variable.</td>
<td>unset VAR</td>
</tr>
<tr>
<td>up</td>
<td></td>
<td>Move the specified number of activation levels up the stack. The default is one.</td>
<td>up [EXP]</td>
</tr>
<tr>
<td>use</td>
<td></td>
<td>Print a list of the source directories, or, if you specify a directory name, this command substitutes the new directories for the previous list.</td>
<td>use [DIR1 DIR2...DIRN]</td>
</tr>
</tbody>
</table>

*Table 5.3 (6 of 7). Command Summary.*
<table>
<thead>
<tr>
<th>Command</th>
<th>Alias</th>
<th>Select command to...</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>whatis</td>
<td></td>
<td>Print the type declaration for the specified name.</td>
<td>whatis VAR</td>
</tr>
<tr>
<td>when</td>
<td></td>
<td>Execute the specified DBX commands during execution.</td>
<td>when [VAR][if EXP]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>{COMMAND_LIST}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>when [VAR] at LINE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[if EXP]{COMMAND_LIST}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>when [VAR] in PROCEDURE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[if EXP]{COMMAND_LIST}</td>
</tr>
<tr>
<td>where</td>
<td>t</td>
<td>Do a stack trace, which shows the current activation levels.</td>
<td>where</td>
</tr>
<tr>
<td>whereis</td>
<td></td>
<td>Print all qualifications of the specified variable name.</td>
<td>whereis VAR</td>
</tr>
<tr>
<td>which</td>
<td></td>
<td>Print the qualification of the variable name currently in use.</td>
<td>which VAR</td>
</tr>
<tr>
<td>examine</td>
<td></td>
<td>Print the contents of the specified address or disassemble the code for the instruction at the specified address.</td>
<td>ADDRESS/&lt;COUNT&gt;&lt;MODE&gt;</td>
</tr>
<tr>
<td>address</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 linenum;
};

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 linenumbers.
     */
    while (fgets(line1.string, sizeof(line1.string), fd) != NULL) {
        int i;
        static curlinenum = 0;
        i = strlen(line1.string);
        if (i == 1 && line1.string[0] == '\n')
            continue; /* don't count blank lines */
        line1.length = i;
        line1.linenum = curlinenum++;
        printline(&line1);
    } /* while */
} /* main */
void printline (pline)
LINETYPE *pline;
{ fprintf (stdout, \"%3d. (%3d) %s\n",
    pline->linenum, 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.

```c
/* @(#)varargs.h 1.2 */
typedef char *va_list;
#define va_dcl int va_dlist;
#define va_start(list) list = (char *) &va_dlist
#define va_end(list) 
#define va_arg(list, mode) ((mode *)(list + \n (char *)(list + sizeof(mode) > 4 ? (int)list + 2*8 - 1 & -8 \n : (int)list + 2*4 - 1 & -4))[1]
*/
```

The `va_dcl` macro declares the formal parameters `va_dlist`, 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 traverse 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>
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_list )
/* 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.

<table>
<thead>
<tr>
<th>C Specification</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nesting levels</td>
<td></td>
</tr>
<tr>
<td>Compound statements</td>
<td>≤ 30</td>
</tr>
<tr>
<td>Iterations</td>
<td></td>
</tr>
<tr>
<td>Selections</td>
<td></td>
</tr>
<tr>
<td>Conditional compilations</td>
<td></td>
</tr>
<tr>
<td>Maximum number of type modifiers (arrays, pointers,</td>
<td>9</td>
</tr>
<tr>
<td>function, volatile)</td>
<td></td>
</tr>
<tr>
<td>Case labels</td>
<td>500</td>
</tr>
<tr>
<td>Function call parameters</td>
<td>150</td>
</tr>
<tr>
<td>Significant characters</td>
<td></td>
</tr>
<tr>
<td>External identifier</td>
<td>≤ 32</td>
</tr>
<tr>
<td>Internal identifier</td>
<td></td>
</tr>
</tbody>
</table>

*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.

```pascal
writeln(output, "\a");
```

The following table lists the escape character sequence used to encode special characters.

<table>
<thead>
<tr>
<th>Character</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>alert (16#07)</td>
</tr>
<tr>
<td>\b</td>
<td>backspace (16#08)</td>
</tr>
<tr>
<td>\f</td>
<td>form feed (16#0c)</td>
</tr>
<tr>
<td>\n</td>
<td>newline (16#0a)</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return (16#0d)</td>
</tr>
<tr>
<td>\t</td>
<td>horizontal tab (16#09)</td>
</tr>
<tr>
<td>\v</td>
<td>vertical feed (16#0b)</td>
</tr>
<tr>
<td>\</td>
<td>backslash (16#5c)</td>
</tr>
<tr>
<td>&quot;</td>
<td>quotation mark (16#22)</td>
</tr>
<tr>
<td>'</td>
<td>single quote (16#27)</td>
</tr>
<tr>
<td>\nnn</td>
<td>character with octal value of nnn.</td>
</tr>
<tr>
<td>\x nnn</td>
<td>character with hexadecimal value of nnn.</td>
</tr>
</tbody>
</table>

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.

```pascal
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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>−</td>
<td>subtraction and unary minus</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>div</td>
<td>integer division</td>
</tr>
<tr>
<td>mod</td>
<td>modulo</td>
</tr>
<tr>
<td>=</td>
<td>equality relation</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>inequality relation</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>(</td>
<td>parenthesis</td>
</tr>
<tr>
<td>bitand</td>
<td>bit-wise and</td>
</tr>
<tr>
<td>bitor</td>
<td>bit-wise or</td>
</tr>
<tr>
<td>bitxor</td>
<td>bit-wise exclusive or</td>
</tr>
<tr>
<td>lshift</td>
<td>logical left shift</td>
</tr>
<tr>
<td>rshift</td>
<td>logical right shift</td>
</tr>
<tr>
<td>lbound</td>
<td>low bound of an array</td>
</tr>
<tr>
<td>hbound</td>
<td>high bound of an array</td>
</tr>
<tr>
<td>first</td>
<td>lowest value of a scalar type</td>
</tr>
<tr>
<td>last</td>
<td>highest value of a scalar type</td>
</tr>
<tr>
<td>sizeof</td>
<td>the size (in bytes) of a data type</td>
</tr>
<tr>
<td>abs</td>
<td>absolute value</td>
</tr>
<tr>
<td>chr</td>
<td>inverse ordinal value for a character</td>
</tr>
<tr>
<td>ord</td>
<td>the ordinal value of a scalar value</td>
</tr>
<tr>
<td>pred</td>
<td>the predecessor to a scalar value</td>
</tr>
<tr>
<td>succ</td>
<td>the successor to a scalar value</td>
</tr>
<tr>
<td>type−functions</td>
<td>converts from one type to another</td>
</tr>
</tbody>
</table>

Pascal does not permit the use of a leading left parenthesis in constant expressions for the lower value of subrange types. That is,

\[
\text{subrange} = (11+12) \cdot 13 \ldots 14+15;
\]

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.

```pascal
case i of
  1900..1999 : writeln('twentieth century');
  1800..1999 : writeln('nineteenth century');
  1700..1799 : writeln('eighteenth century');
  1600..1699 : writeln('seventeenth century');
otherwise :
  writeln(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.

```pascal
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.

```plaintext
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.

```plaintext
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:

```
Header file externs.h

const
  vector_size = 100;
type
  vector_index = 0..vector_size-1;
  vector = array[vector_index] of integer;
var
  v : vector;
procedure initialize;
  external;
function sum(var vect : vector) : integer;
  external;

#include "externs.h"

program main(output);
begin
  initialize;
  writeln(sum(v));
end.
```

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:

\[
\text{var-decl} ::= \text{identifier-list} \text{"." type-denoter} \text{["=" initial-clause]}
\]

\[
\text{initial-clause} ::= \text{constant-expr} \text{["=" \text{initial-value-list}]}
\]

\[
\text{initial-value} ::= \text{constant-expr} \text{[".." constant-expr]}
\]

\[
\text{constant-expr} ::= \text{constant-expr} \text{"otherwise" constant-expr}
\]

\[
\text{initial-value-list} ::= \text{initial-value} \text{["=" \text{initial-value-list}]}
\]

For scalar types, this initialization is very simple:

\[
\text{var}
\]

\[
a: \text{integer} := 5; \]

\[
\text{letter: char} := \text{'}x\text{'}; \]

\[
x1: \text{real} := 6.5; \]

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.

\[
\text{type}
\]

\[
\text{color} = \{\text{red, yellow, blue}\}; \]

\[
\text{hue} = \text{set of color}; \]

\[
\text{vector} = \text{array}[1..100] \text{ of integer}; \]

\[
\text{var}
\]

\[
\text{orange} : \text{hue} := \{\text{red, yellow}\}; \]

\[
\text{black} : \text{hue} := \{\text{first(color)\ldots last(color)}\}; \]

\[
\text{name} : \text{packed array}[1..32] \text{ of char} := \text{'}MIPS \text{ Computer Systems'}; \]

\[
\text{vect} : \text{vector} := \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 30 : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \text{otherwise} : 99\}; \]

\[
\text{pointers} : \text{array}[0..127] \text{ of } \text{^vector} := \{\text{otherwise: nil}\}; \]

The above clauses initialize two sets, a string and two arrays. All elements of the array \text{vect}, except the first ten elements and the ten elements starting at index 30, are initialized to 99. The array \text{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

\[ \text{assert} \left( \text{boolean-exp} \ [\ , \text{string}] \right) \]

The \text{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

\[ \text{assertion error in Pascal program} \]

is generated along with the line number and file name of the \text{assert} statement that causes the error.

Argv

\[ \text{argv} \left( \text{integer-exp}, \text{string-var} \right) \]

The \text{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 \(0..\text{argc}-1\). Argument 0 is the name of the program.

Date

\[ \text{date} \left( \text{string-var} \right) \]

The \text{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

\[ \text{time} \left( \text{string-var} \right) \]

The \text{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, \text{ord} and \text{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.).

```plaintext
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**

```plaintext
scalar-var := min(scalar-expr [, scalar-expr] ...)
```

The min function returns the smallest of its scalar arguments. For example:

\[ \text{min}(-6, 3, 5) \]

returns \(-6\).

**Max**

```plaintext
scalar-var := max(scalar-expr [, scalar-expr] ...)
```

The max function returns the largest of its scalar arguments. For example:

\[ \text{max}(-2, 3, 5) \]

returns \(5\).

**Lbound**

```plaintext
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–dimensioned, then an optional second argument specifies the dimension; 1, specifying the first or outermost dimension, is the default.

**Hbound**

```plaintext
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–dimensioned, then an optional second argument specifies the dimension; 1 is the default which means the first (outermost) dimension.

---

Languages Programmer's Guide
First

\[ \text{scalar-var} := \text{first}(\text{type-identifier}) \]

The \text{first} function returns the first (lowest) value of the named scalar type. For example, \text{first(integer)} returns \(-2147483848\).

Last

\[ \text{scalar-var} := \text{last}(\text{type-identifier}) \]

The \text{last} function returns the last (highest) value of the named scalar type. For example, \text{last(integer)} returns \(2147483847\) (maxint).

Sizeof

\[ \text{sizeof}(\text{type-id} [, \text{tagfield-value}] ... ) \]

The \text{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

\[ \text{integer-var} := \text{argc} \]

The \text{argc} function returns the number of arguments passed to the program. The value of the function is 1 or greater.

Clock

\[ \text{integer-var} := \text{clock} \]

The \text{clock} function returns the milliseconds of processor time used by the current process.

Bitand

\[ \text{integer-var} := \text{bitand}(\text{integer-expr}, \text{integer-expr}) \]

The \text{bitand} function returns the bit-wise \textit{and} of the two integer valued operands.

Bitor

\[ \text{integer-var} := \text{bitor}(\text{integer-expr}, \text{integer-expr}) \]

The \text{bitor} function returns the bit-wise \textit{or} of the two integer valued operands.

Bitxor

\[ \text{integer-var} := \text{bitxor}(\text{integer-expr}, \text{integer-expr}) \]

The \text{bitxor} function returns the bit-wise \textit{exclusive or} of the two integer valued operands.

Bitnot

\[ \text{integer-var} := \text{bitnot}(\text{integer-expr}) \]
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 (coln) is reached.

```
program CountLines(input, output);
type
  string80 = packed array[1..80] of char;
var
  Line : string80;
begin
  i := 0;
  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 speci-
fied as the operand of a READ or a WRITE to a text file.

Reading an enumeration value interprets the programmer–defined name, pre-
ceded 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 de-
limiting 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
#endif
#ifdef LANGUAGE_C
  typedef struct {
    int high, low;
  } pair;
#endif
```

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.

<table>
<thead>
<tr>
<th>Pascal Specification</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal string length</td>
<td>288</td>
</tr>
<tr>
<td>Procedure nesting levels</td>
<td>20</td>
</tr>
<tr>
<td>Set size</td>
<td>481 – 512*</td>
</tr>
<tr>
<td>Significant characters</td>
<td>32</td>
</tr>
</tbody>
</table>

* Depends on actual limits of data item.
Appendix C
Byte Ordering

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.

![Diagram of Word and Halfword byte ordering]

*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.

![Diagram of Word and Halfword]

*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, 1–32
  examples, 1–33
  options, 1–34—1–37
  purpose, 1–32
argv, Pascal predefined function, B–12
argc, 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

Languages Programmer's Guide
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
-ncount 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
unicode object library
  building, 4–24
  using, 4–24
extern (C storage class), 2–6
external directives, in Pascal, B–7
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
–jmove link editor option, description, 1–17
–jmove 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
–nogcount link editor option, 1–17
–ncpp option, 1–9
   See also –cpp option
–nojmove 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

Languages Programmer’s Guide
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
–pf file 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
p1 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
printreg 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
Index

-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
stedi 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
stapi 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
unicode 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

Languages Programmer's Guide
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.
MIPS Computer Systems
928 Arques Avenue
Sunnyvale, CA 94086-9756