MIPSpro™ Compiling, Debugging, and Performance Tuning Guide

About This Guide

This guide describes the MIPSpro™ compiler system, the debugger, and other programming tools and interfaces. It also explains ways to improve program performance.

The IRIX compiler system produces either 64−bit object code or 32−bit object code. This guide describes the MIPSpro compilers that produce 64−bit objects. For information about compilers that produce 32−bit objects and how to debug them, refer to the *IRIX System Programming Guide* and *dbx User's Guide*, respectively.

What This Guide Contains

This guide contains the following chapters:

- **Chapter 1, "About the MIPSpro Compiler System,"** gives an overview of the MIPSpro compiler system.
- **Chapter 2, "Using the MIPSpro Compiler System,"** describes the components of the MIPSpro compiler system.
- **Chapter 3, "Using Dynamic Shared Objects,"** explains how to build and use dynamic shared objects.
- **Chapter 4, "Using the Debugger,"** covers how to use the source level debugger, *dbx*, to debug your programs.
- **Chapter 5, "Using the Performance Tools,"** describes how to use the profiling tool, *prof*, and how to obtain basic block counts with *pixie*.
- **Chapter 6, "Optimizing Program Performance,"** explains how to reduce program execution time by using optimization options and techniques.
- **Chapter 7, "Coding for 64−Bit Programs"** describes how to write or update code that is portable to 64−bit systems.

Some information about the IRIX operating system is covered in the *Indigo Magic Desktop Integration Guide*, namely fonts and internationalization.

What You Should Know Before Reading This Guide

This guide is for anyone who wants to program effectively using the MIPSpro compilers. We assume you are familiar with the IRIX (or UNIX®) operating system and a programming language such as C or Fortran. This guide does not explain how to write and compile programs.

This guide does not cover the differences between 64−bit compilation and 32−bit compilation modes. Refer to the *Application Porting and Transition Guide* for detailed information about the differences between these modes, language implementation differences, source code porting, compilation issues, and run−time execution.

Suggestions for Further Reading

This *Compiling, Debugging, and Performance Tuning Guide* is part of the IRIS Developer Option (IDO), which provides you with the software and documentation that you need to write applications for Silicon Graphics platforms. **Table In−1** lists the IDO online and printed manuals.

Table 0−1 IDO Topics and Manuals

Topic	Manual
IRIX Operating System	IRIX System Programming Guide IRIX Network Programming Guide

ANSI C and Compilation	Compiling, Debugging, and Performance Tuning Guide (MIPSpro compilers and debugger) C Language Reference Manual
Desktop Environment	Indigo Magic Desktop Integration Guide IRIS WorkSpace Integration Guide
Graphics	OpenGL Programming Guide OpenGL Porting Guide
X Window System	XLib Programming Manual (O'Reilly, Vol. 1) X Toolkit Intrinsics Programming Manual (O'Reilly, Vol. 4)
OSF/Motif	OSF/Motif Programmer's Guide OSF/Motif Reference Manual OSF/Motif Style Guide IRIS IM Programming Notes

To read the online manuals after installing them, type **`insight`** or double–click the InSight icon. It's easy to print sections and chapters of the online manuals from InSight. You can also order printed manuals from Silicon Graphics by calling SGI Direct at 1–800–800–SGI1 (800–7441). Outside the U.S. and Canada, contact your local sales office or distributor.

In addition to the IDO, Silicon Graphics offers software options to assist in your software development. The compiler options include *Fortran77*, *Power Fortran*, and *C++*. *CASEVision/Workshop* provides the WorkShop toolset: the Debugger, Static Analyzer, Performance Analyzer, Tester, and Build Manager.

You may also want to learn more about standard UNIX and ANSI C topics. For this information, consult a computer bookstore or manuals such as:

- AT&T. *UNIX System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools*. Englewood Cliffs, NJ: Prentice Hall, 1990.
- Levine, Mason, and Brown. *lex & yacc*. Sebastopol. CA: O'Reilly & Associates, Inc., 1992.
- Oram and Talbott. *Managing Projects with make*. Sebastopol. CA: O'Reilly & Associates, Inc., 1991.
- American National Standards Institute, Inc. *American National Standard, Programming Language—C, ANSI C Standard*. ANSI X3.159–1989.
- International Standard ISO/IEC. *Programming languagues—C, 9899*. 1990(E).

Conventions Used in This Guide

This guide uses these conventions and symbols:

`Courier`	In text, the Courier font represents function names, file names, and keywords. It is also used for command syntax, output, and program listings.
`bold`	Boldface is used along with Courier font to represent user input.
italics	Words in italics represent characters or numerical values that you define. Replace the abbreviation with the defined value. Also, italics are used for manual titles, manual page names, and commands. The section number, in parentheses, follows the name.
_ _	A double underline appears as __ in text.
[ ]	Brackets enclose optional items.
{ }	Braces enclose two or more items; you must specify at least one of the items.
|	The OR symbol separates two or more optional items.
...	A horizontal ellipsis in a syntax statement indicates that the preceding optional items can appear more than once in succession.
( )	Parentheses enclose entities and must be typed.

The following two examples illustrate the syntax conventions:

```
DIMENSION a(d) [,a(d)] ...
```

indicates that the Fortran keyword DIMENSION must be typed as shown, that the user–defined entity *a(d)* is required, and that one or more of *a(d)* can be specified. The parentheses ( ) enclosing *d* are required.

```
{STATIC | AUTOMATIC} v [,v] ...
```

indicates that either the STATIC or AUTOMATIC keyword must be typed as shown, that the user−defined entity *v* is required, and that one or more *v* items can be specified.

Chapter 1

About the MIPSpro Compiler System

The IRIX compiler system produces either 64−bit objects or 32−bit objects. This guide describes the MIPSpro compilers that produce 64−bit objects. For information about producing 32−bit objects, refer to the *IRIX System Programming Guide*.

The MIPSpro complier system components enable you to create executable programs using such languages as C, C++, and Fortran 77. **Table 1−1** summarizes the MIPSpro compiler system components and the task each performs.

Table 1−1 Compiler System Functional Components

Tool	Task	Examples
Text editor	Write and edit programs	*vi*, *jot, emacs*
Compiler driver	Compile, link, and load programs	*cc*, *CC, f77*
Object file analyzer	Analyze object files	*dwarfdump, elfump,* *file*, *nm*, *size, dis*
Profiler	Analyze program performance	*prof*, *pixie*
Archiver	Produce object−file libraries	*ar*
Linker	Link object files	*ld*
Runtime loader	Link Dynamic Shared Objects at runtime	
Debugger	Debug programs	*dbx*

A single program called a compiler driver (such as *cc*, *CC*, or *f77*) invokes the following major components of the compiler system (refer to **Figure 1−1**).

- Macro preprocessor (*cpp*)
- Parallel analyzer (*pca, mpc, fef77p*)
- Scalar optimizer (*copt*)
- Compiler frontend (*fec, fecc*, *fef77*, *fef*77p)
- Compiler back end
- Linker (*ld*)

You can invoke a compiler driver with various options (described later in this chapter) and with one or more source files as arguments. All specified source files are automatically sent to the macro preprocessor.

Although the macro preprocessor was originally designed for C programs, it is now run by default as part of most compilations. To prevent running the preprocessor, use the *-nocpp* option on the driver command line. For C and C++ compilations, preprocessing and front−end compilation is done by *fec* and *fecc*. For all other compilations, preprocessing is done by invocation of a separate executable named *cpp*.

If available, the parallel analyzers *pca* and *fef77p* produce parallelized source code from standard source code. *mpc* takes the output from *pca* and produces parallel C code (part of Power C). The result takes advantage of multiple CPUs (when present) to achieve higher computation rates. *pca* and *mpc* are part of Power C. Power Fortran automatically uses the parallel Fortran compiler, *fef77p*, to produce parallel code. For more information about these packages and how to obtain them, contact your dealer/sales representative.

The compiler front ends (*fec*, *fecc*, *fef77*) translate the source code into an intermediate tree representation. The compiler back end (*be*) translates the intermediate code into object code. The language compilers share the same back end (*be*), which combines optimization and code generation in one phase. (For more information about optimization, see **Chapter 6, "Optimizing Program Performance."**).

The linker *ld* combines several object files into one, performs relocation, and resolves external symbols. The driver automatically runs *ld* unless you specify the *-c* option to skip the linking step.

When you compile or link programs, by default, the compiler searches */usr/lib*, */lib*, and */usr/local/lib*. Certain default libraries are automatically linked. Drivers and their respective libraries are listed in **Table**

1–2 .

Table 1–2 Compilers and Default Libraries

Compiler	Default LIbraies
cc	*libc.so*
CC	*libC.so*, *libc.so*
f77	*libftn.so*, *libc.so*, *libm.so*

To see the various utilities a program passes through during compilation, invoke the appropriate driver with the *-show* option.

Figure 1.1 shows compilation flow from source file (to executable file (a.out).

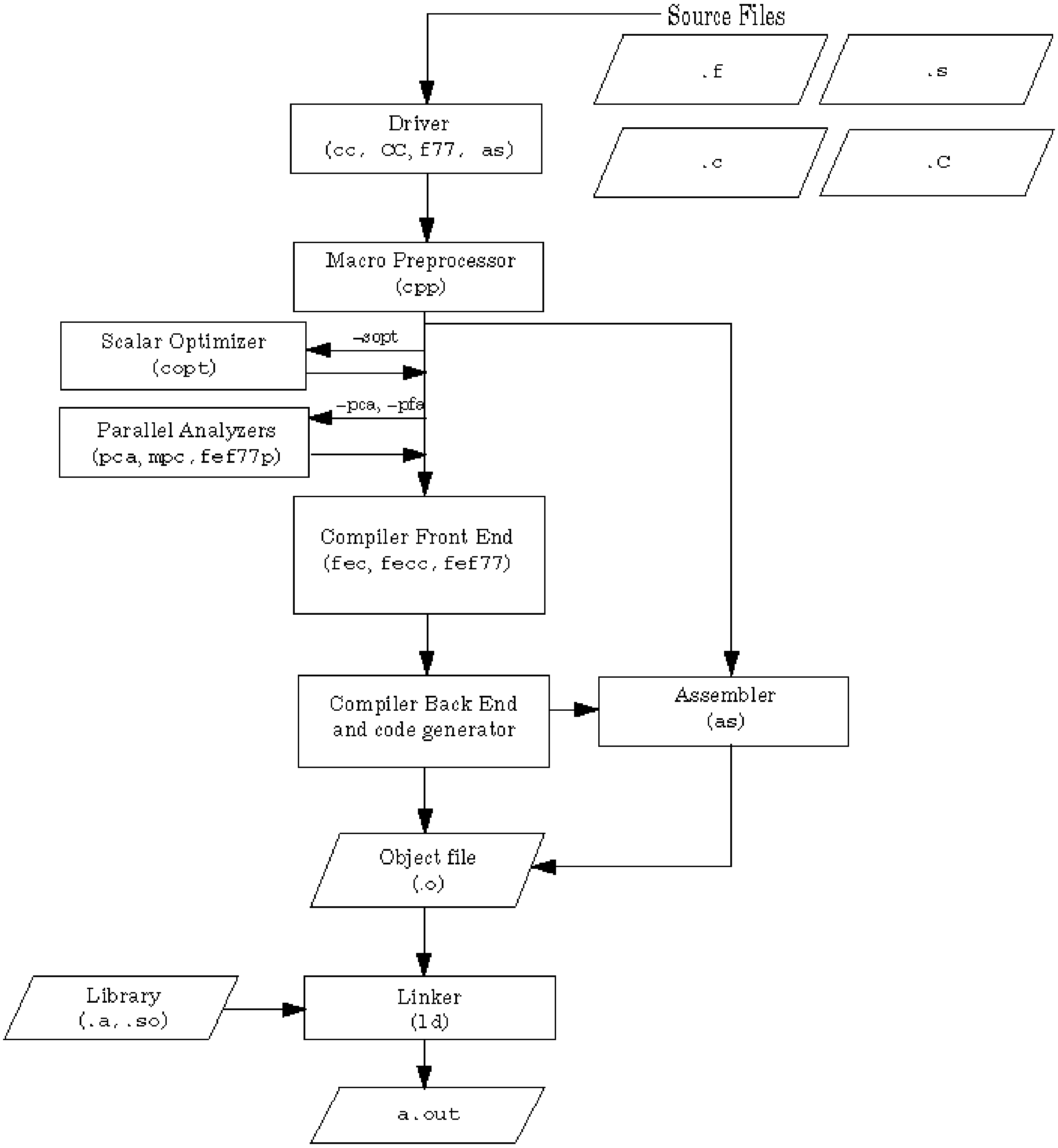

Figure 1–1 Compiler System Flowchart

Chapter 2

Using the MIPSpro Compiler System

This chapter provides information about the MIPSpro compiler system, and describes the object file format and dynamic linking. Specifically, this chapter covers the topics listed below:

- **"Selecting a Compiler"** explains how to specify a 64–bit or 32–bit compilation mode.
- **"Object File Format and Dynamic Linking"** discusses object files including executable and linking format, dynamic shared objects, and position–independent code.
- **"Source File Considerations"** explains source file naming conventions and the procedure for including header files.
- **"Compiler Drivers"** lists and explains general compiler–driver options.
- **"Linking"** explains how to link programs manually (using *ld* or a compiler) and how to compile multilanguage programs. It also covers Dynamic Shared Objects and how to link them into a program.
- **"Debugging"** describes the compiler–driver options for debugging.
- **"Getting Information About Object Files"** provides information on how to use the object file tools to analyze object files.
- **"Using the Archiver to Create Libraries"** explains how to use the archiver, *ar*.

Selecting a Compiler

You can select a compilation mode to compile either 64–bit objects or 32–bit objects by using command–line options and/or setting an environment variable. When determining the compilation mode to use, the system first checks command–line options. Next, it checks the environment variable.

Using Command–Line Options

If you specify the *-32* option to the compiler, the system compiles the source files to 32–bit objects. If you specify the *-64* option on the command line, the system compiles the source files to 64–bit objects.

Setting an Environment Variable

You can set an environment variable to specify the compilation mode to use. To set up your environment for 32–bit compilation, type:

```
setenv SGI_ABI –32
```

To set up your environment for 64–bit compilation, type:

```
setenv SGI_ABI –64
```

If you don't specify *-32* or *-64* on the command line, or don't set an environment variable, the system determines the compilation mode to use.

Object File Format and Dynamic Linking

This section describes how the compiler system

- uses **"Executable and Linking Format"** (ELF) for object files
- uses shared libraries called **"Dynamic Shared Objects"** (DSOs)
- creates **"Position–Independent Code"** (PIC), by default, to support dynamic linking

Executable and Linking Format

The compiler system produces ELF object files. ELF is the format specified by the System V Release 4 Applications Binary Interface (the SVR4 ABI). ELF provides support for Dynamic Shared Objects, described below.

Types of ELF object files are as follows:

- Relocatable files contain code and data in a format suitable for linking with other object files to make a shared object or executable.
- Dynamic Shared Objects contain code and data suitable for *dynamic linking*. Relocatable files may be linked with DSOs to create a dynamic executable. At run time, the run−time linker combines the executable and DSOs to produce a process image.
- Executable files are programs ready for execution. They may or may not be dynamically linked.

COFF executables continue to run on new releases of IRIX, but the current compiler system has no facility for creating or linking COFF executables. COFF and ELF object files may not be linked together. To take advantage of new IRIX features, you must recompile your code.

You can use this version of the compiler system to construct ABI−compliant executables that run on any operating system supporting the MIPS ABI. Be careful to avoid referencing symbols that are not defined as part of the MIPS ABI specification. For more information, see the *System V Applications Binary Interface—Revised Edition* and the *System V ABI MIPS Processor Supplement*.

Dynamic Shared Objects

IRIX uses shared objects called *Dynamic Shared Objects*, or *DSOs*. The object code of a DSO is *position−independent code* (PIC), which can be mapped into the virtual address space of several different processes at once. DSOs are loaded at run time instead of at linking time, by the run−time loader, *rld*. As is true for static shared libraries, the code for DSOs is not included in executable files; thus, executables built with DSOs are smaller than those built with non−shared libraries, and multiple programs may use the same DSO at the same time. For more information on DSOs, see **Chapter 3, "Using Dynamic Shared Objects."**

Static shared libraries are only supported under this release for the purposes of running old (COFF) binaries. The current compiler system has no facilities for generating static shared libraries.

Position−Independent Code

Dynamic linking requires that all object code used in the executable be position−independent code. For source files in high−level languages, you just need to recompile to produce PIC. Assembly language files must be modified to produce PIC; see the *MIPSpro Assembly Language Programmer's Guide* for details.

Position−independent code satisfies references indirectly by using a *global offset table* (GOT), which allows code to be relocated simply by updating the GOT. Each executable and each DSO has its own GOT.

The compiler system produces PIC by default when compiling higher−level language files. All of the standard libraries are provided as DSOs, and therefore contain PIC code; if you compile a program into non−PIC, you will be unable to use those DSOs. One of the few reasons to compile non−PIC is to build a device driver, which doesn't rely on standard libraries; in this case, you should use the *-non_shared* option to the compiler to negate the default option, *-KPIC*. For convenience, the C library and math library are provided in non−shared format as well as in DSO format (although the non−shared versions are not installed by default). You can link these libraries *-non_shared* with other non−PIC files.

Source File Considerations

This section describes conventions for naming source files and including header files. Topics covered include:

- **"Source File Naming Conventions"**
- **"Header Files"**

Source File Naming Conventions

Each compiler driver recognizes the type of an input file by the suffix assigned to the filename. **Table 2−1** describes the possible filename suffixes.

Table 2−1 Driver Input File Suffixes

Suffix	Description
.s	Assembly source

.i	Preprocessed source code in the language of the processing driver
.c	C source
.C, .cxx, .cc, .c++	C++ source
.f, .F, .for, .FOR	Fortran 77 source
.o	Object file
.a	Object library archive
.so	Dynamic shared object library

The following example compiles preprocessed source code:

```
f77 −c tickle.i
```

The Fortran compiler, *f77,* assumes the file *tickle.i* contains Fortran statements (because the Fortran driver is specified). *f77* also assumes the file has already been preprocessed (because the suffix is *.i*), and therefore does not invoke the preprocessor.

Header Files

Header files, also called *include* files, contain information about the libraries with which they're associated. They define such things as data types, data structures, symbolic constants, and prototypes for functions exported by the library. To use those definitions without having to type them into each of your source files, you can use the *#include* directive to tell the macro preprocessor to include the complete text of the given header file in the current source file. When you include header files in your source files you can specify such definitions conveniently and consistently in each source file that uses any of the library routines.

By convention, header filenames have a *.h* suffix. Each programming language handles these files the same way, via the macro preprocessor. For example, the *stdio.h* header file describes, among other things, the data types of the parameters required by *printf()*.

For detailed information about standard header files and libraries, see the International Standard ISO/IEC. *Programming languages—C, 9899*. 1990. Also see **"Using Typedefs"** for information about the *inttypes.h* header file.

Specifying a Header File

The *#include* directive tells the preprocessor to replace the *#include* line with the text of the indicated header file. The usual way to specify a header file is with the line

```
#include <filename>
```

where *filename* is the name of the header file to be included. The angle brackets (< >) surrounding the filename tell the macro preprocessor to search for the specified file only in directories specified by command−line options and in the default header−file directory (*/usr/include*).

Another specification format exists, in which the filename is given between double quotation marks. In this case, the macro preprocessor searches for the specified header file in the current directory first, then (if it doesn't find the requested file) goes on and searches in the other directories as in the angle−bracket specification.

Creating a Header File for Multiple Languages

A single header file can contain definitions for multiple languages; this setup allows you to use the same header file for all programs that use a given library, no matter what language those programs are in.

To set up a shareable header file, create a *.h* file and enter the definitions for the various languages as follows:

```
#ifdef _LANGUAGE_C

C definitions

#endif
```

```
#ifdef _LANGUAGE_C_PLUS_PLUS

C++ definitions

#endif

#ifdef _LANGUAGE_FORTRAN

Fortran definitions

#endif
```

Note: You must specify _LANGUAGE_ before the language name and to indicate C++ definitions, you must use _LANGUAGE_C_PLUS_PLUS, not _LANGUAGE_C++.

You can specify the various language definitions in any order.

Compiler Drivers

The driver commands, such as *cc* and *f77* call subsystems that compile, optimize, assemble, and link your programs. This section describes:

- **"Default Behavior for Compiler Drivers"**
- **"General Options for Compiler Drivers"**

Default Behavior for Compiler Drivers

At compilation time, you can select one or more options that affect a variety of program development functions, including debugging, profiling, and optimizing. You can also specify the names assigned to output files. Note that some options have default values that apply if you do not specify them.

When you invoke a compiler driver with source files as arguments, the driver calls other commands that compile your source code into object code. It then optimizes the object code (if requested to do so) and links together the object files, the default libraries, and any other libraries you specify.

Given a source file *foo.c*, the default name for the object file is *foo.o*. The default name for an executable file is *a.out*. The following example compiles source files *foo.c* and *bar.c* with the default options:

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

This example produces two object files (*foo.o* and *bar.o*), then links them with the default C library *libc* to produce an executable called *a.out*.

Note: If you compile a single source directly to an executable, the compiler does not create an object file.

General Options for Compiler Drivers

The command–line options for MIPSpro compiler drivers are listed and explained in **Table 2–2** . The table lists only the most frequently used options, not all available options. See the appropriate compiler reference (manual) page for additional details on these commands.

In addition to the general options in **Table 2–2** , each driver has options that you typically won't use. These options primarily aid compiler development work. For information about nonstandard driver options, consult the appropriate driver reference page.

You can use the compiler system to generate profiled programs that, when executed, provide operational statistics. To perform this procedure, use the *-p* compiler option (for pc sampling information) and the *prof* command (for profiles of basic block counts). Refer to **Chapter 5, "Using the Performance Tools,"** for details.

Table 2–2 General Driver Options

Option	Purpose
-32	Generates a 32–bit object.
-64	Generates a 64–bit object. This is the default.
-ansi	Strict ANSI/ISO C compilation mode. Preprocessing adds only standard predefined symbols to the name space, and standard include files declare only standard symbols.
-c	Prevents the linker from linking your program after code generation. This option forces the driver to produce a *.o* file after the back–end phase, and prevents the driver from producing an executable file.
-C	Used with the *-P* or *-E* option. Prevents the macro preprocessor from stripping comments. Use this option when you suspect the preprocessor is not producing the intended code and you want to examine the code with its comments.
-cckr	K&R/Version7 C compatibility compilation mode. Preprocessing may add more predefined symbols to the name space than in *-ansi* mode. Compilation adheres to the K&R language semantics.
-D*name*[=*def*]	Defines a macro *name* as if you had specified a *#define* in your program. If you do not specify a definition with *=def*, *name* is set to 1.
-E	Runs only the macro preprocessor and sends results to the standard output. To retain comments, use the *-C* option as well. Use *-E* when you suspect the preprocessor is not producing the intended code.
-fullwarn	Checks code and produces additional warnings that are normally suppressed. This option is recommended for all compiles during software development. You can turn off warnings selectively by using the *-woff* option.
-g[*num*]	Produces debugging information. The default is *-g0*: do not produce debugging information.
-G[*num*]	Specifies the maximum size, in bytes, of a data item that is accessed from the global pointer. The default is *-G8*.
-help	Lists the available compiler options.
-Idirname	Adds *dirname* to the list of directories to be searched for specified header files. These directories are always searched before the default directory, */usr/include*.
-KPIC	Generates position–independent code. This is the default and is required for programs linking with dynamic shared objects. Specify *-non_shared* if you don't want to generate PIC code.
-mips1	Generates code using the instruction set of the MIPS R2000/R3000 RISC architecture. This option implies *-32*.
-mips2	Generates code using the MIPS II instruction set (MIPS I + R4000 specific extensions). Note that code compiled with *-mips2* does not run on R2000/R3000–based machines. This option implies *-32*.
-mips3	Generates code using the full MIPS R4000 instruction set, including 64–bit code. This option implies *-64*.
-mips4	Generates code using the MIPS R8000 instruction set. This option implies

	-64.
-nocpp	Suppresses running of the macro preprocessor on the source files prior to processing.
-non_shared	Turns off the default option, *-KPIC*, to produce non−shared code. This code can be linked to only a few standard libraries (such as *libc.a* and *libm.a*) that are provided in non−shared format in the directory */usr/lib/nonshared*. You should use this option only when building device drivers.
-nostdinc	Suppresses searching of */usr/include* for the specified header files.
-o filename	Names the result of the compilation *filename*. If an executable is being generated, it is named *filename* rather than the default name, *a.out*.
-Onum	Specifies optimization options. *-O0* turns off optimizations. *-O1* turns on local optimizations that can be done quickly. *-O*2 (*-O*) turns on global optimizations. This is the default. *-O3* turns on all optimizations. For more information, see **Chapter 6, "Optimizing Program Performance."**
-OPT:options	Controls optimization options. For more information, see **Chapter 6, "Optimizing Program Performance."**
-p	Sets up for profiling by periodically sampling the value of the program counter (only effects the loading).
-P	Runs only the macro preprocessor on the files and puts the result of each file in a *.i* file. Specify both *-P* and *-C* to retain comments.
-S	Similar to *-c*, except that it produces assembly code in a *.s* file instead of object code in a *.o* file. *-S* provides information about the code and comments about such things as software pipelining, the loops it works on, and the results.
-show	Lists compiler phases as they are executed. Use this option to see the default options for each compiler phase along with the options you've specified.
-SWP:options	Controls software pipelining of inner loops. For more information, see **Chapter 6, "Optimizing Program Performance."**
-TARG:options	Controls the target architecture and machine for which code is generated. For more information, see **Chapter 6, "Optimizing Program Performance."**
-TENV:options	Controls the target environment assumed by the compiler. For more information, see **Chapter 6, "Optimizing Program Performance."**
-Uname	Overrides a definition of the macro *name* that you specified with the *-D* option, or that is defined automatically by the driver. Note that this option does not override a macro definition in a source file, only on the command line.
-woff n	Suppresses ANSI/ISO warning message number

	n. Suppress multiple warning numbers by using a comma−separated list *(-woff n1,n2*...), a range of warning numbers by using a hyphen−separated list (*-woff n1−n5*), or any combination thereof. You can suppress all warning messages via *-woff all.*
-xansi	Compilation follows an extended ANSI/ISO C language semantics, which is more lenient in terms of the forms of expressions it allows. Preprocessing combines predefined macros. This is the default C compilation mode.

Note: To use 4.3 BSD extensions in C, compile using *-xansi* or by using the -D__EXTENSIONS__ option on the command line. For example:

```
cc prog.c −ansi −prototypes −fullwarn −lm −D__EXTENSIONS__
```

Linking

The linker, *ld*, combines one or more object files and libraries (in the order specified) into one executable file, performing relocation, external symbol resolutions, and all other required processing. Unless directed otherwise, the linker names the executable file *a.out*.

This section summarizes the functions of the linker. It also covers how to link a program manually (without using a compiler driver) and how to compile multilanguage programs. Refer to the *ld*(1) reference page for complete information on the linker. Specifically, this section describes:

- **"Invoking the Linker Manually"**
- **"Linking Assembly Language Programs"**
- **"Linking Libraries"**
- **"Linking to Previously Built Dynamic Shared Objects"**
- **"Linking Multilanguage Programs"**

Invoking the Linker Manually

Usually the compiler invokes the linker as the final step in compilation (as explained in **"Compiler Drivers"**). If object files exist that were produced by previous compilations, and you want to link them, invoke the linker by using a compiler driver instead of calling *ld* directly. Just pass the object filenames to the compiler driver in place of source filenames. If the original source files are in a single language, simply invoke the associated driver and specify the list of object files. (For information about linking objects derived from several languages, see **"Linking Multilanguage Programs."**)

A few command−line options to *ld*, such as *-p*, have different meanings when used as command−line options to *cc*. To pass such options to *ld* through an invocation of a compiler driver, use the *-Wl* option to the driver (see the reference page for details).

Typically, the compiler driver invokes *ld* as necessary. Circumstances exist under which you may need to invoke *ld* directly, such as when you're building a shared object or doing special linking not supported by compiler drivers (such as building an embedded system). To build C++ shared objects, use the CC driver.

Linker Syntax

A summary of *ld* syntax follows.

```
ld options   object1 [object2...objectn]
```

options	One or more of the options listed in **Table 2−3** .
object	Specifies the name of the object file to be linked.

Table 2−3 contains only a partial list of linker options. Many options that apply only to creating shared objects are discussed in **Chapter 3, "Using Dynamic Shared Objects."** For complete information on options and libraries that affect linker processing, refer to the *ld*(1) reference page.

Table 2−3 Linker Options

Option	Purpose

-32	Links 32–bit programs and DSOs.
-64	Links 64–bit programs and DSOs.
-lname	Specifies the name of a library, where *lname* is the library name. The linker searches for *lname.so* (and then *lname.a*) first in any directories specified by *-L dirname* options, and then in the standard directories: */usr/lib*, */lib*, and */usr/local/lib*.
-L dirname	Adds *dirname* to the list of directories to be searched for (as well as libraries searched for) as specified by subsequent *-lname* options.
-m	Produces a linker memory map, listing input and output sections of the code, in System V format.
-M	Produces a link map in BSD format, listing the names of files to be loaded.
-nostdlib	This option must be accompanied by the *-L dirname* option. If the linker does not find the library in *dirname* list, then it does not search any of the standard library directories.
-o filename	Specifies a name for your executable. If you do not specify *filename*, the linker names the executable *a.out*.
-s	Strips debugging information from the program object, reducing its size. This option is useful for linking routines that are frequently linked into other program objects, but may hamper debugging.
-v	Produces verbose linker output providing information about various linker passes.
-ysymname	Reports all references to, and definitions of, the symbol *symname*. Useful for locating references to undefined symbols.

Linker Example

The following command tells the linker to search for the DSO *libcurses.so* in the directory */usr/lib*. If it does not find that DSO, the linker then looks for *libcurses.a* in */lib*.

```
ld foiled.o again.o –lcurses
```

If the linker doesn't find an appropriate library, it looks in */usr/local/lib* for *libcurses.a*. (Note that the linker does not look for DSOs in */usr/local/lib*, so don't put shared objects there.) If found in any of these places, the DSO or library is linked with the objects *foiled.o* and *again.o*; otherwise an error is generated.

Linking Assembly Language Programs

The assembler driver (*as*) does not run the linker. To link a program written in assembly language, use one of these procedures:

- Assemble and link 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.
- Assemble the file using *as*; then link the resulting object file with the *ld* command.

Linking Libraries

The linker processes its arguments from left to right as they appear on the command line. Arguments to *ld* can be DSOs, object files, or libraries.

When *ld* reads a DSO, it adds all the symbols from that DSO to a cumulative symbol table. If it encounters a symbol that's already in the symbol table, it does not change the symbol table entry. If you define the same symbol in more than one DSO, only the first definition is used.

When *ld* reads an archive, usually denoted by a filename ending in *.a*, it uses only the object files from that archive that can resolve currently unresolved symbol references. (When a symbol is referred to but not defined in any of the object files that have been loaded so far, it's called unresolved.) Once a library has been searched in this way, it is never searched again. Therefore, libraries should come after object files on the command line in order to resolve as many references as possible. Note that if a symbol is already in the cumulative symbol table from having been encountered in a DSO, its definition in any subsequent archive or DSO is ignored.

Specifying Libraries and DSOs

You can specify libraries and DSOs either by explicitly stating a pathname or by use of the library search rules. To specify a library or DSO by path, simply include that path on the command line (relative to the current directory, or else absolute):

```
ld myprog.o /usr/lib/libc.so.1 mylib.so
```

libc.so.1 is the name of the standard C DSO, replacing the older *libc.a*. Similarly, *libX11.so.1* is the X11 DSO. Most other DSOs are simply named *name.so*, without a *.1* extension.To use the linker's library search rules, specify the library with the *-lname* option:

```
ld myprog.o −lmylib
```

When the *-lmylib* argument is processed, *ld* searches for a file called *libmylib.so*. If it can't find *libmylib.so* in a given directory, it tries to find *libmylib.a* there; if it can't find that either, it moves on to the next directory in its search order. The default search order is to look first in */usr/lib*, then in */lib*.

If *ld* is invoked from one of the compiler drivers, all *-L* and *-nostdlib* options are moved up on the command line so that they appear before any *-lname* option. For example, consider the command:

```
cc file1.o −lm −L mydir
```

This command invokes, at the linking stage of compilation, the following:

```
ld −L mydir file1.o −lm
```

Note: There are three different kinds of files that contain object code files: non−shared libraries, PIC archives, and DSOs. Non−shared libraries are the old−style library, built using *ar* from *.o* files that were compiled with *-non_shared*. These archives must also be linked *-non_shared*. PIC archives are the default, built using *ar* from *.o* files compiled with *-KPIC* (the default option); they can be linked with other PIC files. DSOs are built from PIC *.o* files by using *ld -shared*; see **Chapter 3** for details.

If the linker tells you that a reference to a certain function is unresolved, check that function's reference page to find out which library the function is in. If it isn't in one of the standard libraries (which *ld* links in by default), you may need to specify the appropriate library on the command line. For an alternative method of finding out where a function is defined, see **"Finding an Unresolved Symbol With ld."**

Note: Simply including the header file associated with a library routine is not enough; you also must specify the library itself when linking (unless it's a standard library). No automatic connection exists between header files and libraries; header files only give prototypes for library routines, not the library code itself.

Examples of Linking DSOs

To link a sample program *foo.c* with the math DSO, *libm.so*, enter:

```
cc foo.c −lm
```

To specify the appropriate DSOs for a graphics program *foogl.c*, enter:

```
cc foogl.c −lgl −lX11
```

Linking to Previously Built Dynamic Shared Objects

This section describes how to link your source files with previously built DSOs; for more information about how to build your own DSOs, see **Chapter 3, "Using Dynamic Shared Objects."**

To build an executable that uses a DSO, call a compiler driver just as you would for a non−shared library. For instance,

```
cc needle.c −lthread
```

This command links the resulting object file (*needle.o*) with the previously built DSO *libthread.so* (and the standard C DSO, *libc.so.1*), if available. If no *libthread.so* exists, but a PIC archive named *libthread.a* exists, that archive is used with *libc.so.1*, so you still get dynamic (run time) linking. Note that even *.a* libraries now contain position−independent code by default, though it is also possible to build non−shared *.a* libraries that do not contain PIC.

Linking Multilanguage Programs

The source language of the main program may differ from that of a subprogram. In this case, you can link multilanguage programs.

Follow the steps below to link multilanguage programs. (Refer to **Figure 2−1** for an illustration of the process.)

1. Compile object files from the source files of each language separately by using the *-c* option.

   For example, if the source consists of a Fortran main program (*main.f*) and two files of C functions (*more.c* and *rest.c*), use the commands:

   ```
   cc −c more.c rest.c
   f77 −c main.f
   ```

   These commands produce the object files *main.o, more.o,* and *rest.o*.

2. Use the compiler associated with the language of the main program to link the objects:

   ```
   f77 main.o more.o rest.o
   ```

   The compiler drivers supply the default set of libraries necessary to produce an executable from the source of the associated language. However, when producing executables from source code in several languages, you may need to specify the default libraries explicitly for one or more of the languages used. For instructions on specifying libraries, see **"Linking Libraries."**

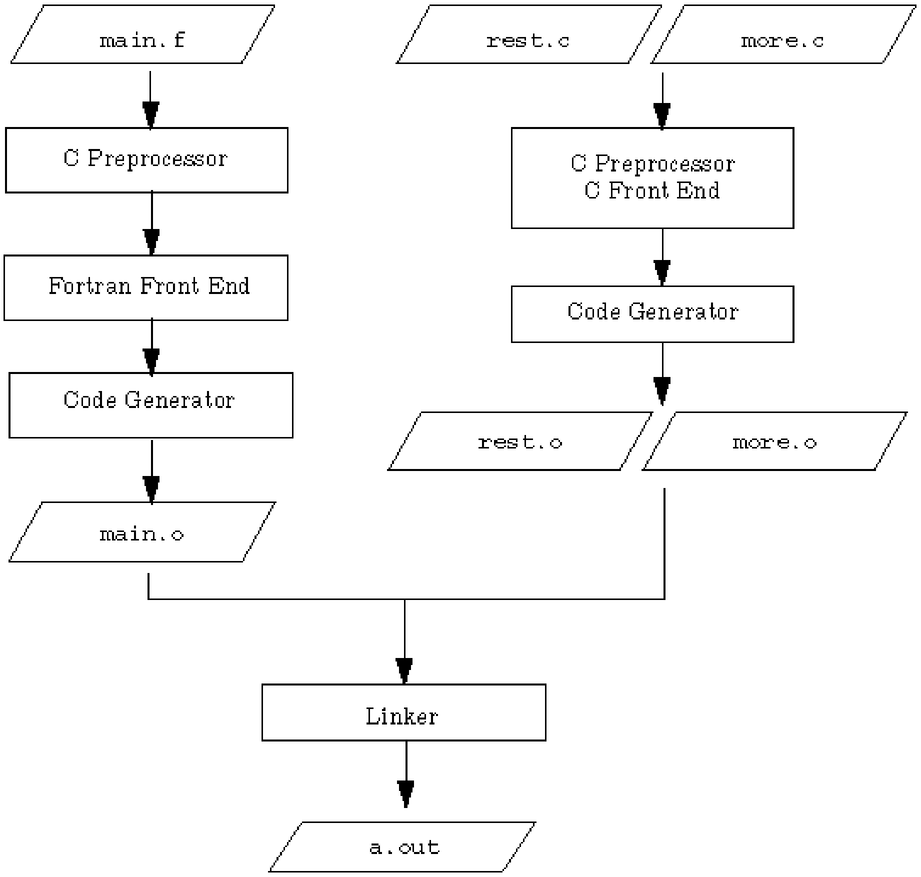

Figure 2−1 Compilation Control Flow for Multilanguage Programs

Use caution when passing pointers and longs between languages as some languages use different type sizes and structures for data types.For specific details about compiling multilanguage programs, refer to the programming guides for the appropriate languages.

Finding an Unresolved Symbol With *ld*

You can use *ld* to locate unresolved symbols. For example, suppose you're compiling a program, and *ld* tells you that you're using an unresolved symbol. However, you don't know where the unresolved symbol is referenced.

To find the unresolved symbol, enter:

```
ld −ysymbol file1... filen
```

You can also enter:

```
cc prog.o −Wl,−ysymbol
```

The output lists the source file that references *symbol*.

Debugging

The compiler system provides a debugging tool, *dbx*, described fully in **Chapter 4, "Using the Debugger."** Before using *dbx*, specify the *-g* driver option (see **Table 2−4**) to produce executables containing information that the debugger can use.

Table 2−4 Driver Options for Debugging

Option	Purpose
-g0	Produces a program object with a minimum of source−level debugging information. This is the default. Reduces the size of the program object but allows optimizations. Use this option with the *-O* option after you finish debugging.
-g, *-g2*	Produces additional debugging information for full symbolic debugging. This option overrides the optimization options (*-Onum*).
-g3	Produces additional debugging information for full symbolic debugging of fully optimized code. This option makes the debugger less accurate. You can use *-g3* with an optimization option (*-Onum*).

Getting Information About Object Files

The following tools provide information on object files:

- *dis* disassembles an object file into machine instructions.
- *dwarfdump* lists headers, tables, and other selected parts of a DWARF−format object file or archive file.
- *elfdump* lists the contents (including the symbol table and header information) of an ELF−format object file.
- *file* provides descriptive information on the properties of a file.
- *nm* lists symbol table information.
- *size* prints the size of each section of an object file (some such sections are named *text*, *data*, and *sbss*).
- *strip* removes symbol table and relocation bits.

Disassembling Object Files with *dis*

The *dis* tool disassembles object files into machine instructions. You can disassemble an object, archive library, or executable file.

dis Syntax

```
dis options   filename1   [filename2...filenamen]
```

options	One or more of the options listed in **Table 2–5** .
filename	Specifies the name of one or more files to disassemble.

dis Options

Table 2–5 lists *dis* options. For more information, see the *dis*(1) reference page.

Table 2–5 *dis* Options

Option	Description
-b begin_address	Starts disassembly at *begin_address*. You can specify the address as decimal, octal (with a leading *0*), or hexadecimal (with a leading *0x*).
-d section	Disassembles the named *section* as data, and prints the offset of the data from the beginning of the section.
-D section	Disassembles the named *section* as data, and prints the address of the data.
-e end_address	Stops disassembly at *end_address*. You can specify the address as decimal, octal (with a leading *0*), or hexadecimal (with a leading *0x*).
-F function	Disassembles the named *function* in each object file you specify on the command line.
-h	Substitutes the hardware register names for the software register names in the output.
-H	Removes the leading source line, and leaves the hex value and the instructions.
-i	Removes the leading source line and hexadecimal value of disassembly, and leaves only the instructions.
-I directory	Uses *directory* to help locate source code.
-l string	Disassembles the archive file specified by *string*.
-L	Looks up source labels for subsequent printing.
-o	Prints numbers in octal. The default is hexadecimal.
-s	Performs symbolic disassembly where possible. Prints (using C syntax) symbol names on the line following the instruction. Displays source code mixed with assembly code
-t section	Disassembles the named *section* as text.
-T	Specifies the trace flag for debugging the disassembler.
-V	Prints (on *stderr*) the version number of the disassembler being executed.
-w	Prints source code to the right of assembly code (produces wide output). Use this option with the *-s* option.
-x	Prints offsets in hexadecimal (the default).

Listing Parts of DWARF Object Files With *dwarfdump*

The *dwarfdump* tool provides debugging information from selected parts of DWARF symbolic information in an Elf object file. For more information on DWARF, see files in the *4Dgifts* directory.

dwarfdump Syntax

```
dwarfdump options   filename
```

options	One or more of the options listed in **Table 2–6** .
filename	Specifies the name of the object file whose contents are to be dumped.

dwarfdump Options

Table 2–6 lists *dwarfdump* options. For more information, see the *dwarfdump*(1) reference page.

Table 2–6 *dwarfdump* Options

Option	Dumps
-a	All sections.
-b	The *.debug_abbrev* section.

-c	The *.debug_loc* section.
-d	Uses dense mode. Prints die information of the *.debug_info* section. Does not imply the *-i* option.
-e	Uses ellipsis mode. Uses the short names for DW_TAG_* and DW_ATTR_* in the output for the *.debug_info* section.
-f	The *.debug_frame* section.
-i	The *.debug_info* section.
-l	The *.debug_line* section.
-m	The *.debug_macinfo* section.
-o	The *.rel.debug_** sections.
-p	The *.debug_pubnames* section.
-r	The *.debug_aranges* section.
-s	The *.debug_string* section.
-ta	The *.debug_static_funcs* and *.debug_static_vars* sections (same as *-tfv*).
-tf	The *.debug_static_funcs* section.
-tv	The *.debug_static_vars* section.
-u file	Sections to the named *file*.
-v	Prints detailed information (verbose mode).
-w	The *.debug_weaknames* section.
-y	The *.debug_types* section.

Listing Parts of ELF Object Files and Libraries with *elfdump*

The *elfdump* tool lists headers, tables, and other selected parts of an ELF−format object file or archive file.

elfdump Syntax

```
elfdump options   filename1   [filename2...filenamen]
```

options	One or more of the options listed in **Table 2−7** .
filename	Specifies the name of one or more object files whose contents are to be dumped.

elfump Options

Table 2−7 lists *elfdump* options. For more information, see the *elfdump*(1) reference page.

Table 2−7 *elfdump* Options

Option	Dumps
-a	Archive header of each member of the archive.
-A	Beginning address of a section.
-c	String table.
-C	Decoded C++ symbol names.
-cmt	The *.comment* sections.
-cnt	The *.content* sections.
-d	Range of sections.
-Dc	Conflict list in Dynamic Shared Objects.
-Dg	Global Offset Table in Dynamic Shared Objects.
-Dinfo	The *.MIPS.dclass* section.
-Dinst	The *.MIPS.inst* section.
-Dl	Library list in Dynamic Shared Objects.
-Dsym	The *.MIPS.sym* section.
-Dsymlib	Symbol library table (*.MIPS.symlib*).
-Dt	String table entries of the dynamic symbol table in Dynamic Shared Objects.
-e	Events sections.
-f	Each file header.
-F	Literal tables (*.lit4* and *.lit8* sections).
-g	Archive symbol table.
-G	The global pointer table information.
-h	All section headers in the file.

-hash	Hash table entries.
-i	The *.interp* section, which lists the path name of the program interpreter.
-info	Prints whether the object is marked quickstart, or is corded and is marked requickstart.
-l	The *.line* section.
-L	Dynamic linking information in Dynamic Shared Objects.
-n	The specified section (such as .MIPS.content, .dynamic, .got, MIPS.sym, .liblist, .conflict, .reginfo, and so forth).
-o	Each program execution header.
-op	Options section.
-p	Suppresses the printing of headings.
-r	Relocation information.
-R	Register information.
-reg	The *.reginfo* section.
-rpt	The run−time procedure table.
-s	Section contents.
-svr4	Information in SVR4−style format.
-t	Symbol table entries.
-T	Symbol table range.
-u	Updates a COFF file to an ELF file.
-v	Prints detailed information (verbose mode).
-V	Version information only.

Determining File Type with *file*

The *file* tool lists the properties of program source, text, object, and other files. This tool attempts to identify the contents of files using various heuristics. It is not exact and often erroneously recognizes command files as C programs. For more information, see the *file*(1) reference page.

file Syntax

```
file filename1   [filename2...filenamen]
```

Each *filename* is the name of a file to be examined.

file Example

Information given by *file* is self−explanatory for most kinds of files, but using *file* on object files and executables gives somewhat cryptic output. In this example, "MSB" indicates Most Significant Byte, also called Big−Endian; "relocatable" means the object contains relocation information that allows it to be linked with other objects to form an executable; "executable" means an executable file; and "dynamic lib" indicates a DSO.

```
file test.o a.out /lib/libc.so.1
test.o:         ELF 64-bit MSB relocatable MIPS - version 1
a.out:          ELF 64-bit MSB executable MIPS - version 1
/lib/libc.so.1:   ELF 64-bit MSB dynamic lib MIPS - version 1
```

Listing Symbol Table Information: *nm*

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

nm Syntax

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

options	One or more of the options listed in **Table 2−8** .
filename	Specifies the object files or archive files from which symbol table information is to be extracted. If you do not specify a filename, *nm* assumes the file is named *a.out*.

nm Symbol Table Options

Table 2–8 lists *nm* symbol table options. For more information, see the *nm*(1) reference page.

Table 2–8 Symbol Table *nm* Options

Option	Purpose
-a	Prints debugging information.
-A	Prints the listing in System V format (default).
-b	Prints the value field in octal.
-B	Prints the listing in BSD format.
-C	Prints decoded C++ names.
-d	Prints the value field in decimal (the default for System V output).
-g	Prints globally visible names.
-h	Suppresses printing of headers.
-l	Produces a long listing.
-n	Sorts external symbols by name for System V format. Sorts all symbols by value for BSD format (by name is the BSD default output).
-o	Prints value field in octal (System V output). Prints the filename immediately before each symbol name (BSD output).
-p	Produces easily parsible, terse output similar to the BSD format.
-r	Prepends the name of the object file or archive to each output line.
-T	Truncates characters in exceedingly long symbol names; inserts an asterisk as the last character of the truncated name. This option may make the listing easier to read.
-u	Prints only undefined symbols.
-v	Sorts external symbols by value (default for BSD format).
-V	Prints the version number of *nm*.
-x	Prints the value field in hexadecimal.

Table 2–9 defines the one–character codes shown in an *nm* listing. Refer to the example that follows the table for a sample listing.

Table 2–9 Character Code Meanings

Key	Description
a	Local absolute data
A	External absolute data
b	Local zeroed data
B	External zeroed data
C	Common data
d	Local initialized data
D	External initialized data
E	Small common data
G	External small initialized data
N	Nil storage class (unused external reference)
r	Local read–only data
R	External read–only data
s	Local small zeroed data
S	External small zeroed data
t	Local text
T	External text
U	External undefined data
V	External small undefined data

nm Example of Obtaining a Symbol Table Listing

This example demonstrates how to obtain a symbol table listing. Consider the following program, *tnm.c*:

```
#include <stdio.h>
#include <math.h>
#define LIMIT 12
int unused_item = 14;
double mydata[LIMIT];
```

```
main()
{
      int i;
      for(i = 0; i < LIMIT; i++) {
          mydata[i] = sqrt((double)i);
      }
      return 0;
}
```

Compile the program into an object file by entering:

`cc -c tnm.c`

To obtain symbol table information for the object file *tnm.o* in BSD format, use the *nm -B* command:

```
0000000000 T main
0000000000 B mydata
0000000000 U sqrt
0000000000 D unused_item
00000000 N _bufendtab
```

To obtain symbol table information for the object file *tnm.o* in SVR4 format, use the *nm* command without any options:

```
Symbols from tnm.o:

[Index]   Value     Size    Class    Type         Section    Name

[0]     |        0|        |File    |ref=4       |Text     | tnm.c
[1]     |        0|        |Proc    |end=3 int   |Text     | main
[2]     |      116|        |End     |ref=1       |Text     | main
[3]     |        0|        |End     |ref=0       |Text     | tnm.c
[4]     |        0|        |File    |ref=6       |Text     |
 /usr/include/math.h
[5]     |        0|        |End     |ref=4       |Text     |
 /usr/include/math.h
[6]     |        0|        |Global  |            |Data     | unused_item
[7]     |        0|        |Global  |            |Bss      | mydata
[8]     |        0|        |Proc    |ref=1       |Text     | main
[9]     |        0|        |Proc    |            |Undefined| sqrt
[10]    |        0|        |Global  |            |Undefined| _gp_disp
```

Determining Section Sizes with *size*

The *size* tool prints information about the sections (such as *text*, *rdata*, and *sbss*) of the specified object or archive files. The *elf*(4) reference page describes the format of these sections.

size Syntax

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

options	Specifies the format of the listing (see **Table 2–10**).
filename	Specifies the object or archive files whose properties are to be listed. If you do not specify a filename, the default is *a.out*.

size Options

Table 2–10 lists *size* options. For more information, see the *size*(1) reference page.

Table 2–10 *size* Options

Option	**Action**
-A	Prints data section headers in System V format (default).

-B	Prints output in BSD−style format.
-d	Prints sizes in decimal (default).
-f	Prints data on allocatable sections including the size, permission flags, and the total of the loadable sizes.
-F	Prints data on loadable segments including the name and the total of the section sizes.
-n	Prints nonloadable and nonallocatable section sizes.
-o	Prints sizes in octal.
-svr4	Prints output in SVR4−style format.
-V	Prints the version of size that you are using.
-x	Prints sizes in hexadecimal.

Size Example

An example of the *size* command and the listings produced follows.

```
size a.out
```

```
   Section          Size            Physical               Virtual
                                     Address               Address
         .interp      21           268435856             268435856
   .MIPS.options     104           268435880             268435880
        .dynamic     464           268435984             268435984
        .liblist      20           268436448             268436448
     .MIPS.symlib     30           268436468             268436468
            .msym    240           268436500             268436500
          .dynstr    312           268436744             268436744
          .dynsym    720           268437056             268437056
            .hash    256           268437776             268437776
      .MIPS.stubs     56           268438032             268438032
            .text    460           268438088             268438088
            .init     24           268438548             268438548
            .data     17           268505088             268505088
           .sdata      8           268505108             268505108
             .got    112           268505120             268505120
             .bss     36           268505232             268505232
```

Removing Symbol Table and Relocation Bits with *strip*

The *strip* tool removes symbol table and relocation bits that are attached to the assembler and loader. Use *strip* to save space after you debug a program. The effect of *strip* is the same as that of using the *−s* option to *ld*.

strip Syntax

```
strip options   filename1   [filename2...filenamen]
```

options	One or more of the options listed in **Table 2−11** .
filename	Specifies the name of one or more object files whose contents are to be stripped.

For more information, see the *strip*(1) reference page.

Table 2−11 *strip* Options

Option	Description
-l	Strips line number information, and keeps the symbol table and debugging information.
-o filename	Puts the stripped information in the *filename* that you specify.
-V	Prints the version number of *strip*.
-x	Keeps symbol table information, but may strip debugging and line number information.

Using the Archiver to Create Libraries

An archive library is a file that includes the contents of one or more object (*.o*) files. When the linker (*ld*) searches for a symbol in an archive library, it loads only the code from the object file where that symbol was defined (not the entire 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 around within the library
- Extracting individual objects from the library
- Creating a symbol table for the linker to search symbols

The following section explains the syntax of the *ar* command and lists some examples of how to use it. See the *ar*(1) reference page for details.

Note: *ar* simply strings together whatever object files you tell it to archive; thus, it can be used to build either non−shared or PIC libraries, depending on how the included *.o* files were built in the first place. If you do create a non−shared library with *ar*, remember to link it *-non_shared* with your other code. For information about building DSOs and converting libraries to DSOs, see **Chapter 3** .

ar Syntax

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

options	Specifies the action that the archiver is to take. **Table 2−12** and **Table 2−13** list the available options. To specify more than one option, don't use a dash or put spaces between the options. For example, use *ar ts*, not *ar -t -s*.
posObject	Specifies 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. This parameter is required when the *a*, *b*, or *i* suboptions are specified with the *m* or *r* option. The last example in **"ar Examples,"** shows the use of a *posObject* parameter.
libName	Specifies the name of the archive library you are creating, updating, or extracting information from.
object	Specifies the name(s) of the object file(s) to manipulate.

ar Options

When running the archiver, specify exactly one of the options *d*, *m*, *p*, *q*, *r*, *t*, or *x* (listed in **Table 2−12**). In addition, you can optionally specify any of the modifiers in **Table 2−13** .

Table 2−12 Archiver Options

Option	Purpose
d	Deletes the specified objects from the archive.
f	Adds padding to the end of each object file archived, using the character *\n*. This enables the loader to have faster access to members in the archive while performing static linking. *Warning*: This option results in a permanent change in the size of object files.
p	Prints the specified objects in the archive on the standard output device (usually the terminal screen).
q	Adds the specified object files to the end of the archive. This option is similar to the *r* option (described below), but is faster and does not remove any older versions of the object files that may already be in the archive. Use the *q* option when creating a new library.
r	Adds the specified object files to the end of the archive file. If an object file with the same name already exists in the archive, the new object file overwrites it. Use the *r* option when updating existing libraries.
t	Prints a table of contents on the standard output (usually the screen) for the specified object or archive file.
x	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, in which case the date stamp on the archive file is the last modified date). If no objects are specified, x copies all the library objects into the current directory.

Table 2–13 Archiver Modifiers

Option	Purpose
c	Suppresses the warning message that the archiver issues when it discovers that the archive you specified does not already exist.
l	Puts the archiver's temporary files in the current working directory. Ordinarily, the archiver puts those files in */tmp* (unless the STMDIR environment variable is set, in which case *ar* stores temporary files in the directory indicated by that variable). This option is useful when */tmp* (or STMDIR) is full.
s	Creates a symbol table in the archive. This modifier is rarely necessary since the archiver updates the symbol table of the archive library automatically. Options *p*, *q*, and *r*, in particular, create a symbol table by default and thus do not require *s* to be specified.
v	Lists descriptive information during the process of creating or modifying the archive. When specified with the *t* option, produces a verbose table of contents.

ar Examples

To create a new library, *libtest.a*, and add object files to it, enter:

```
ar cq libtest.a mcount.o mon1.o string.o
```

The *c* option suppresses an archiver message during the creation process. The *q* option creates the library and puts *mcount.o*, *mon1.o*, and *string.o* into it.

To replace an object file in an existing library, enter:

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

The *r* option replaces *mon1.o* in the library *libtest.a*. If *mon1.o* does not already exist in the library *libtest.a* , it is added.

Note: If you specify the same file twice in an argument list of files to be added to an archive, that file appears twice in the archive.

Chapter 3

Using Dynamic Shared Objects

A dynamic shared object (DSO) is an object file that's meant to be used simultaneously—or *shared*—by multiple applications (*a.out* files) while they're executing. This chapter contains the following sections:

- **"Using DSOs"** tells you how to obtain the most benefit from using DSOs when creating your executable.
- **"Taking Advantage of QuickStart"** discusses an optimization you can use to make sure that the DSOs you build load as quickly as possible.
- **"Building DSOs"** describes how to build a DSO.
- **"Run–Time Linking"** discusses the run–time linker, and how it locates DSOs at run time.
- **"Dynamic Loading Under Program Control"** explains the use of *dlopen() and dlsym()* to control run–time linking.
- **"Versioning of DSOs"** discusses a versioning mechanism for DSOs that allows binaries linked against different, incompatible versions of the same DSO to run correctly.

You can use DSOs in place of archive libraries (they replace static shared libraries provided with earlier releases of IRIX).

Since DSOs contain shared components, using them provides several substantial benefits. First, overall memory usage is minimized because code is shared. Two executables that use the same DSO and that run simultaneously will have only one copy of the instruction from the shared component loaded into memory. For example, if executable A and executable B both link with the same DSO C, and if A and B are both running at the same time, the total memory used is what's required for A, B, and C, plus some small overhead. If C was an unshared library, the memory used would be what's required for A, B, and two copies of C.

A related second benefit is that executables linked with DSOs are smaller than those linked with unshared libraries because the shared objects aren't part of the executable file image, so disk usage is minimized.

Yet another feature of DSOs is that they are much easier to use, build, and debug than static shared libraries. Most of the libraries supplied by Silicon Graphics are available as DSOs. In the past, only a few static shared libraries have been available; most libraries were unshared.

Additionally, executables that use a DSO don't have to be relinked if the DSO changes; when the new DSO is installed, the executable automatically starts using it. This feature makes it easier to update end users with new software versions. It also allows you to create hardware−independent software packages more easily.

Suppose, for example, you want to build both MipsIV and a MipsIII versions of a shared object. You want your program to use the MipsIV version when it is running on a Power Challenge (R8000) system, and also run correctly on another 64−bit platform. Suppose you want to do the above with the routines in as library named *libchange.so*. To do this, build one version of the routines in *libchange* using the *-mips4* option, and place it in */usr/lib64/mips4* on a Power Challenge system. Next, build another version using the *-mips3* option, and place it in */usr/lib64*. Then, when you build an executable that uses *libchange*, use the *-rpath* option to tell the run−time linker to look first for MipsIV versions of the libraries. For example:

```
cc −mips3 −o myprog myprog.o −rpath /usr/lib64/mips4 −lchange
```

As a result, *myprog* runs on any IRIX 6.0 system, and it automatically takes advantage of any MipsIV libraries whenever it runs on a Power Challenge system.

Finally, DSOs and the executables that use them are mapped into memory by a run−time loader, *rld*, which resolves external references between objects and relocates objects at run time. (DSOs contain only position−independent code [PIC], so they can be loaded at any virtual address at run time.) With *rld*, the binding of symbols can be changed at run time at the request of the executing program. You could use this feature to dynamically change the feature set presented to a user of your application, for example, while minimizing start−up time. The application could be started quickly, with a subset of the features available and then, if the user needs other features, those can be loaded in under programmatic control.

Naturally, some costs are involved with using DSOs, and these are explained in the next section, **"Using DSOs."** The sections after that explain how to build and optimize DSOs and how *rld* works. The *dso*(5) reference page also contains more information about DSOs.

Using DSOs

Using DSOs is easy—the syntax is the same as for an archive (*.a*) library. The following compile line creates the executable *yourApp* by linking with the DSOs *libyours.so* and with *libc.so.1*:

```
cc yourApp.c −o yourApp −lyours
```

If *libyours.so* isn't available, but the archive version *libyours.a* is available, that archive version is used along with *libc.so.1*.

A significant difference exists between DSOs and archive libraries in terms of what is mapped into the address space when an application is executing. With an archive library, only the text portion of the library that the application actually requires (and the data associated with that text) is mapped, not the entire library. In contrast, the entire DSO that's linked is mapped; in many cases, however, the DSO is shared and already mapped into the address space. Thus, to conserve address space and save time at startup, don't link with DSOs unless your application actually needs them.

Avoid listing any archive libraries on the compile line after you list shared libraries; instead, list the archive libraries first and then the DSOs.

You may want to take advantage of the QuickStart optimization that minimizes start–up times for executables. You can use QuickStart when using or building DSOs. At link time, when an executable or a DSO is being created, the linker *ld* assigns initial addresses to the object and attempts to resolve all references. Since DSOs are relocatable, these initial address assignments are really only guesses about where the object will be really loaded. At run time, *rld* verifies that the DSO being used is the same one that was linked with and what the real addresses are. If the DSOs are the same and if the addresses match the initial assignments, *rld* doesn't have to perform any relocation work, and the application starts up very quickly (or QuickStarts). When an application QuickStarts, memory use is less since *rld* doesn't have to read in the information necessary to perform relocations.

To determine whether your application (or DSO) is able to do a QuickStart, use the *-quickstart_info* flag when building the executable (or DSO). If the application or DSO can't do a QuickStart, you'll be given information about what to do. The next section goes into more detail about why an executable may not be able to use QuickStart.

In summary, when you use DSOs to build an executable,

- link with only the DSOs that you need
- make sure that unshared libraries precede DSOs on the compile line
- use the *-quickstart_info* flag

Taking Advantage of QuickStart

QuickStart is an optimization designed to reduce start–up times for applications that link with DSOs. Each time *ld* builds a DSO, it updates a registry of shared objects. The registry contains the preassigned QuickStart addresses of a group of DSOs that typically cooperate by having nonoverlapping locations. (See **"Using Registry Files"** for more information about how to use the registry when you're building a DSO.) If you compile your application by linking with registered DSOs, your application takes advantage of QuickStart: all the DSOs are mapped at their QuickStart addresses, and *rld* won't need to move any of them to an unused address and perform a relocation pass to resolve all references.

Suppose you compile your application using the *-quickstart_info* flag, and Quickstart fails. It may fail because:

- Your application has directly or indirectly linked with two different versions of the same DSO, as shown in **Figure 3–1** . In this example, *yourApp* links with *libyours.so*, *libmotif.so*, and *libc.so.1* on the compile line. When the DSO *libyours.so* was built, however, it linked with *libmalloc.so*, which in turn linked with *libc.so.1* when it was created. If the two versions of *libc.so.1* weren't identical, *yourApp* won't be able to use QuickStart.

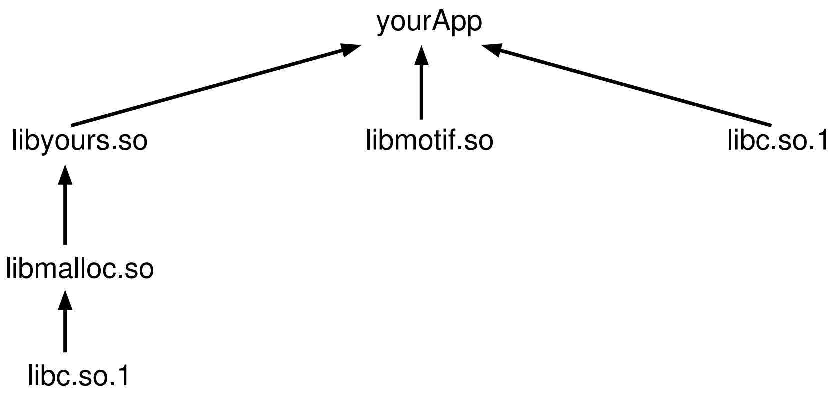

Figure 3–1 An Application Linked with DSOs

- You link with a DSO that can't use QuickStart. This may occur because the DSO wasn't registered and therefore was assigned a location that overlaps with the location assigned to another DSO.
- Your application pulls in incompatible shared objects (in a manner similar to the example shown in **Figure 3–1**).
- Your application contains an unresolved reference to a function (where it takes the address of the function).
- The DSO links with another DSO that can't use QuickStart.

Even if QuickStart officially succeeds, your application may have name space collisions and therefore may not start up as fast as it should. This is because *rld* has to bring in more information to resolve the conflicts. In general, you should avoid having conflicts both because of the detrimental effect on start−up time and because conflicts make it difficult to ensure the correctness of an application over time.

In the example shown in **Figure 3−1** , you may have written your own functions to allocate memory in *libmalloc.so* for *libyours.so* to use. If you didn't use unique names for those functions (instead of *malloc()*, for example) the way this particular compile and link hierarchy is set up, the standard *malloc()* function defined in *libc.so.1* is used instead of the one defined in *libmalloc.so*.

For example, suppose the diagram in Figure 3−3 corresponds to the following command:

```
cc −lyours −lmotif −lc
```

Since shared objects mentioned on the command line always take precedence over those that are not mentioned, the command above uses the standard *malloc()* defined in *libc.so.1*.

To get your own version of *malloc()* defined in *libmalloc.so* for *libyours.so* to use, enter:

```
cc −lyours −motif −malloc −lc
```

However, in both of the above examples, if *lyours* contains *malloc()*, you'll get that *malloc()*. (In the examples above, you do not need to specify *-lc*; it was added for clarity).

Note that conflicts are resolved by proceeding through the hierarchy from left to right and then moving to the next level. See **"Searching for DSOs at Run Time"** for more information about how the run−time linker searches for DSOs.

Thus, it's not a good idea to allow more than one DSO to define the same function. Even if the DSOs are synchronized for their first release, one of them may change the definition of the function in a subsequent release. Of course, you can use conflicts to intentionally override function definitions; however, make sure you control what is overriding what.

If you use the *-quickstart_info* option, *ld* tells you if conflicts arise. It also tells you to run *elfdump* with the *-Dc* option to find the conflicts. See the *elfdump*(5) reference page for more information about how to read the output produced by *elfdump*.

Building DSOs

In most cases, you can build DSOs as easily as archive libraries. If your library is written in a high−level language, such as C or Fortran, you won't have to make any changes to the source code. If your code is written in assembly language, you must modify it to produce PIC. This is described in "Position−Independent Coding in Assembly Language" in the *MIPSpro Assembly Language Programmer's Guide*.

To create a DSO from a set of object files, use *ld* with the *-shared* option:

```
ld −shared stuff.o nonsense.o −o libdada.so
```

The above example creates a DSO, *libdada.so*, from two object files, *stuff.o* and *nonsense.o*. Note that DSO names should begin with "*lib*" and end with "*.so*", for ease of use with the compiler driver's -*l*lib argument. If you're already building an archive library (*.a* file), you can create a DSO from the library by using the *-shared* and *-all* arguments to *ld*:

```
ld −shared −all libdada.a −o libdada.so
```

The *-all* argument specifies that all of the object files from the library, *libdada.a*, should be included in the DSO.

Making DSOs Self−Contained

When building a DSO, be sure to include any archives required by the DSO on the link line so that the DSO is self−contained (that is, it has no unresolved symbols). If the DSO depends on libraries not explicitly named on the link line, subsequent changes to any of those libraries may result in name space collisions or other incompatibilities that can prevent any applications that use the DSO from doing a QuickStart. Such incompatibilities can also lead to unpredictable results over time as the libraries change asynchronously. Suppose you want to make the archive *libmine.a* into a DSO, and *libmine.a* depends on routines in another archive, *libutil.a*. In this case, include *libutil.a* on the link line:

```
ld -shared -all -no_unresolved libmine.a -o libmine.so -none libutil.a
```

This causes the modules in *libutil.a* that are referenced in *libmine.a* to be included in the DSO, but these modules won't be exported. (For more information about exported symbols, see **"Controlling Symbols to Be Exported or Loaded."**) The *-no_unresolved* option causes a list of unresolved symbols to be created; generally, this list should be empty to enable using QuickStart.

Similarly, if a DSO relies on another DSO, be sure to include that DSO on the link line. For example:

```
ld -shared -all -no_unresolved libbtree.a -o libtree.so -lyours
```

This example places *libyours.so* in the *liblist* of the new DSO, *libtree.so*. This ensures that *libyours.so* is loaded whenever an executable that uses *libtree.so* is launched. Again, symbols from *libyours.so* won't be exported for use by other libraries. (You can use the *-exports* flag to reverse this exporting behavior; the *-hides* flag specifies the default exporting behavior.)

Controlling Symbols to Be Exported or Loaded

By default, to help avoid conflicts, symbols defined in an archive or a DSO that's used to build another DSO aren't externally visible. You can explicitly export or hide symbols with the *-exported_symbol* and *-hidden_symbol* options:

```
-exported_symbol name1, name2, name3
-hidden_symbol name4, name5
```

By default, if you explicitly export any symbols, all other symbols are hidden. If you both explicitly export and explicitly hide the same symbol on the link line, the first occurrence determines the behavior. You can also create a file of symbol names (delimited by white space) that you want explicitly exported or hidden, and then refer to the file on the link line with either the *–exports_file* or *–hiddens_file* option:

```
-exports_file yourFile
-hiddens_file anotherFile
```

These files can be used in addition to explicitly naming symbols on the link line.

Another useful option, *-delay_load*, prevents a library from being loaded until it's actually referenced. Suppose, for example, that your DSO contains several functions that are likely to be used in only a few instances. Furthermore, those functions rely on another library (archive or DSO). If you specify *-delay_load* for this other library when you build your DSO, the run–time linker loads that library only when those few functions that require it are used. Note that if you explicitly export any symbols defined in a library that the run–time linker is supposed to delay loading, the export behavior takes precedence and the library is automatically loaded at run time.

Delay–loaded shared objects do not function properly if direct references to data symbols exist in the delay–loaded object, or if the address of the function in the delay–loaded object is used. Therefore, only use *-delay_load* to load shared objects that have a purely functional interface.

Note: You can build DSOs using *cc*. However, if you want to export symbols/files or use *-delay_load*, use *ld* to build DSOs.

Using DSOs With C++

To make a DSO, build the C++ objects as you would normally:

```
CC -c
```

Then type:

```
CC -shared -o libmylib.so <list your objects here>
```

For example:

```
CC -shared -o libmylib.so a.o b.o c.o
```

The CC driver passes the *-l* and *-L* options to *ld*. However, the CC driver doesn't pass most *ld* options. If you want to specify other options, first determine the options that you must pass to *ld* via a direct invocation. These options include:

```
-init _main
-fini _fini
-hidden_symbol _main
-hidden_symbol _fini
-hidden_symbol __head
-hidden_symbol __endlink
```

Also, you must link */usr/lib/c++init.o*. To add *-delay_load -lbother,* for example, the result is similar to the following:

```
ld -shared
-o libmylib.so
a.o b.o c.o
/usr/lib/c++init.o
-fini _fini
-hidden_symbol _main
-hidden_symbol _fini
-hidden_symbol __head
-hidden_symbol __endlink
-delay_load
-delay_load -lbother
```

Using Registry Files

You can make sure that your DSOs don't conflict with each other by using a QuickStart registry file. The registry files contain location information for shared objects. When creating a shared object, you can specify a registry file to *ld*, and *ld* ensures that your shared object doesn't conflict with any of the shared objects listed in the registry. A registry file containing the locations of all the shared objects provided with the system is supplied in */usr/lib/so_locations*.

You can use two options to *ld* to specify a registry file: *-check_registry* and *-update_registry*. When you invoke *ld* to build a shared object, with the argument *-check_registry file*, *ld* makes sure that the new shared object doesn't conflict with any of the shared objects listed in *file*. When invoked with *-update_registry file*, *ld* checks the registry in the same way, but when it's done, it writes an entry in *file* for the DSO being built. If *file* isn't writable, *-update_registry* acts like *-check_registry*. If *file* isn't readable, both *-update_registry* and *-check_registry* are ignored.

By exchanging registry files, providers of DSOs can avoid collisions between their shared objects. It's best to start with a copy of */usr/lib/so_locations*, so that your shared objects don't conflict with any of the standard DSOs. However, remember that when collisions occur between shared objects, the only effect is slowing program startup.

Registry File Format

Three types of lines in the registry file include:
- comment lines, which begin with a pound sign (#)
- directive lines, which begin with a dollar sign ($)
- shared object specification lines, which begin with the name of a shared object

Comment lines are ignored by *ld*. Directive lines and shared object specification lines are described below.

Directive Lines

Directive lines specify global parameters that apply to all the DSOs listed in the registry.

```
$text_align_size=align padding=pad-size
$data_align_size=align padding=pad-size
```

These two directives specify the alignment and padding requirements for text and data segments, respectively. The current default segment alignment is 64K, which is the minimum permissible. The size value of a segment of a DSO appearing in the registry file is calculated based on the actual section size plus padding, and is aligned to the section align size (either the default or the one specified by the above directive). The align values for text and data as well as the padding values must be aligned to the minimum alignment size (64K). If not, *ld* generates a warning message and aligns these values to the minimum alignment.

```
$start_address=addr
```

This directive specifies where to start looking for addresses to put shared objects. The default start_address is 0x6000000.

```
$data_after_text={ 1 | 0 }
```

In this directive, a value of one instructs the linker to place data immediately after the text at specified text and data alignment requirements. A value of zero (the default) allows the linker to place these segments in different portions of the address space.

Shared Object Specification Lines

Shared object specification lines have the following format:

```
so_name [ :st = {.text | .data | $range} base_addr,padded_size : ] *
```

where:

so_name	full path name (or trailing component) of a shared object
:st =	literal string indicating the beginning of the segment description
.text, .data	segment types: text or data
$range	range of addresses that can be used
base_addr	address where the segment starts
padded_size	padded size of the segment
:	literal string indicating the end of the segment description

A shared object specification can span several lines by "escaping" the newline character (using "\" as the last character on the line that is being continued). The following is an example of a shared object specification line:

```
libc.so.1 \
              :st = $range 0x5fc00000, 0x00400000:\
              :st = .text 0x5fe40000, 0x000a0000:\
              :st = .data 0x5fee0000, 0x00030000:
```

This specification instructs *ld* to relocate all segments of *libc.so.1* in the range 0x5fc00000 to 0x5fc00000+0x0040000, and, if possible, to place the text segment at 0x5fe40000 and the data segment at 0x5fee0000. The text segment should be padded to 0xa0000 bytes and the data segment to 0x3000 bytes. See */usr/lib/so_locations* for examples of shared object specifications.

When building a DSO with the *-check_registry* or *-update_registry* flag, if an entry corresponding to this DSO exists in the registry file, the linker tries to assign the indicated addresses for text and data. However, if the size of the DSO changes and no longer fits in the specified location, the linker searches for another location that fits. If the *$range* option is specified, the linker places the DSO only in the specified range of addresses. If there isn't enough room, an error is returned.

Run−Time Linking

This section explains the search path followed by the run−time linker and how you can cause symbols to be resolved at run time rather than link time.

Searching for DSOs at Run Time

When you run a dynamically linked executable, the run–time linker, *rld*, identifies the DSOs required by the executable, loads the required DSOs, and if necessary relocates DSOs within the process's virtual address space, so that no two DSOs occupy the same location.The program header of a dynamically linked executable contains a field, the *liblist*, which lists the DSOs required by the executable.

When looking for a DSO, *rld* searches directories in the following sequence:

1. the path of the DSO in the *liblist* (if an explicit path is given)
2. RPATH if it's defined in the main executable
3. LD_LIBRARY_PATH if defined
4. the default path (*/usr/lib:/lib*)

RPATH is a colon–separated list of directories stored in the main executable. You can set RPATH by using the *-rpath* argument to *ld*:

```
ld –o myprog myprog.c –rpath /d/src/mylib libmylib.so –lc
```

This example links the program against *libmylib.so* in the current directory, and configures the executable such that *rld* searches the directory */d/src/mylib* when searching for DSOs.

The LD_LIBRARY_PATH environment variable is a colon–separated list of directories to search for DSOs. This can be very useful for testing new versions of DSOs before installing them in their final location. You can set the environment variable _RLD_ROOT to a colon–separated list of directories. The run–time linker prepends these to the paths in RPATH and the paths in the default search path.

In all of the colon–separated directory lists, an empty field is interpreted as the current directory. A leading or trailing colon counts as an empty field. Thus, if you set LD_LIBRARY_PATH to:

```
/d/src/lib1:/d/src/lib2:
```

the run–time linker searches the directory */d/src/lib1*, then the directory */d/src/lib2*, and then the current directory.

Note: For security reasons, if an executable has its set–user–ID or set–group–ID bits set, the run–time linker ignores the environment variables LD_LIBRARY_PATH and _RLD_ROOT. However, it still searches the directories in RPATH and the default path.

Run Time Symbol Resolution

Dynamically linked executables can contain symbol references that aren't resolved before run time. Any symbol references in your main program or in an archive must be resolved at link time, unless you specify the *-ignore_unresolved* argument to *cc*. DSOs may contain references that aren't resolved at link time. All data symbols must be resolved at run time. If *rld* finds an unresolvable data symbol at run time, it will cause the executable to exit with an error. Text symbols are resolved only when they're used, so a program can run with unresolved text symbols, as long as the unresolved symbols aren't used.

You can force *rld* to resolve text symbols at run time by setting the environment variable LD_BIND_NOW. If unresolvable text symbols exist in your executable and LD_BIND_NOW is set, the executable will exit with an error, just as if there were unresolvable data symbols.

Compiling with *-Bsymbolic*

When you compile a DSO with *-Bsymbolic*, the dynamic linker resolves referenced symbols from itself first. If the shared object fails to supply the referenced symbol, then the dynamic linker searches the executable file and other shared objects. For example:

main—defines *x*
x.so—defines and uses *x*

If you compile *x.so* with *-Bsymbolic* on, the linker tries to resolve the use of *x* by looking first for the definition in *x.so* and then by looking in *main*.

In FORTRAN programs, the linker allocates space for *COMMON* symbols and the compiler allocates space for *BLOCK DATA*. The first kind of symbol (with *COMMON* blocks present) appears in the symbol table as *SHN_MIPS_ACOMMON* (uninitialized *DATA*) whereas the second kind of symbol (with *BLOCK DATA* present) appears as *SHN_DATA* (initialized *DATA*). In general, initialized data takes precedence when the dynamic linker tries to resolve a symbol. However, with *-Bsymbolic*, whatever is defined in the current object takes precedence, whether it is initialized or uninitialized.

Variables that are declared at file scope in C with *-cckr* are also treated this way. For example:

```
int foo[100];
```

is *COMMON* if *-cckr* is used and *DATA* if *-xansi* or *-ansi* is used.

For example:

In *main*:

```
COMMON i, j /* definition of i, j with initial values */
DATA i/1/, j/1/
CALL junk
END
```

In *x.so*:

```
SUBROUTINE junk
COMMON i, j
/* definition of i, j with NO initial values */
/* initialized by kernel to all zeros */
PRINT *, i, j
END
```

When you build *x.so* using *-Bsymbolic*, this program prints `0 0`.
When you build *x.so* without *-Bsymbolic*, the program prints `1 1`.

Converting Libraries to DSOs

When you link a program with a DSO, all of the symbols in the DSO become associated with the executable. This can cause unexpected results if archives that contain unresolved externals are converted to DSOs. When linking with a PIC archive, the linker links in only those object files that satisfy unresolved references. If an object file in an archive contains an unresolved external reference, the linker tries to resolve the reference only when that object file is linked in to your program. In contrast, a DSO containing an external data reference that cannot be resolved at run time causes the program to fail. Therefore, use caution when converting archives with external data references to DSOs.

For example, suppose you have an archive, *mylib.a*, and one of the object files in the archive, *has_extern.o*, references an external variable, *foo*. As long as your program doesn't reference any symbols in *has_extern.o*, the program will link and run properly. If your program references a symbol in *has_extern.o* and doesn't define *foo*, then the link will fail. However, if you convert *mylib.a* to a DSO, then any program that uses the DSO and doesn't define *foo* will fail at run time, regardless of whether the program references any symbols from *has_extern.o*.

Two possible solutions exist for this problem. The first consists of simply adding a "dummy" definition of the data to the DSO. A data definition appearing in the main executable preempts one appearing in the DSO itself. This may, however, be misleading for executables that use the portion of the DSO that needs the data, but that failed to define it in the main program. The second solution consists of separating the routines that use the data definition into a second DSO, and placing dummy functions for them in the first DSO.

The second DSO can then be loaded dynamically the first time any of the dummy functions is accessed. Each of the dummy functions must verify that the second DSO was loaded before calling the real function (which must have a unique name). This way, programs run whether or not they supply the missing external data, as long as they don't call any of the functions that require the data. The first time one of the dummy functions is called, it tries to load the second DSO dynamically. Programs that do not supply the missing data fail at this point. For more information on dynamic loading, see **"Dynamic Loading Under Program Control"** below.

Dynamic Loading Under Program Control

IRIX provides a library interface to the run−time linker that allows programs to load and unload DSOs dynamically. The functions in this interface are part of *libc* (see **Table 3−1**).

Table 3−1 *Functions to Load/Unload DSOs*

dlopen()	Load a DSO
dlsym()	Find a symbol in a loaded DSO
dlclose()	Unload a DSO
dlerror()	Report errors

To load a DSO, call *dlopen()*:

```
include <dlfcn.h>
void *dlhandle;
     ..
dlhandle = dlopen("/usr/lib/mylib.so", RTLD_LAZY);
if (dlhandle == NULL) {
     /* couldn’t open DSO */
     printf("Error: %s\n", dlerror());
}
```

The first argument to *dlopen()* is the pathname of the DSO to be loaded. This may be either an absolute or a relative pathname. When you call this routine, the run−time linker tries to load the specified DSO. If any unresolved references exist in the executable that are defined in the DSO, the run−time linker resolves these references on demand. You can also use *dlsym()* to access symbols in the DSO, whether or not the symbols are referenced in your executable.

When a DSO is brought into the address space of a process, it may contain references to symbols whose addresses are not known until the object is loaded. These references must be relocated before the symbols can be accessed. The second argument to *dlopen()* governs when these relocations take place.

This argument can have the following values:

RTLD_LAZY	Under this mode, only references to data symbols are relocated when the object is loaded. References to functions are not relocated until a given function is invoked for the first time. This mode may result in better performance, since a process may not reference all of the functions in any given shared object.
RTLD_NOW	Under this mode, all necessary relocations are performed when the object is first loaded. This may result in some wasted effort if relocations are performed for functions that are never referenced. However, this option is useful for applications that need to know as soon as an object is loaded that all symbols referenced during execution will be available.

You can also dynamically load shared objects by using *sgidladd()*, which is similar to *dlopen()*. However, unlike *dlopen()*, all the names in the shared object become available to satisfy references in shared objects during lazy text resolution. Furthermore, it’s not necessary to use *dlsym()* to gain access to the symbols in the shared object. *sgidladd()* is available as part of *libc*. For more information, see the *sgidladd*(3) reference page.

To access symbols that are not referenced in your program, use *dlsym()*:

```
#include <dlfcn.h>
void *dlhandle;
int (*funcptr)(int);
int i,j;
     .. load DSO ...
funcptr = (int (*)(int)) dlsym(dlhandle, "factorial");
if (funcptr == NULL) {
```

```
        /* couldn't locate the symbol */
        exit();
}
i = (*funcptr)(j);
```

This example looks up the address of the function *factorial()* and assigns it to the function pointer *funcptr*.

If you encounter an error (*dlopen()* or *dlsym()* returns NULL), you can get diagnostic information by calling *dlerror().* The *dlerror()* function returns a string describing the cause of the latest error. You should call *dlerror()* only after an error has occurred; at other times, its return value is undefined.

To unload a DSO, call *dlclose()*:

```
#include <dlfcn.h>
void *dlhandle;
... load DSO, use DSO symbols ...
dlclose(dlhandle);
```

The *dlclose* function frees up the virtual address space that has been *mmap*ed by the *dlopen* call of that file (similar to a *munmap* call). The difference, however, is that a *dlclose* on a file that has been opened multiple times (either through *dlopen* or program startup) does not cause the file to be *munmap*ed until the file is no longer needed by the process.

Versioning of DSOs

In the IRIX 5.0.1 release, a mechanism for the versioning of shared objects was introduced for the Silicon Graphics shared objects and executables. Note that this mechanism is outside the scope of the MIPS ABI, and, thus, must not be relied on for code that must be MIPS ABI–compliant and run on other vendors' platforms. Currently, all executables produced on Silicon Graphics systems are marked SGI_ONLY to allow use of the versioning mechanism.

Versioning is of interest mainly to developers of shared objects. It may not be of interest to you if you simply *use* shared objects. Versioning allows a developer to update a shared object in a way that may be incompatible with executables previously linked against the shared object. You can accomplish this by renaming the original shared object and providing it along with the (incompatible) new version.

What Is a Version?

A version is part or all of an identifying *version_string* that can be associated with a shared object by using the *-set_version version_string* option to *ld*(1) when the shared object is created.

A *version_string* consists of one or more versions separated by colons (:). A single version has the form:

[*comment***#**]**sgi***major***.***minor*

where:

comment	is a comment string, which is ignored by the versioning mechanism. It consists of any sequence of characters followed by a pound sign (#). The comment is optional.
sgi	is the literal string *sgi*.
major	is the major version number, which is a string of digits [0–9].
.	is a literal period.
minor	is the minor version number, which is a string of digits [0–9].

Building a Shared Library Using Versioning

Follow these instructions when building your shared library:

When you first build your shared library, give it an initial version, for example, *sgi1.0*. Add the option *-set_version sgi1.0* to the command to build your shared library (*cc -shared*, *ld -shared*).

Whenever you make a *compatible* change to the shared object, create another version by changing the minor version number (for example, *sgi1.1*) and add it to the end of the *version_string*. The command to set the version of the shared library now looks like *-set_version "sgi1.0:sgi1.1"*.

When you make an *incompatible* change to the shared object:

1. Change the filename of the old shared object by adding a dot followed by the major number of one of the versions to the filename of the shared object. Do not change the *soname* of the shared object or its contents. Simply rename the file.
2. Update the major version number and set the *version_string* of the shared object (when you create it) to this new version; for example, *-set_version sgi2.0*.

This versioning mechanism affects executables in the following ways:

- When an executable is linked against a shared object, the last version in the shared object's *version_string* is recorded in the executable as part of the *liblist*. You can examine this using *elfdump -Dl*.
- When you run an executable, *rld* looks for the proper filename in its usual search routine.
- If a file is found with the correct name, the version specified in the executable for this shared object is compared to each of the versions in the *version_string* in the shared object. If one of the versions in the *version_string* matches the executable's version exactly (ignoring comments), then that library is used.
- If no proper match is found, a new filename for the shared object is built by combining the *soname* specified in the executable for this shared object and the *major* number found in the version specified in the executable for this shared object (*soname.major*). Remember that you did *not* change the *soname* of the object, only the filename. The new file is searched for using *rld*'s usual search procedure.

Example of Versioning

For example, suppose you have a shared object *foo.so* with initial version *sgi10.0*. Over time, you make two compatible changes for *foo.so* that result in the following final *version_string* for *foo.so*:

```
initial_version#sgi10.0:upgrade#sgi10.1:new_devices#sgi10.2
```

You then link an executable that uses this shared object, *useoldfoo*. This executable specifies version *sgi10.2* for *soname* *foo.so*. (Remember that the executable inherits the last version in the *version_string* of the shared object.)

The time comes to upgrade *foo.so* in an incompatible way. Note that the *major* version of *foo.so* is 10, so you move the existing *foo.so* to the filename *foo.so.10* and create a new *foo.so* with the *version_string*:

```
efficient_interfaces#sgi11.0
```

New executables linked with *foo.so* use it directly. Older executables, like *useoldfoo*, attempt to use *foo.so*, but find that its version (*sgi11.0*) is not the version they need (*sgi10.2*). They then attempt to find a *foo.so* in the filename *foo.so.10* with version *sgi10.2*.

When a needed DSO has its interface changed, then a new version is created. If the interface change is not compatible with older versions, then a consuming shared object needs incompatible versions in order to use the new version, even if it doesn't use that part of the interface that is changed.

Chapter 4

Using the Debugger

dbx is a source–level debugger that you can use to debug programs such as C, C++, and Fortran77. This chapter describes *dbx* and covers the topics listed below:

- **"Getting Started With dbx"** introduces some basic *dbx* commands and provides tips about how to approach a debugging session
- **"Running dbx"** explains how to run *dbx* and perform basic *dbx* control functions
- **"Examining Source Files"** explains how to examine source files under *dbx*
- **"Controlling dbx"** describes features of *dbx* that affect its operation while debugging a program
- **"Examining and Changing Data"** describes how to examine and change data in your program while running it under *dbx*
- **"Controlling Program Execution"** describes how to use the *dbx* commands that control execution

of your program

- **"Debugging Machine Language Code"** explains how to debug machine language code
- **"Multiple Process Debugging"** explains multiprocess debugging procedures
- **"dbx Commands," "Predefined Aliases,"** and **"Predefined dbx Variables"** list commands, aliases, and variables, respectively.

You can also find information about *dbx* in the *dbx*(1) reference page; *odbx*(1) describes *dbx* for IRIX releases 5.2 and below.

Getting Started With *dbx*

You can use *dbx* to trace problems in a program at the source code level, rather than at the machine code level. *dbx* enables you to 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.

This section introduces some basic *dbx* commands and discusses some tips about how to approach a debugging session. In particular, this section describes:

- **"Examining Core Dumps to Determine Cause of Failure"**
- **"Debugging Your Programs"**
- **"Studying a New Program"**
- **"Avoiding Common Pitfalls"**

Examining Core Dumps to Determine Cause of Failure

Even if your program compiles successfully, it still might crash when you try to run it. When a program crashes, it generates a terminating signal that instructs the system to write out to a *core* file. The *core* file is the memory image of the program at the time it crashed.

You can examine the *core* file with *dbx* to determine at what point your program crashed. To determine the point of failure, follow these steps:

1. If the *core* file is not in the current directory, specify the pathname of the *core* file on the *dbx* command line.

   Note: If the source code for the program is on a different machine or the source was moved, provide *dbx* with the pathname to search for source code (also see **"Specifying Source Directories"**).

2. Invoke *dbx* for the failed program as described in **"Invoking dbx".** *dbx* automatically reads in the local *core* file.
3. Perform a stack trace using the *where* command (described in **"Examining the Stack"**) to locate the failure point.

For example, suppose you examine the *core* file for a program called *test*. Suppose the stack trace appears as follows:

```
(dbx) where
>  0 foo2(i = 5) ["/usr/tmp/test.c":44, 0x1000109c]
   1 foo(i = 4) ["/usr/tmp/test.c":38, 0x1000105c]
   2 main(argc = 1, argv = 0xffffffad78) ["/usr/tmp/test.c":55, 0x10001104]
   3 __start() ["/shamu/crt1text.s":137, 0x10000ee4]
```

In this case, *test* crashed at line 44 of the source file *test.c*. The program crashed while executing the function *foo2*. *foo2* was called from line 38 in the function *foo*, which was in turn called from line 55 in the function *main*. You can then use the other features of *dbx* to examine values of program variables and otherwise investigate why *test* crashed.

Note: If you use *dbx* to debug code that wasn't compiled using the *-g* option, local variables are invisible to *dbx*, and source lines may appear to jump around as a result of various optimizations. If the code is stripped of its debugging information, *dbx* can display very little information.

Debugging Your Programs

Debugging a program consists primarily of stopping your program under certain conditions and then examining the state of the program stack and the values stored in program variables.

You stop execution of your program by setting *breakpoints* in your program. Breakpoints can be *unconditional*, in which case they always stop your program when encountered, or *conditional*, in which case they stop your program only if a test condition that you specify is true. (See **"Setting Breakpoints"** for more information.)

To use breakpoints to debug your program, examine your program carefully to determine where problems are likely to occur, and set breakpoints in these problem areas. If your program crashes, first determine which line causes it to crash, then set a breakpoint just before that line.

Several *dbx* commands can be used to trace a variable's value. Here's a simple method for tracing a program variable:

1. Use the *stop* command (described in **"Setting Breakpoints"**) to set breakpoints in the program at locations where you want to examine the state of the program stack or the values stored in program variables.
2. Use the *run* or *rerun* command (described in **"Running Your Program"**) to run your program under dbx. The program stops at the first breakpoint that it encounters during execution.
3. Examine the program variable as described in **"Displaying the Value of a Variable".** Examine the program stack as described in **"Examining the Stack".**
4. Use the *cont* command (described in **"Continuing Execution After a Breakpoint"**) to continue execution past a breakpoint. However, you cannot continue execution past a line that crashes the program.

Studying a New Program

Use *dbx* to examine the flow of control in a program. When studying the flow of control within a program, use the *dbx* commands *stop*, *run*/*rerun*, *print*, *next*, *step*, and *cont*. This is the procedure:

1. Use the *stop* command to set breakpoints in the program. When you execute the program under *dbx*, it stops execution at the breakpoints.

   If you want to review every line in the program, set a breakpoint on the first executable line. If you don't want to look at each line, set breakpoints just before the sections you intend to review.
2. Use the *run* and *rerun* commands to run the program under *dbx*. The program stops at the first breakpoint.
3. Use the *print* command to print the value of a program variable at a breakpoint.
4. Use the *step*, *next*, or *cont* command to continue past a breakpoint and execute the rest of the program.
   - *step* executes the next line of the program. If the next line is a procedure call, *step* steps down into the procedure. *step* is described in **"Stepping Using the step Command".**
   - *next* executes the next line; if it is a procedure, *next* executes it but does not step down into it. *next* is described in **"Stepping Using the next Command".**
   - *cont* resumes execution of the program past a breakpoint and does not stop until it reaches the next breakpoint or the end of the program. *cont* is explained in "Continuing Execution after a Breakpoint."

Another tool that you can use to follow the execution of your program is the *trace* command. With it you can examine:

- values of variables at specific points in your program or whenever variables change value
- parameters passed to and values returned from functions
- line numbers as they are executed

The *trace* command is described in **"Tracing Program Execution".**

Avoiding Common Pitfalls

You may encounter some problems when you debug a program. Common problems and their solutions are listed below:

- If *dbx* does not display variables, recompile the program with the *-g* compiler option. Note that in some cases, this may cause the problem to go away, or its symptoms to change.
- If the debugger's listing seems confused, try separating the lines of source code into logical units. The debugger may get confused if more than one source statement occurs on the same line.
- If the debugger's executable version of the code doesn't match the source, recompile the source code. The code displayed in the debugger is identical to the executable version of the code.
- If code appears to be missing, it may be contained in an include file or a macro. The debugger treats macros as single lines. To debug a macro, expand the macro in the source code.

Running *dbx*

This section explains how to run *dbx*—specifically, it covers:

- **"Compiling a Program for Debugging Under dbx"**
- **"Compiling and Linking Programs With Dynamic Shared Objects"**
- **"Invoking dbx"** from a shell
- **"Running Your Program"**
- **"Automatically Executing Commands on Startup"**
- **"Using Online Help"**
- **"Entering Multiple Commands on a Single Line"**
- **"Spanning a Command Across Multiple Lines"**
- **"Invoking a Shell"** from *dbx*
- **"Quitting dbx"**

Compiling a Program for Debugging Under *dbx*

Before using *dbx* to debug a program, compile the program with the *-g* option. The *-g* option includes additional debugging information in your program object so that *dbx* can list local variables and find source lines.

If you use *dbx* to debug code that was not compiled using the *-g* option, local variables are invisible to *dbx*, and source lines may appear to jump around oddly as a result of various optimizations. It is more difficult to debug code without reliable references to lines of source code. For more information about the *-g* option, see **"Debugging".**

Compiling and Linking Programs With Dynamic Shared Objects

This section summarizes a few things you need to know if you compile and link your program with Dynamic Shared Objects (DSOs). A DSO is a relocatable shared library. By linking with a DSO, you keep your program size small and use memory efficiently. See also the *dbx* help section on *hint_dso* for more information on *dbx* and DSOs.

If you compile and link with DSOs, *dbx* automatically notices their use in the program and picks up the relevant debugging information. The *dbx* command *listobj* shows any DSOs in a process. The *dbx* command *whichobj* lists all DSOs in which the named variable is present.

See **"Running Your Program"** for more description of the differences between programs compiled and linked with DSOs and programs compiled and linked with non–shared libraries. For more information, see **"Using Dynamic Shared Objects"** .

Invoking *dbx*

To invoke *dbx* from the shell command line, type `dbx`. The syntax is:

```
dbx [options] [object_file [corefile]]
```

dbx Options

Table 4–1 lists options you can give to *dbx*. These options are described in detail later in this chapter.

Table 4–1 *dbx Command–Line Options*

Option	Description
-I dir	Tells *dbx* to look in the specified directory (in addition to the current directory and the object file's directory) for source files. To specify multiple directories, use a separate *-I* for each. If no directory is specified when you invoke *dbx*, it looks for source files in the current directory and in the object file's directory. From *dbx*, changes the directories searched for source files with the *use* and *dir* commands.
-c file	Selects a command file other than *.dbxinit* to execute on starting *dbx*. For information on *.dbxinit*, see **"Automatically Executing Commands on Startup."**
-e num	Chooses a large size for the evaluation stack (as large as you want). The default stack size is 20,000 bytes. *num* = number of bytes. If you see the message `too large to evaluate`, rerun *dbx* suppling a value greater than 20,000.
-k	Turns on kernel debugging. When debugging a running system, specify */dev/kmem* as the core file.
-i	Uses interactive mode. This option prompts for source even when it reads from a file and treats data in a file as if it comes from a terminal (*stdin*). This option does not treat "#" characters as comments in a file.
-p pid	Debugs the process specified by the *pid* number.
-P name	Debugs the running process with the specified *name* (*name* as described in the *ps*(1) reference page).
-r program [*arg*]	Runs the named program upon entering *dbx*, using the specified arguments. The *.dbxinit* file (if any) is read and executed *after* executing the *object_file*. You cannot specify a core file with *-r*.

Specifying Object and Core Files

The *object_file* is the name of the executable object file that you want to debug. It provides both the code that *dbx* executes and the symbol table that provides variable and procedure names and maps executable code to its corresponding source code in source files.

A *core file* is produced when a program exits abnormally and produces a core dump. *dbx* allows you to provide the name of a core file that it uses as "the contents of memory" for the program that you specify. If you provide a core file, *dbx* lists the point of program failure. You can then perform stack traces and examine variable values to determine why a program crashed. However, you cannot force the program to execute past the line that caused it to crash.

If you don't specify a *corefile*, *dbx* examines the current directory for a file named *core*. If it finds *core*, and if *core* seems (based on data in the core file) to be a core dump of the program you specified, *dbx* acts as if you had specified *core* as the core file.

You can specify object and core files either as arguments when you invoke *dbx* or as commands that you enter at the *dbx* prompt.

The *dbx* Prompt

Once *dbx* starts, it displays the prompt:

```
(dbx)
```

To change this prompt, change the value of the *dbx* *$prompt* variable. **"Setting dbx Variables"** describes how to set *dbx* variables.

Specifying Files with *dbx* Commands

The *givenfile* and *corefile* *dbx* commands allow you to set the object file and the core file, respectively, while *dbx* is running:

`givenfile` [*file*]

If you provide a filename, *dbx* kills the currently running processes and loads the executable code and debugging information found in *file*.
If you do not provide a filename, *dbx* displays the name of the program that it is currently debugging without changing it.

`corefile` [*file*]

If you provide a filename, *dbx* uses the program data stored in the core dump *file*.
If you do not provide a filename, *dbx* displays the name of the current core file without changing it.

Running Your Program

You can start your program under *dbx* using the *run* or *rerun* command:

`run` *run−arguments*

The *run* command starts your program and passes to it any arguments that you provide. *run* accepts */bin/sh* syntax for a simple command. All shell processing is accepted, such as expansion and substitution of * and ? in filenames. Redirection of the program's standard input and standard output, and/or standard error is also done by the shell. In other words, the *run* command does exactly what typing *target run−arguments* at the shell prompt does. You can specify *target* either on *dbx* invocation or in a prior *givenfile* command. *dbx* passes *./target* as *argv[0]* to *target* when you specify it as a relative pathname.
A *run* command must appear on a line by itself and cannot be followed by another *dbx* command separated by a semi−colon (;). Terminate the command line with a return (new−line). Note that you cannot include a *run* command in the command list of a *when* command.

`rerun` [*run−arguments*]

The *rerun* command, without any arguments, repeats the last *run* command if applicable. Otherwise *rerun* is equivalent to the *run* command without any arguments.

The *sort* command takes an input file and produces a sorted output file; you can specify input and output files either through command−line arguments or file redirection.

For example, from the command line you can enter:

```
% sort −i input −o output
% sort < input2 > output2
```

If you are debugging the *sort* program, the equivalent **`dbx`** commands are:

```
(dbx) run −i input −o output
(dbx) run < input2 > output2
```

If you execute these *run* commands in the order presented, you can repeat the last run command by using the *rerun* command:

```
(dbx) rerun
```

Automatically Executing Commands on Startup

You can use an editor to create a *.dbxinit* command file. This file contains various *dbx* commands that automatically execute when you invoke *dbx*. You can put any *dbx* command in the *.dbxinit* file. If a command requires input, the system prompts you for it when you invoke *dbx*.

On invocation, *dbx* looks for a *.dbxinit* file in the current directory. If the current directory does not contain a *.dbxinit* file, *dbx* looks for one in your home directory. (This assumes that you have set the IRIX system *HOME* environment variable.)

Using Online Help

The *dbx* command *help* has several options:

`help`	shows the supported *dbx* commands
`help` *keyword*	shows information pertaining to the given *keyword*, such as *alias, help, most_used, quit, playback, record,* and so on
`help all`	shows the entire *dbx* help file

When you type `help all`, *dbx* displays the file using the command name given by the *dbx* *$pager* variable. The *dbx* help file is large and can be difficult to read even if you use a simple paging program like *more*(1). You can set the *$pager* variable to a text editor like *vi*(1) or to your favorite editor.

For example, just add the following command in your *.dbxinit* file:

```
set $pager = vi
```

When the above entry is in your *.dbxinit* file, *dbx* displays the help file in *vi*. You can then use the editor's search commands to look through the help file quickly. Quit the editor to return to *dbx*.

Entering Multiple Commands on a Single Line

You can use a semicolon (;) as a separator to include multiple commands on the same command line. This is particularly useful with commands such as *when* (described in **"Writing Conditional Commands"**) as it allows you to include multiple commands in the command block. For example:

```
(dbx)   when at "myfile.c":37 {print a ; where ; print b}
```

Spanning a Command Across Multiple Lines

You can use a backslash (\) at the end of a line of input to indicate that the command is continued on the next line. This can be convenient when entering complex commands such as an alias definition (aliases are discussed in **"Creating and Removing dbx Aliases"**). For example:

```
(dbx) alias foll "print *(struct list *)$p ; \
set $p = (int)((struct list *)($p))->next"
```

Hint: You can also use the *hed* command for creating and modifying commands. **"The History Editor"** has details on this command.

Invoking a Shell

To invoke a subshell, enter `sh` at the dbx prompt, or enter `sh` and a shell command at the dbx prompt. After invoking a subshell, type `exit` or `<Ctrl-d>` to return to dbx.

The syntax for the *sh* command is:

`sh`	Invoke a subshell.
`sh` *command*	Execute the specified shell command. *dbx* interprets the rest of the line as a command to pass to the spawned shell process, unless you enclose the command in double−quotes or you terminate your shell command with a semicolon (;).

For example, to spawn a subshell, enter:

```
(dbx) sh
%
```

To display the end of the file *datafile*, enter:

```
(dbx) sh tail datafile
```

Quitting *dbx*

To end a *dbx* debugging session, enter the *quit* command at the *dbx* prompt:

```
(dbx) quit
```

Examining Source Files

This section explains how to examine source files under *dbx*. It describes:

- **"Specifying Source Directories"**
- **"Changing Source Files"**
- **"Listing Source Code"**
- **"Searching Through Source Code"**
- **"Calling an Editor"**

Specifying Source Directories

Based on the information contained in an object file's symbol table, *dbx* determines from which source files the program was compiled and prints portions of these files as appropriate.

Object files compiled with *-g* record both the source filenames and the absolute path names to the source files. Each time *dbx* needs a source file, it first searches the absolute path for the source file. If the source file is not present (or if the object file was not compiled with *-g*), *dbx* checks its own list of directories for source files.

By default, the *dbx* directory list contains only the current directory (from which you invoked *dbx*) and the object file's directory (if it is different from the current directory). Each time *dbx* searches this list, it searches all directories in the list in the order in which they appear until it finds the file with the specified name.

Specifying Source Directories With Arguments

You can specify additional source directories when you invoke *dbx* with the *-I* option. To specify multiple directories, use a separate *-I* for each.

For example, consider debugging a program called *look* in */usr/local/bin*, the source for which resides in */usr/local/src/look.c*. To debug this program, you can invoke *dbx* from the */usr/local/bin* directory by entering:

```
% dbx –I /usr/local/src look
```

Specifying Source Directories With *dbx* Commands

The *dir* and *use* commands allow you to specify a source directory list while *dbx* is running.

`dir` [*dir ...*]	If you provide one or more directories, *dbx* adds those directories to the end of the source directory list. If you do not provide any directories, *dbx* displays the current source directory list.
`use` [*dir ...*]	If you provide one or more directories, *dbx* replaces the source directory list with the directories that you provide. If you do not provide any options, *dbx* displays the current source directory list.

Note: Both the *dir* and *use* commands recognize absolute and relative pathnames (for example, *../src*); however, they do not recognize C shell tilde (~) syntax (for example, *~kim/src*) or environment variables (for example, *$HOME/src*).

Examples of *dir* and *use*

Let's debug the *look* program in */usr/local/bin*. Recall that the source resides in */usr/local/src/look.c*. If you invoke *dbx* from the */usr/local/bin* directory without specifying */usr/local/src* as a source directory, it will not initially appear in the directory list:

```
(dbx) dir
.
```

However, you can add */usr/local/src* with the *dir* command by entering:

```
(dbx) dir /usr/local/src
(dbx) dir
. /usr/local/src
```

If you use the *use* command instead, the current directory is no longer contained in the source directory list:

```
(dbx) use /usr/local/src
(dbx) use
/usr/local/src
```

Using Path Remapping

Files compiled with *-g* have full pathnames to source files. If you're debugging a program that was compiled somewhere else and you want to specify a new path to the sources, use path remapping. Just substitute one pattern for another pattern to remap the path so *dbx* can find the source file.

`dir` *pattern1*`:`*pattern2*
The *dir* (or *use*) command allows you to remap directories and specify a new path to the source. *dbx* substitutes *pattern2* for *pattern1*.

For example, a compiled program's source is */x/y/z/kk.c*. The source was moved to */x/y/zzz/kk/kk.c*. Specify the *dir* (or *use*) command to remap the path:

```
(dbx) dir /z/:/zzz/kk/
```

The new path is */x/y/zzz/kk/kk.c* where */z/* is replaced by the path specified after the colon.

Changing Source Files

The file command changes the current source file to a file that you specify. The new file becomes the current source file, on which you can search, list, and perform other operations. For example, to set the current source file to *"Examining the Stack" on page 54procedure.c*, enter:

```
(dbx) file procedure.c
```

Note: If your program is large, typically you store the source code in multiple files. *dbx* automatically selects the proper source file for the section of code that you are examining. Thus, many *dbx* commands reset the current source file as a side effect. For example, when you move up and down activation levels in the stack using the *up* and *down* commands, *dbx* changes the current source file to whatever file contains the source for the procedure (see **"Examining the Stack"** for more information on activation levels).

If you enter the *file* command without any arguments, *dbx* prints the current source file:

```
(dbx) file
procedure.c
```

You can also change the current source file by typing:

(dbx) **`func`** *procedure*

You can use the *tag* command to search the tag file for *procedure*:

```
(dbx) tag   procedure
```

The *tag* command finds C preprocessor macros if they have arguments (**`func`** *procedure* cannot). For more information about the tag file, see *ctags*(1).

Listing Source Code

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 active frame and line of the current source file unless overridden by a *file* command. Any execution of the program overrides the *file* command by establishing a new current source file.

The syntax for the *list* command is:

`list` Lists *$listwindow* lines beginning at the current line (or list the line of the current *pc* if

	the current line is unknown or not set).
list *exp*	Lists *$listwindow* lines starting with the line number given by the expression *exp*. The expression can be any valid expression that evaluates to an integer value as described in **"Using Expressions".**
list *exp1*: *exp2*	Lists *exp2* lines, beginning at line *exp1*.
list *exp1*, *exp2*	Lists all source between line *exp1* and line *exp2* inclusive.
list *func*	Lists *$listwindow* lines starting at procedure *func*.
list *func*, *exp*	Lists all source between *func* and *exp*, inclusive.
list *func*: *exp*	Lists *exp* lines, beginning at *func*.

A > symbol prints to the left of the line that is the current line. A * symbol prints to the left of the line of the current *pc* location.

For example, to list lines 20-35 of a file, enter:

```
(dbx) list 20,35
```

In response to this command, *dbx* displays lines 20 through 35 and sets the current line to 36.

To list 15 lines starting with line 75, enter:

```
(dbx) list 75:15
```

In response to this command, *dbx* displays lines 75 through 89 and sets the current line to 90.

Searching Through Source Code

Use the forward slash (*/*) and question mark (*?*) commands to search through the current file for regular expressions in source code. For a description of regular expressions, see the *ed*(1) reference page.

The search commands have the following syntax:

/[*reg_exp*]	Search forward through the current file from the current line for the regular expression *reg_exp*. If *dbx* reaches the end of the file without finding the regular expression, it wraps around to the beginning of the file. *dbx* prints the first source line containing a match of the search expression. If you don't supply a regular expression, *dbx* searches forward based on the last regular expression searched.
?[*reg_exp*]	Search backward through the current file from the current line for the regular expression *reg_exp*. If *dbx* reaches the beginning of the file without finding the regular expression, it wraps around to the end of the file. *dbx* prints the first source line containing a match of the search expression. If you don't supply a regular expression, *dbx* searches backward based on the last regular expression searched.

For example, to search forward for the next occurrence of the string "errno," enter:

```
(dbx) /errno
```

To search backward for the previous occurrence of either "img" or "Img," enter:

```
(dbx) ?[iI]mg
```

Calling an Editor

The *edit* command lets you edit files from within *dbx*:

edit	*The edit* command invokes an editor (*vi* by default) on the current source file. If you set the *dbx* variable *$editor* to the name of an editor, the *edit* command invokes that editor. If you do not set the *$editor*, *dbx* checks the environment variable *EDITOR* and, if set, invokes that editor. When you exit the editor, you return to the *dbx* prompt.
edit *file*	The *edit* command invokes the editor on the given file.
edit *procedure*	The *edit* command invokes the editor on the file that contains the source for the given procedure.

For example, to edit a file named *soar.c* from within *dbx*, type:

```
(dbx) edit soar.c
```

The *edit* command is also useful for editing *dbx* script files. See **"Executing dbx Scripts"** for more information on script files.

Controlling *dbx*

This section describes features of *dbx* that affect its operation while debugging a program. Specifically, this section covers:

- **"Creating and Removing dbx Variables"**
- **"Using the History Feature and the History Editor"**
- **"Creating and Removing dbx Aliases"**
- **"Recording and Playing Back dbx Input and Output"**
- **"Executing dbx Scripts"**

Creating and Removing *dbx* Variables

dbx allows you to define variables that you can use within *dbx* to store values. These variables exist entirely in *dbx*; they are not part of your program. You can use *dbx* variables for a variety of purposes while debugging. For example, you can use *dbx* variables as temporary storage, counters, or pointers that you use to step through arrays.

dbx also provides many predefined variables that control how various *dbx* commands function. **"Predefined dbx Variables"** provides a complete list of predefined *dbx* variables and their purposes.

A dbx variable does not have a fixed type. You can assign a *dbx* variable any type of value, even if it already has a value of a different type. However, a variable predefined by *dbx* does have a fixed predefined type.

You can use almost any name for *dbx* variables. A good practice to follow is to use a dollar sign ($) as the first character of all *dbx* variables to prevent conflicts with most program variable names. All of *dbx*'s predefined variables begin with a dollar sign.

The commands described in this section apply only to the manipulations of *dbx* variables, not program variables. **"Displaying and Changing Program Variables"** describes how to manipulate program variables.

Setting *dbx* Variables

The *set* command sets a *dbx* variable to a given value, defining the variable if it does not exist:

set *var* **=** *exp* Define (or redefine) the specified *dbx* variable, setting its value to that of the expression you provide.

You can display the value of a variable with the *print* command. For example:

```
(dbx) set $k = 1
(dbx) print $k
1
(dbx) set $k = $k +23
(dbx) print $k
24
(dbx) print $k / 11
2
```

In the above example, *dbx* performs an integer division because both the variable *$k* and the constant 11 are integers. If you assign a floating point value to *$k* and evaluate the expression again, *dbx* performs a floating point division:

```
(dbx) set $k = 24.0
(dbx) print $k
24.0
(dbx) print $k / 11
2.1818181818181817
```

Note: We recommend that you begin a *dbx* variable with a *$* to avoid confusion with a program variable. A *dbx* variable without a leading *$* hides any program variable that has the same name. The only way to see the program variable is to remove the *dbx* variable with an *unset* command.

Listing *dbx* Variables

If you enter the *set* command without providing any arguments, *dbx* displays (in alphabetical order) a list of all currently defined *dbx* variables, including predefined variables. Partial output looks like this:

```
(dbx) set
$addrfmt        "0x%x"
$addrfmt64      "0x%llx"
$assignverify   1
$casesense      2
$ctypenames     1
$curevent       3
$curline        44
$curpc          268439708
...
$stacktracelimit        1024
$stdc           0
$stepintoall    0
$tagfile        "tags"
```

Removing Variables

The *unset* command removes a *dbx* variable. For example, to delete the variable *$k*, enter:

```
(dbx) unset $k
```

Using the History Feature and the History Editor

The *dbx* history feature is similar to the C shell's history feature in that it allows you to repeat commands that you have entered previously. However, unlike the C shell's history feature, *dbx* does not allow you to execute a history command anywhere except the beginning of a line. Also, *dbx* does not support history substitution of command arguments such as the C shell *!$* argument.

Examining the History List

dbx stores all commands that you enter in the history list. The value of the *dbx* variable *$lines* determines how many commands are stored in the history list. The default value is 100.

Display the history list with the *history* command. For example, after setting a breakpoint, running a program, and examining some variables, your history list might look something like this:

```
(dbx) history
    1     set $prompt = "(dbx)"
    2     set $page=0
    3     set $pimode=1
    4     stop in main
    5     r
    6  history
```

Repeating Commands

You can execute any of the commands contained in the history list. Each history command begins with an exclamation point (*!*):

`!!` Repeats the previous command. If the value of the *dbx* variable *$repeatmode* is set to 1, then entering a carriage return at an empty line is equivalent to executing *!!*. By

	default, *$repeatmode* is set to 0.
! *string*	Repeats the most recent command that starts with the specified *string*.
! *integer*	Repeats the command associated with the specified *integer* in the history list.
!-*integer*	Repeats the command that occurred *integer* times before the most recent command. Entering `!-1` executes the previous command, `!-2` the command before that, and so forth.

You can use the *!!* command to facilitate single−stepping through your program. (Single−stepping is described in **"Stepping Through Your Program".**) The following illustrates using the *next* command to execute 5 lines of source code and then using the *!!* command to repeat the *next* command.

For example:

```
(dbx) next 5
Process 22545 (test) stopped at [main:60 ,0x10001150]
    60  total += j;
(dbx) !!
(!! = next 5)
Process 22545 (test) stopped at [main:65 ,0x100011a0]
    65  printf("i = %d, j = %d, total = %d\n",i,j,total);
```

Another convenient way to repeat a commonly used command is with **!** *string*. For example, suppose that you occasionally print the values of certain variables using the *printf* command while running your program under *dbx*. (The *printf* command is described in **"Printing Expressions".**) In this case, as long as you do not enter any command beginning with "pr" after you enter the *printf* command, you can repeat the *printf* command by entering `!pr`. For example:

```
(dbx) printf "i = %d, j = %d, total = %d\n", i, j, total
i = 4, j = 25, total = 1
 ...
(dbx) !pr
i = 12, j = 272, total = 529
```

Using **!** *integer*, you can repeat any command in the history list. If you want to repeat the *printf* command, but you have entered a subsequent *print* command, examine the history list and then explicitly repeat the *printf* command using its reference number. For example:

```
(dbx) history
    1     set $prompt = "(dbx)"
    2     set $page=0
 ...
 45 printf "i = %d, j = %d, total = %d\n", i, j, total
 46 next
 ...
 49 print j
 ...
 53 history
(dbx) !45
(!45 = printf "i = %d, j = %d, total = %d\n", i, j, total)
i = 9, j = 43, total = 1084
```

The History Editor

The history editor, *hed*, lets you use your favorite editor on any or all of the commands in the current *dbx* history list. When you enter the *hed* command, *dbx* copies all or part of the history list into a temporary file that you can edit. When you quit the editor, any commands left in this temporary file are automatically executed by *dbx*.

If you have set the *dbx* variable *$editor* to the name of an editor, the *hed* command invokes that editor. If you have not set the *dbx* variable *$editor*, *dbx* checks whether you have set the environment variable *EDITOR* and, if so, invokes that editor. If you have not set either the *dbx* variable or the environment variable, *dbx* invokes the *vi* editor.

The syntax for the *hed* commands is:

hed	Edits only the last line of the history list (the last command executed).
hed *num1*	Edits line *num1* in the history list.
hed *num1*, *num2*	Edits the lines in the history list from *num1* through *num2*.
hed all	Edits the entire history list.

By default, *dbx* doesn't display the commands that it executes as a result of the *hed* command (the *dbx* variable *$pimode* is set to 0). If *$pimode* is set to 1, *dbx* displays the commands as it executes them. See *$pimode* in **"Predefined dbx Variables"** for more information.

Creating and Removing *dbx* Aliases

You can create *dbx* aliases for debugger commands. Use these aliases as you would any other *dbx* command. When *dbx* encounters an alias, it expands the alias using the definition you provided.

dbx has a group of predefined aliases that you can modify or delete. These aliases are listed and described in **"Predefined Aliases".**

If you find that you often create the same aliases in your debugging sessions, you can include their definitions in your *.dbxinit* file so that they are automatically defined for you. See **"Automatically Executing Commands on Startup"** for more information on the *.dbxinit* file.

Listing Aliases

You can display the definition of aliases using the *alias* command:

alias	Lists all existing aliases.
alias *name*	Lists the alias definition for *name*.

For example, to display the definitions of the predefined aliases "l" and "bp," enter:

```
(dbx) alias l
"list"
(dbx) alias bp
"stop in"
```

Creating Command Aliases

You can use the *alias* command to define new aliases:

alias *name command*	Defines *name* as an alias for *command*.
alias *name "string"*	Defines *name* as an alias for *string*. With this form of the *alias* command, you can provide command arguments in the alias definition.
alias *name*(*arg1* [, ... *argN*]) *"string"*	Defines *name* as an alias for *string*. *arg1* through *argN* are arguments to the alias, appearing in the *string* definition. When you use the alias, you must provide values for the arguments, which *dbx* then substitutes in *string*.

The simplest form of an alias is to redefine a *dbx* command with a short alias. Many of the predefined *dbx* aliases fall into this category: "a" is an alias for the *assign* command, "s" is an alias for the *step* command. When you use one of these aliases, *dbx* simply replaces it with the command for which it is an alias. Any arguments that you include on the command line are passed to the command.

For example, if you to create "gf" as an alias for the *givenfile* command, enter:

```
(dbx) alias gf givenfile
(dbx) alias gf
"givenfile"
(dbx) gf
Current givenfile is test
(dbx) gf test2
Process 22545 (test) terminated
```

```
Executable /usr/var/tmp/dbx_examples/test2
(dbx) gf
Current givenfile is test2
```

More complex alias definitions require more than the name of a command. In these cases, you must enclose the entire alias definition string in double quotation marks. For example, you can define a brief alias to print the value of a variable that you commonly examine. Note that you must use the escape character (\) to include the double quotation marks as part of the alias definition. For example:

```
(dbx) alias pa "print \"a =\", a"
(dbx) alias pa
"print "a =", a"
(dbx) pa
a = 3
```

You can also define an alias so that you can pass arguments to it, much in the same way that you can provide arguments in a C language macro definition. When you use the alias, you must include the arguments. *dbx* then substitutes the values that you provide in the alias definition.

To illustrate this, consider the following alias definition:

```
(dbx) alias p(arg1, arg2, arg3, arg4) "print '|arg1|arg2|arg3|arg4|'"
(dbx) alias p
(arg1, arg2, arg3, arg4)"print '|arg1|arg2|arg3|arg4|'"
```

The "p" alias takes four arguments and prints them surrounded by vertical bars (|). For example:

```
(dbx) p(1,2,3,4)
|1|2|3|4|
(dbx) p( first, second, 3rd,4)
| first| second| 3rd|4|
```

In the previous example, *dbx* retains any spaces that you enter when calling an alias.

You can also omit arguments when calling an alias as long as you include the commas as argument separators in the alias call:

```
(dbx) p(a,,b,c)
|a||b|c|
(dbx) p(,first missing, preceding space,)
||first missing| preceding space||
(dbx) delete
delete
```

Removing Command Aliases

The *unalias* command removes the alias you provide as an argument. For example, to remove the "pa" alias defined in the previous section, enter:

```
(dbx) unalias pa
```

You can remove any of the predefined *dbx* aliases; however, these aliases are restored the next time you start *dbx*.

Alias Example

An easy way to follow linked lists is to use aliases and casts. This example shows how to construct an alias that follows a simple linked list with members defined by the following structure:

```
struct list { struct list *next; int value; };
```

In this example, a *dbx* variable called *$p* is used as a pointer to a member of the linked list. You can define an alias called "foll" to print the contents of the list member to which *$p* currently points and then advance to the next list member. Because the command is too long to fit onto one line, this example uses the backslash character (\) to continue the command on a second line:

```
(dbx) alias foll "print *(struct list *)$p ; \
set $p = (long)((struct list *)($p))->next"
```

Casting *$p* to an integer type when following the link (the second assignment in the alias) is essential. If omitted, *dbx* may leave the *$p* reference symbolic and if so, goes into an infinite loop. (Type **`<Ctrl-c>`** to interrupt *dbx* if it gets into the infinite loop.)

Before using this alias, you must set *$p* to point at the first list member. In this example, assume that the program variable *top* points to the first list member. Then you can use the "foll" alias to follow the linked list, printing the contents of each member as you proceed:

```
(dbx) set $p = top
(dbx) foll
struct list {
 next = 0x7fffc71c
 value = 57
}
(dbx) foll
struct list {
 next = 0x7fffc724
 value = 3
}
(dbx) foll
struct list {
 next = 0x7fffc72c
 value = 12
}
```

Recording and Playing Back *dbx* Input and Output

dbx allows you to play back your input and record *dbx*'s output. *dbx* saves the information that you capture in files, which allows you to create command scripts that you can use in subsequent *dbx* sessions.

Recording Input

Use the *record input* command to start an input recording session. Once you start an input recording session, all commands to *dbx* are copied to the specified file. If the specified file already exists, *dbx* appends the input to the existing file. You can start and run as many simultaneous *dbx* input recording sessions as you need.

Each recording session is assigned a number when you begin it. Use this number to reference the recording session with the *unrecord* command described in **"Ending a Recording Session".**

After you end the input recording session, use the command file with the *playback input* or *pi* commands to execute again all the commands saved to the file. See **"Playing Back Input".**

For example, to save the recorded input in a file called *script*, enter:

```
(dbx) record input script
[4] record input script (0 lines)
```

If you do not specify a file to *record input*, *dbx* creates a temporary *dbx* file in the */tmp* directory. The name of the temporary file is stored in the *dbx* variable *$defaultin*. You can display the temporary filename using the *print* command as follows:

```
(dbx) print $defaultin
```

Because the *dbx* temporary files are deleted at the end of the *dbx* session, use the temporary file to repeat previously executed *dbx* commands in the current debugging session only. If you need a command file for use in subsequent *dbx* sessions, you must specify the filename when you invoke *record input*. If the specified file exists, the new input is appended to the file.

Ending a Recording Session

To end input or output recording sessions, use the *unrecord* command.

`unrecord` *session1* [`,` *session2* ...]
Turns off the specified recording session(s) and closes the file(s) involved.

`unrecord all` Turns off all recording sessions and closes all files involved.

For example, to stop recording session 4, enter the *dbx* command:

```
(dbx) unrecord 4
```

To stop all recording sessions, enter:

```
(dbx) unrecord all
```

The *dbx status* command does not report on recording sessions. To see whether or not any active recording sessions exist, use the *record* command described in **"Examining the Record State".**

Playing Back Input

Use *playback input* to execute commands that you recorded with the *record input* command. Two aliases exist for *playback input*: *pi* and *source*.) 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 *$pimode* to nonzero, the commands are printed out as they are played back. By default, *$pimode* is set to zero.

Recording Output

Use the *record output* command to start output recording sessions within *dbx*. During an output recording session, *dbx* copies its screen output to a file. If the specified file already exists, *dbx* appends to the existing file. You can start and run as many simultaneous *dbx* output recording sessions as you need.

By default, the commands you enter are not copied to the output file; however, if you set the *dbx* variable *$rimode* to a nonzero value, *dbx* also copies the commands you enter.

Each recording session is assigned a number when you begin it. Use this number to reference the recording session with the *unrecord* command described in **"Ending a Recording Session".**

The *record output* command is very useful when the screen output is too large for a single screen (for example, printing a large structure). Within *dbx*, you can use the *playback output* command (described in **"Playing Back Output"**) to look at the recorded information. After quitting *dbx*, you can review the output file using any IRIX system text viewing command (such as *vi*(1)).

For example, to record the *dbx* output in a file called *gaffa*, enter:

```
(dbx) record output gaffa
```

To record both the commands and the output, enter:

```
(dbx) set $rimode=1
(dbx) record output gaffa
```

If you omit the filename, *dbx* saves the recorded output in a temporary file in */tmp*. The temporary file is deleted at the end of the *dbx* session. To save output for use after the *dbx* session, you must specify the filename when giving the *record output* command. The name of the temporary file is stored in the *dbx* variable *$defaultout*.

To display the temporary filename, type:

```
(dbx) print $defaultout
```

Playing Back Output

The *playback output* command displays output saved with the *record output* command. This command works the same as the *cat*(1) command. If you don't specify a filename, *dbx* uses the current temporary file created for the *record output* command.

For example, to display the output stored in the file *script*, enter:

```
(dbx) playback output script
```

Examining the Record State

The *record* command displays all *record input* and *record output* sessions currently active. For example:

```
(dbx) record
[4] record input /usr/demo/script (12 lines)
[5] record output /tmp/dbxoXa17992 (5 lines)
```

Executing *dbx* Scripts

You can create *dbx* command scripts using an external editor and then execute these scripts using the *pi* or *playback input* command. This is a convenient method for creating and executing automated test scripts.

You can include comments in your command scripts by using a single pound sign (`#`) to introduce a comment. To include a `#` operator (described in **"Operators"**) in a *dbx* script, use two pound signs (for example, `##27`). When *dbx* sees a pound sign in a script file, it interprets all characters between the pound sign and the end of the current line as a comment.

Examining and Changing Data

This section describes how to examine and change data in your program while running it under *dbx*. Topics covered include:

- **"Using Expressions"**
- **"Printing Expressions"**
- **"Using Data Types and Type Coercion (Casts)"**
- **"Displaying and Changing Program Variables"**
- **"Determining the Scope of Variables"**
- **"Displaying Type Declarations"**
- **"Examining the Stack"**
- **"Using Interactive Function Calls"**
- **"C++ Considerations"**

Using Expressions

Many *dbx* commands accept one or more expressions as arguments. Expressions can consist of constants, *dbx* variables, program variables, and operators. This section discusses operators and constants. **"Creating and Removing dbx Variables"** describes *dbx* variables, and **"Displaying and Changing Program Variables"** describes program variables.

Operators

In general, *dbx* recognizes most expression operators from C, Fortran 77, and Pascal. *dbx* also provides some of its own operators. Operators follow the C language precedence. You can also use parentheses to explicitly determine the order of evaluation.

Table 4−2 lists the operators provided by *dbx*.

Table 4−2 *dbx Language Independent Operators*

Operator	Description
not	Unary operator returning false if the operand is true
or	Binary logical operator returning true if either operand is nonzero
xor	Binary operator returning the exclusive−OR of its operands
/	Binary division operator (you can also use "`//`" for division)
div	Binary operator that coerces its operands to integers before dividing
mod	Binary operator returning *op1* modulo *op2*. This is equivalent to the C "`%`" operator
#*exp*	Unary operator returning the address of source line specified by *exp*

"*file*" **#***exp*	Unary operator returning the address of source line specified by *exp* in the file specified by *file*
proc **#***exp*	Unary operator returning the address of source line specified by *exp* in the file containing the procedure *proc*

The **#** operator takes the line number specified by the expression that follows it and returns the address of that source line. If you precede the **#** operator with a filename enclosed in quotation marks, the **#** operator returns the address of the line number in the file you specify. If you precede the **#** operator with the name of a procedure, *dbx* identifies the source file that contains the procedure and returns the address of the line number in that file.

For example, to print the address of line 27 in the current source file, enter:

```
(dbx) print #27
```

To print the address of line 27 in the source file *foo.c* (assuming that *foo.c* contains source that was used to compile the current object file), enter:

```
(dbx) print "foo.c"   #27
```

To print the address of line 27 in the source file containing the procedure *bar* (assuming that *bar* is a function in the current object file), enter:

```
(dbx) print bar #27
```

Note: A pound sign (**#**) introduces a comment in a *dbx* script file. When *dbx* sees a pound sign in a script file, it interprets all characters between the pound sign and the end of the current line as a comment. See **"Executing dbx Scripts"** for more information on *dbx* script files.
To include the **#** operator in a *dbx* script, use two pound signs (for example, `##27`).

Table 4–3 lists the C language operators recognized by *dbx*.

Table 4–3 C Language Operators Recognized by *dbx*

Type	Operators
Unary	`!` `&` `+` `–` `*` `sizeof()`
Binary	`%` `<<` `>>` `==` `<=` `>=` `!=` `<` `>` `&` `&&` `|` `||` `+` `–` `*` `/` `[ ]` `–>` `.`

Note: C does not allow you to use the *sizeof* operator on bit fields. However, *dbx* allows you to enter expressions using the *sizeof* operator on bit fields; in these cases, *dbx* returns the number of bytes in the data type of bit fields (such as *int* or unsigned *int*).
The C language "**^**" exclusive–OR operator is not supported. Use the dbx "**`xor`**" operator instead.

Table 4–4 lists the Pascal operators recognized by *dbx*.

Table 4–4 Pascal Operators Recognized by *dbx*

Type	Operators
Unary	**not** ^ + –
Binary	mod = <= >= <> < > and or + – * / div [ ]

Table 4–5 lists the Fortran 77 language operators recognized by *dbx*. Note that *dbx* does not recognize Fortran logical operators (such as `.or.` and `.TRUE.`).

Table 4–5 Fortran 77 Operators Recognized by *dbx*

Type	Operators
Unary	**+** **–**
Binary	+ – * /

Note: Fortran array subscripting must use "[]" not "()". For example, if `x` is a two–dimension Fortran array, `print x(1,2)` won't work; however, `print x[1,2]` will work. If an array is defined as a Fortran array, refer to it with the standard Fortran subscript ordering.

Constants

You can use both numeric and string constants under *dbx*.

Note: Expressions cannot contain constants defined by *#define* declarations to the C preprocessor.

Numeric Constants

In numeric expressions, you can use any valid integer or floating point constants. By default, *dbx* assumes that numeric constants are in decimal. You can set the default input base to octal by setting the *dbx* variable *$octin* to a nonzero value. You can set the default input base to hexadecimal by setting the *dbx* variable *$hexin* to a nonzero value. If you set both *$octin* and *$hexin* to nonzero values, *$hexin* takes precedence.

You can override the default input type by prefixing "0x" to indicate a hexadecimal constant, or "0t" to indicate a decimal constant. For example, "0t23" is decimal 23 (which equals hexadecimal 0x17), and "0x2A" is hexadecimal 2A (which equals decimal 42).

By default, *dbx* prints the value of numeric expressions in decimal. You can set the default output base to octal by setting the *dbx* variable *$octints* to a nonzero value. You can set the default output base to hexadecimal by setting the *dbx* variable *$hexints* to a nonzero value. If you set both *$octints* and *$hexints* to nonzero values, *$hexints* takes precedence.

String Constants

Most *dbx* expressions cannot include string constants. The *print* and *printf* commands are two of the *dbx* commands that accept string constants as arguments. You can also use the *set* command to assign a string value to a *dbx* variable.

Otherwise, string constants are useful only as arguments to functions that you call interactively. See **"Using Interactive Function Calls"** for information on interactive function calls.

You can use either the double−quote character (`"`) or the single−quote character (`'`) to quote strings in *dbx*.

In general, *dbx* recognizes the following escape sequences in quoted strings (following the standard C language usage):

```
\\ \n \r \f \b \t \' \" \a
```

Printing Expressions

dbx provides the following commands for printing values of expressions:

`print` [*exp1* [, *exp2*, ...]]
Prints the value(s) of the specified expression(s).

`printd` [*exp1* [, *exp2*, ...]]
Prints the value(s) of the specified expression(s) in decimal. (**`pd`** is an alias for **`printd`**. See **"Creating and Removing dbx Aliases"** for more information about *dbx* aliases.)

`printo` [*exp1* [, *exp2*, ...]]
Prints the value(s) of the specified expression(s) in octal. (**`po`** is an alias for **`printo`**.)

`printx` [*exp1* [, *exp2*, ...]]
Prints the value(s) of the specified expression(s) in hexadecimal. (**`px`** is an alias for **`printx`**.)

The variable types are listed in **Table 4−6** .

Table 4−6 Variable Types

Type	Variable Name	Value
signed char	**sc**	0xff
unsigned char	**usc**	0xff

signed short	**`ssh`**	0xffff
unsigned short	**`ush`**	0xffff

Examples include:

```
(dbx)   pd   sc
-1
(dbx)   pd   ssh
-1
(dbx)   px sc
0xff
(dbx)   px ssh
0xffff
(dbx)   pd usc
255
(dbx)   pd ush
65535
```

dbx always prints the bits in the appropriate type. **`pd`** is an exception; it expands signed types with sign extension so the decimal value looks correct.

Another example:

```
(dbx)   print   sc,   usc
'\377' '\377'
```

If *$hexchars* is set, this command displays ’`0xff`’ ‘`0xff`’. (This is a change from releases previous to IRIX 5.2. Previously, the **`px`**, **`po`** cases on signed short expanded to 32 bits, so **`px sc`** printed `0xffffffff`.)

If the printed data type is *pointer*, *dbx* uses the format specified in the *$addrfmt* or *$addrfmt64* predefined *dbx* variable (*$addrfmt64* is used on only 64–bit processes).

`printf` *string* [, *exp1* [, *exp2*, ...]]

Print the value(s) of the specified expression(s) in the format specified by the string, *string*. The *printf* command supports all formats of the IRIX *printf* command except "`%s` ." For a list of formats, see the *printf*(3S) reference page.

Using Data Types and Type Coercion (Casts)

You can use data types for type conversion (casting) by including the name of the data type in parentheses before the expression you want to cast. For example, to convert a character into an integer, use **`(int)`** to cast the value:

```
(dbx) print (int) 'b'
98
```

To convert an integer into a character, use **`(char)`** to cast the value:

```
(dbx) print (char) 67
'C'
```

This is standard C language type casting.

Displaying and Changing Program Variables

You can use the value of program variables in *dbx* expressions. You can also change the value of program variables while running your program under *dbx* control.

Qualifying Variable Names

You can use the same name for different variables multiple times in the same program. For example, you can use a temporary counter named "i" in many different functions.

During program execution, this potential ambiguity presents no problem. The scope of each variable is local; space is allocated for it when the function is called and freed when the function returns.

Under *dbx*, however, you need to be able to distinguish between different variables that may have the same name. To do so, you can qualify a reference to a variable to specify its scope.

dbx qualifies variables with the file (also called module), the procedure, a block, or a structure. You can manually specify the full scope of a variable by separating scopes with periods. For example, in the expression:

```
mrx.main.i
```

i is the variable name, *main* is a procedure in which it appears, and *mrx* is the source file (omitting the file extension) in which the procedure is defined.

To illustrate, consider a C program called *test* that contains a function *compare*. In this example, the variable *i* is declared in both the *main* procedure and the *compare* function:

```
int compare ( int );

main( argc, argv )

int argc;
char **argv;
{
     int i;
...
}

int compare ( arg1, arg2 )
{
     int i;
...
}
```

To trace the value of the *i* that appears in the function *compare*, enter:

```
(dbx) trace test.compare.i
```

To print the value of the *i* that appears in the procedure *main*, enter:

```
(dbx) print test.main.i
```

A leading dot (a period at the beginning of the identifier) tells *dbx* that the first qualifier is not a module (file).

The leading dot is useful when a file and a procedure have the same name. For instance, suppose *mrx.c* contains a function called *mrx*. Further, suppose that *mrx.c* contains a global variable called *mi* and a local variable, also called *mi*. To refer to the global variable, use the qualified form *.mrx.mi*, and to refer to the local variable, use the qualified form *mrx.mrx.mi*.

Variable Scope

You can access the value of a variable only while it is in scope. The variable is in scope only if the block or procedure with which it is associated is active.

After you start your program, whenever your program executes a block or procedure that contains variables, your program allocates space for those variables and they "come into scope." You may access the values of those variables as long as the block or procedure is active. Once the block or procedure ends, the space for those variables is deallocated and you may no longer access their values.

Displaying the Value of a Variable

You can display the value of a program variable using the *print*, *printd*, *printf*, *printo*, and *printx* commands and the *pd*, *po*, and *px* aliases described in **"Printing Expressions".** For example, to print the value of the program variable *total*, enter:

```
(dbx) print total
235
```

The *print* command also displays arrays, structures, and other complex data structures. For example, if *message* is a character array (a string), *dbx* prints the string:

```
(dbx) print message
"Press <Return> to continue."
```

As a more complex example, consider a simple linked list stored as an array of elements, each element consisting of a pointer to the next element and an integer value. If the array is named *list*, print the entire array by entering:

```
(dbx) print array
```

dbx prints the value of each element in the array:

```
{
       [0] struct list {
           next = (nil)
           value = 1034
       }
       [1] struct list {
           next = 0x10012258
           value = 1031
       }
       [2] struct list {
           next = 0x10012270
           value = 1028
       }
       [3] struct list {
           next = 0x10012288
           value = 1025
       }
       [4] struct list {
           next = 0x100122a0
           value = 1022
       }
       [5] struct list {
           next = 0x100122b8
           value = 1019
       }
...
}
```

To print an individual element, enter a command such as:

```
(dbx) print array[5]
struct list {
      next = 0x100122b8
      value = 1019
}
```

No simple method exists for examining a portion of an array with the *print* command. However, if your array consists of simple elements such as integers or floating point values, you can directly examine the contents of memory using the `/`(examine forward) command described in **"Examining Memory and Disassembling Code".**

Suppose a single–precision floating point array is named *float_vals*. To see the six consecutive elements beginning with the fifth element, enter:

```
(dbx) &float_vals[4] / 6f
10012018:  0.25000000000000000 0.20000000298023224 0.16666699945926666
```

```
0.14280000329017639
10012028:  0.12500000000000000 0.11111100018024445
```

Changing the Value of a Variable

The *assign* command changes the value of existing program variables. You can also use the *assign* command to change the value of machine registers, as described in **"Changing Register Values".**

The syntax of the *assign* command is:

`assign` *variable* **`=`** *expression*

Assigns the value of *expression* to the program variable, *variable*.

For example:

```
(dbx) assign x = 27
27
(dbx) assign y = 37.5
37.5
```

If you receive an `incompatible types` error when you try to assign a value to a pointer, use casts to make the assignment work. For example if *next* is a pointer to a structure of type "element," you can assign *next* a null pointer by entering:

```
(dbx) assign *(int *) (&next) = 0
0
(dbx) assign next = 0
(nil)
(dbx) assign next = (struct list*) 0;
(nil)
```

In this example, `nil` denotes that the value of the pointer is 0; `nil` is similar to NULL in the C language.

Conflicts Between Variable Names and Keywords

When naming variables in your program, avoid using any *dbx* keywords. If you have a variable with the same name as a *dbx* keyword and you attempt to use that variable in a *dbx* command, *dbx* reports a syntax error.

If you do have a program variable with the same name as a *dbx* command, you can force *dbx* to treat it as a variable by enclosing the variable in parentheses. For example, if you try to print the value of a variable named *in* by entering the following command, *dbx* displays an error.

```
(dbx) print in

print in
  ^ syntax error
Suggestion: in is a dbx keyword; a revised command is in history.
Type !16 or !! to execute: print (in)
```

The correct way to display the value of *input* is to enter:

```
(dbx) print (in)
34
```

dbx keywords include:

```
all    not
and    or
at     pgrp
div    pid
if     sizeof
in     to
mod    xor
```

By default, *dbx* treats as keywords the following C type keywords:

```
signed     struct
unsigned   union
short      enum
long       double
int        float
char
```

However, if your program is not written in C or C++, you may wish to disable these keywords. The *dbx* variable *$ctypenames* determines whether or not C type keywords are treated as *dbx* keywords. If *$ctypenames* is set to 1 (the default), then C type keywords are treated as *dbx* keywords; if *$ctypenames* is set to 0, they are not.

Case Sensitivity in Variable Names

Whether or not *dbx* is case sensitive when it evaluates program variable names depends on the value of the *dbx* variable *$casesense*.

If *$casesense* is 2 (the default), then the language in which the variable was defined is taken into account (for example, C and C++ are case sensitive while Pascal and Fortran are not). If *$casesense* is 1, case is always checked. If *$casesense* is 0, case is always ignored. Note that file (module) names are always case sensitive since they represent UNIX filenames.

Determining the Scope of Variables

The *which* command allows you to determine the scope of variables. This command is useful for programs that have multiple variables with the same name occurring in different scopes.

The *which* command prints the fully qualified name of the active version of a specified variable. For example, to determine the scope of the variable *i*, enter:

```
(dbx) which i
.foo.foo2.i
```

In the example above, the variable *i* that is currently active is local to the procedure *foo2* that appears in the module *foo* (corresponding to the file *foo.c* in a C language program).

Displaying Type Declarations

The *whatis* command displays the type declaration for a specified variable or procedure in your program.

For example, to display the type declaration for the variable *i*, enter:

```
(dbx) whatis i
int i;
```

The following example illustrates the output of *whatis* for an array of structures:

```
(dbx) whatis array
struct list {
      struct list* next;
      int value;
} array[12];
```

When you provide a procedure name to *whatis*, *dbx* reports the type of the value returned by the procedure and the types of all arguments to the procedure:

```
(dbx) whatis foo
int foo(i)
int i;
(dbx) whatis main
int main(argc, argv)
int argc;
```

```
char** argv;
```

Examining the Stack

Each time your program executes a procedure, the information about where in the program the call was made from is saved on a stack. The stack also contains arguments to the procedure and all of the procedure’s local variables. Each procedure on the stack defines an *activation level*. Activation levels can also consist of blocks that define local variables within procedures.

The activation level determines the scope of many *dbx* commands and expressions. For example, unless you qualify a variable, as described in **"Qualifying Variable Names"** *dbx* assumes that variables you reference are local to the current activation level.

The most recently called procedure or block is numbered 0. The next active procedure (the one that called the current procedure) is numbered 1. The last activation level is always the main program block.

Printing Stack Traces

The *where* command prints stack traces. Stack traces show the current activation levels (procedures) of a program.

For example, consider the following stack trace for a program called *test*:

```
(dbx) where
>  0 foo2(i = 5) ["/usr/var/tmp/dbx_examples/test.c":44, 0x1000109c]
     1 foo(i = 4) ["/usr/var/tmp/dbx_examples/test.c":38, 0x1000105c]
     2 main(argc = 1, argv = 0xffffffad78)
 ["/usr/var/tmp/dbx_examples/test.c":55, 0x10001104]
     3 __start() ["/shamu/lib/libc/libc_64/crt1text.s":137, 0x10000ee4]
```

This program has four activation levels. The most recent, a call of the procedure *foo2*, is numbered 0. The currently selected activation level is 0, indicated by the ">" character.

The stack trace also reports that *foo2* was passed one argument: the value `5` was assigned to the local variable *i*. The trace indicates that the program was stopped at line 44 of the file *test.c*, which translates to machine address 0x1000109c.

The stack trace reports similar information for the next two activation levels in this example. You can see that the function *foo* called *foo2* from line 38 in *test.c*. In turn, *foo* was called by *main* at line 55 of the file *test.c*. Finally, the run−time start−up level was called at line 137 from the file *ctrltext.s*.

If you compile with *-g0* or with no *-g* option, limited symbols are reported. In cases such as this, where detailed symbolic information is not available, the four hexadecimal values returned represent *dbx*’s guess that the function has four integer arguments. The following example illustrates such a case:

```
(dbx) where
>  0 fooexample(0x300000000, 0x4000000ff, 0x5000000ff, 0x0)
 ["/usr/var/tmp/dbx_examples/test3.c":10, 0x10000cf8]
     1 main(0x3, 0x4, 0x5, 0x0) ["/usr/var/tmp/dbx_examples/test3.c":5,
 0x10000cbc]
     2 __start() ["/shamu/lib/libc/libc_64/csu/crt1text.s":137, 0x10000c64]
(dbx) quit
Process 22582 terminated
int fooexample(int,int,int);
int main()
{
         fooexample(3,4,5);
         return 0;
}
int fooexample(int i, int j, int k)
{
         int x = i + j + 3*k;
         return x;
}
```

Moving Within the Stack

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 described in **"Moving to a Specified Procedure".**

The *up* and *down* commands have the following syntax:

up [*num*]	Moves up the specified number of activation levels in the stack. The default is one level.
down [*num*]	Moves down the specified number of activation levels in the stack. The default is one level.

When you change activation levels, your scope changes. For example, unless you qualify a variable, as described in **"Qualifying Variable Names"** , *dbx* assumes that variables you reference are local to the current activation level. Also, *dbx* changes the current source file to the file containing the procedure's source.

Consider examining the stack trace for a program called *test4* and moving up in the activation stack:

```
(dbx) where
>  0 foo2(i = 5) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
     1 foo(i = 4) ["/usr/var/tmp/dbx_examples/foo.c":40, 0x100011d4]
     2 main(argc = 1, argv = 0xffffffad78)
 ["/usr/var/tmp/dbx_examples/test4.c":25, 0x10000fa0]
     3 __start() ["/shamu/lib/libc/libc_64/csu/crt1text.s":137, 0x10000f34]
(dbx) print i
5
(dbx) up
foo:  40  r = foo2(i+1);
```

The current activation level is now the procedure *foo*. As indicated in the output, the variable *i* receives the argument passed to *foo* and is therefore local to *foo*. The variable *i* at this activation level is different from the variable *i* in the *foo2* activation level. You can reference the currently active *i* as "i"; whereas you must qualify the reference to the *i* in *foo2*:

```
(dbx) print i
4
(dbx) print foo2.i
<symbol not found>
```

Moving up one more activation level brings you to the *main* procedure:

```
(dbx) up
main:  25  j = foo(j);
(dbx) file
/usr/var/tmp/dbx_examples/test4.c
```

In this example, the source for *main* is in *test4.c*, whereas the source for *foo* and *foo2* is in *foo.c*; therefore, *dbx* changes the current source file when you move up to the *main* activation level.

dbx resets the source file when you return to the *foo2* activation level:

```
(dbx) down 2
foo2:  46  printf("foo2 arg is %d\n",i);
(dbx) file
/usr/var/tmp/dbx_examples/foo.c
```

Moving to a Specified Procedure

The *func* command moves you up or down the activation stack. You can specify the new activation level by providing either a procedure name or an activation level number.

The syntax for the *func* command is:

`func` {*activation_level* | *procedure*}
Changes the current activation level. If you specify an activation level by number, *dbx* changes to that activation level. If you specify a procedure, *dbx* changes to the activation level of that procedure. If you specify a procedure name and that procedure has called itself recursively, *dbx* changes to the most recently called instance of that procedure.

`func` Displays the name of the procedure corresponding to the current activation level.

When you change your activation level, your scope changes. For example, unless you qualify a variable as described in **"Qualifying Variable Names"** , *dbx* assumes that variables you reference are local to the current activation level. Also, *dbx* changes the current source file to the one containing the procedure's source and the current line to the first line of the procedure.

You can also give the *func* command the name of a procedure that is not on the activation stack, even when your program is not executing. In this case, *dbx* has no corresponding activation level to make current. However, *dbx* still changes the current source file to the one containing the procedure's source and the current line to the first line of the procedure.

For example, consider the following activation stack:

```
(dbx) where
>  0 foo2(i = 5) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
     1 foo(i = 4) ["/usr/var/tmp/dbx_examples/foo.c":40, 0x100011d4]
     2 main(argc = 1, argv = 0xffffffad78)
 ["/usr/var/tmp/dbx_examples/test4.c":25, 0x10000fa0]
     3 __start() ["/shamu/lib/libc/libc_64/csu/crt1text.s":137, 0x10000f34]
```

In this case, you can go to the main activation stack by entering:

```
(dbx) func main
main:  25  j = foo(j);
```

This command changes the current activation level to "2" and changes the current source file to *test4.c*.

If you use the *func* command to go to a function that is not on the activation stack, *dbx* changes only the current source file to the one containing the procedure's source and the current line to the first line of the procedure:

```
(dbx) func bar
     3  {
(dbx) file
/usr/var/tmp/dbx_examples/bar.c
```

Printing Activation Level Information

The *dump* command prints information about the variables in an activation level:

`dump` Prints information about the variables in the current procedure.

`dump` *procedure* Prints information about the variables in the specified procedure. The procedure must be active. Starts searching for procedure at the current activation level as set by the *up* or *down* command. (See **"Moving Within the Stack"** for more information about the *up* and *down* commands.)

`dump .` Prints information about the variables in all procedures in all activation levels.

For example, executing *dump* while in a function called *foo2* appears as:

```
(dbx) dump
foo2(i = 5) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
```

To examine the information for the procedure *main*, enter:

```
(dbx) dump main
main(argc = 1, argv = 0xffffffad78) ["/usr/var/tmp/dbx_examples/test4.c":25,
0x10000fa0]
j = 4
```

```
i = 12
r = <expression or syntax error>
a = 0
total = 0
```

To perform a complete dump of the program's active variables, enter:

```
(dbx) dump .
>  0 foo2(i = 5) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
     1 foo(i = 4) ["/usr/var/tmp/dbx_examples/foo.c":40, 0x100011d4]
r = 0
     2 main(argc = 1, argv = 0xffffffad78)
 ["/usr/var/tmp/dbx_examples/test4.c":25, 0x10000fa0]
j = 4
i = 12
r = <bad operand>
a = 0
total = 0
```

Using Interactive Function Calls

You can interactively call a function in your program from *dbx.*

If the function returns a value, you can use that function in a normal *dbx* expression. For example, consider a function *prime* defined in your program that accepts an integer value as an argument, and returns 1 if the value is prime and 0 if it is not. You can call this function interactively and print the results by entering a command such as:

```
(dbx) print prime(7)
1
```

Using ccall

If your function does not return a value, or if you want to execute a function primarily for its side effects, you can execute the function interactively with the *dbx* command *ccall*:

`ccall` *func***(***arg1*, *arg2*, ... , *argn***)**

This command calls a function with the given arguments. Regardless of the language the function was written in, the call is interpreted as if it were written in C, and normal C calling conventions are used.

Note: Structure and union arguments to a function, and structure and union returns from a function, are not supported.

Functions called interactively honor breakpoints. Thus you can debug a function by setting breakpoints and then calling it interactively.

If you perform a stack trace using the *where* command while stopped in a routine executed interactively, *dbx* displays only those activation levels created by your interactive function call. The activation levels for your active program are effectively invisible. For example, a stack trace looks like this during an interactive function call:

```
(dbx) where
>  0 foo2(i = 9) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
     1 foo(i = 8) ["/usr/var/tmp/dbx_examples/foo.c":40, 0x100011d4]

===== interactive function call =====

     2 foo2(i = 5) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
     3 foo(i = 4) ["/usr/var/tmp/dbx_examples/foo.c":40, 0x100011d4]
     4 main(argc = 1, argv = 0xffffffad78)
 ["/usr/var/tmp/dbx_examples/test4.c":25, 0x10000fa0]
```

```
     5 __start() ["/shamu/lib/libc/libc_64/csu/crt1text.s":137, 0x10000f34]
```

If you stop execution of an interactively called function, you are responsible for eventually "unstacking" the call and returning from the function call. To unstack a call, you can complete the call using *dbx* commands such as *cont*, *resume*, *next*, or *step* as many times as necessary. If you *run* or *rerun* your program, *dbx* automatically unstacks all interactive function calls.

Using clearcalls

Another way to unstack an interactive function call is to execute the *clearcalls* command, which clears all stopped interactive calls.

```
(dbx) clearcalls
```

When stopped or faulted within one or more nested interactive calls, the *clearcalls* command removes these calls from the stack and returns the program to its regular callstack. This command is useful when a segmentation fault, infinite loop, or other fatal error is encountered within the interactive call.

When stopped in an interactive call, the call stack displayed by *where* shows the following line at the end of each stack of interactive call instantiation.

```
==== interactive function call ====
```

For example, if the procedure *foo()* is interactively called from *main()*, you see the following stack:

```
>  0 foo2(i = 9) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
     1 foo(i = 8) ["/usr/var/tmp/dbx_examples/foo.c":40, 0x100011d4]

===== interactive function call =====

     2 foo2(i = 5) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
     3 foo(i = 4) ["/usr/var/tmp/dbx_examples/foo.c":40, 0x100011d4]
     4 main(argc = 1, argv = 0xffffffad78)
 ["/usr/var/tmp/dbx_examples/test4.c":25, 0x10000fa0]
     5 __start() ["/shamu/lib/libc/libc_64/csu/crt1text.s":137, 0x10000f34]
```

Nesting Interactive Function Calls

You can also nest interactive function calls. In other words, if you have one or more breakpoints in a function, and you call that function repeatedly, each interactive call is stacked on top of the previous call. Breakpoints in a function affect all nesting levels, so you cannot have different breakpoints at different nesting levels.

The *where* command shows the entire stack trace from which you can determine the nesting depth. The following example has two nesting levels.

```
(dbx) where
>  0 foo2(i = 17) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
     1 foo(i = 16) ["/usr/var/tmp/src/dbx_examples/foo.c":40, 0x100011d4]

===== interactive function call =====

     2 foo2(i = 9) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
     3 foo(i = 8) ["/usr/var/tmp/dbx_examples/foo.c":40, 0x100011d4]

===== interactive function call =====

     4 foo2(i = 5) ["/usr/var/tmp/dbx_examples/foo.c":46, 0x10001214]
     5 foo(i = 4) ["/usr/var/tmp/dbx_examples/foo.c":40, 0x100011d4]
     6 main(argc = 1, argv = 0xffffffad78)
 ["/usr/var/tmp/src/dbx_examples/test4.c":25, 0x10000fa0]
     7 __start() ["/shamu/lib/libc/libc_64/csu/crt1text.s":137, 0x10000f34]
```

To set a conditional breakpoint, for example, type:

```
(dbx) stop in foo if j == 7
Process     0: [3] stop in foo if j==7
```

If *j* is not within the scope of *foo*, then you will receive an error message if you attempt to call *foo* interactively. To prevent this, disable or delete any such breakpoints, conditional commands, or traces before executing the interactive function call.

C++ Considerations

Debugging a program written in C++ is somewhat different from debugging programs written in other languages. This section describes these differences.

Accessing C++ Member Variables

Typically you use standard C++ syntax to access member variables of objects. For example, if the string *_name* is a member variable of the object *myWindow*, you can print its value by entering:

```
(dbx) print myWindow._name
0x1001dc1c = "MenuWindow"
```

To display a static member variable for a C++ class, you must specify the variable with the class qualifier. For example, to print the value of the static member variable *costPerShare* of the class *CoOp*, enter:

```
(dbx) print CoOp::costPerShare
25.0
```

Referring to C++ Functions

For the purpose of *dbx* debugging, functions in C++ programs fall into three general categories:

Member functions	Refers to member functions using the syntax *classname::functionname*. For example, refers to the member function *foo* in the class *Window* as *Window::foo*.
Global C++ functions	Refers to global functions using the syntax *::functionname*. For example, refers to the global function *foo* as *::foo*.
Non−C++ functions	Refers to non−C++ functions using the syntax *functionname*. For example, refers to the function *printf* as *printf*.

A restriction to keep in mind when using *dbx* with C++ is that you cannot distinguish between overloaded functions. For example, consider two functions:

```
print(int);
print(float);
```

The following command sets a breakpoint in both functions:

```
(dbx) stop in ::print
```

The following contrived example illustrates various possibilities:

```
#include <stdio.h>
class foo {
     int n;
     public:
     foo() {n = 0;}
     foo(int x);
     int bar();
     int bar(int);
};
```

```
int foo:: bar()
{
     return n;
}

int foo:: bar(int x)
{
     return n + x;
}

foo::foo(int x)
{
     n = x;
}

int square(int x)
{
     return x*x;
}

main()
{
     foo a;
     foo b = 11;
     int x = a.bar();
     int y = b.bar(x) + square(x);
     printf("y = %d\n", y);
}
```

If you enter:

```
(dbx) stop in foo::foo
```

dbx stops execution in the constructor for the variable *b*; *dbx* may stop in the constructor for the variable *a* (the ability to stop in an inline function may not yet be fully implemented).

If you enter:

```
(dbx) stop in foo::bar
```

dbx stops execution both when *a.bar* is called and when *b.bar* is called, because *dbx* is unable to distinguish between the overloaded functions.

To stop in *square*, enter:

```
(dbx) stop in ::square
```

To stop in *printf* (a C function), enter:

```
(dbx) stop in printf
```

Controlling Program Execution

A program typically runs until it exits or encounters an unrecoverable error. You can use *dbx*, however, to stop a program under various conditions, step through your program line by line, stop execution on receiving a signal, and execute conditional commands based on your program's status.

This section covers:

- **"Setting Breakpoints"**
- **"Continuing Execution After a Breakpoint"**
- **"Tracing Program Execution"**
- **"Writing Conditional Commands"**

- **"Managing Breakpoints, Traces, and Conditional Commands"**
- **"Using Signal Processing"**
- **"Stopping at System Calls"**
- **"Stepping Through Your Program"**
- **"Starting at a Specified Line"**

Setting Breakpoints

Breakpoints allow you to stop execution of your program. Breakpoints can be *unconditional*, in which case they always stop your program, or *conditional*, in which case they stop your program only if a test condition that you specify is true.

Note: All breakpoints halt program execution *before* executing the line on which they are set. Therefore, if you want to examine the effects of a line of code, you should set the breakpoint on the line of code following the one whose effects you want to study.

Each breakpoint is assigned a number when you create it. Use this number to reference a breakpoint in the various commands provided for manipulating breakpoints (for example, *disable*, *enable*, and *delete*, all described in **"Managing Breakpoints, Traces, and Conditional Commands"**).

Setting Unconditional Breakpoints

To set an unconditional breakpoint, you simply specify the point at which you want to stop program execution, using one of the following forms of the *stop* command:

`stop at` Sets a breakpoint at the current source line.

`stop at` *line* Sets a breakpoint at the specified source line in the current source file.

`stop in` *procedure*
Sets a breakpoint to stop execution upon entering the specified procedure.

`stop at` *file:line*
Sets a breakpoint in the specified file at the specified line.

Caution: If your program has multiple source files, be sure to set the breakpoint in the correct file. To do so, you can explicitly set the source file using *dbx*'s *file* command (see **"Changing Source Files"**), or you can use the *func* command to go to a source file containing a specified function (see **"Moving to a Specified Procedure"**).

Setting Conditional Breakpoints

An unconditional breakpoint is the simplest type of breakpoint; your program stops every time it reaches a specified place. On the other hand, a *conditional* breakpoint stops your program only if a condition that you specify is true. The two conditions that you can test are:

- Has the value of a variable or other memory location changed?
- Is a test expression true?

Stopping If a Variable or Memory Location Has Changed

By including a *variable* clause in your *stop* command, you can cause *dbx* to stop if the value of a variable or the contents of a memory location has changed.

If you provide only a variable name in your variable clause, the breakpoint stops your program if the value of the variable has changed since the last time *dbx* checked it. If instead of a variable name, you provide an expression of type pointer, *dbx* checks the data pointed to. If the data pointed to is a structure, *dbx* checks that structure. If you provide an expression that's not of type pointer, *dbx* evaluates the expression and uses the result as an address in memory. The breakpoint stops your program if the contents of the memory location (32 bits) has changed since the last time *dbx* checked it.

The points at which *dbx* checks the value of a variable or memory location depend on the command that you use to set the breakpoint:

`stop` [*expression*|*variable*]
Inspects the value before executing each source line. If the expression is of type

If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address). For example, consider the command:

```
stop (struct s*) 0x12345678
```

This command checks the contents of the structure located at 0x12345678.

stop [*expression*|*variable*] **at** *line*

Inspects the value at the given source line. Stops if the value has changed.
If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

stop [*expression*|*variable*] **in** *procedure*

Inspects the value at every source line within a given procedure. Stops if the value has changed.
If the expression is of type pointer, look at the data pointed to and watch until it changes.
If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

Using Fast Watchpoints

You can use fast watchpoints with the *stop* command. A fast watchpoint watches a specified variable or memory address without severely impacting the performance of the program being debugged.

In previous versions of *dbx*, the debugger had to single−step the process being debugged and check if the value of a variable had changed after each instruction. With fast watchpoints, the debugger uses a hardware virtual memory *write protect mechanism* to allow the program to run freely until the variable being watched changes. The program being debugged stops only when the virtual memory page containing the variable is written to. If the value of the variable being watched does not changed, *dbx* continues the execution of the process. If a *write* modifies a watched variable, *dbx* notifies you of the change.

Consider a small program that contains a global variable called `global`:

```
stop global
```

This command causes the program to stop if the value of the variable `global` changes. The program runs virtually at full speed until `global` gets assigned a new value. Similarly, consider the command:

```
stop 0x100100
```

This command stops when the 32− bit integer residing at address `0x100100` is modified, and runs at nearly full speed until the value changes. This form of the *stop* command is useful for watching the contents of anonymous memory, such as the memory returned by *malloc()*.

dbx still needs to use the single−step approach if the *stop* command contains an expression to watch, such as in `stop if global == 1`. The performance of the debugged program can be greatly enhanced by including a variable to watch in the *stop* command.

For example, the previous *stop* command can be expressed equivalently as `stop global if global == 1`. This instructs the debugger to check only the expression `global == 1` if the value of `global` changes. For situations where the expression does not depend upon a particular variable getting modified such as `stop if global == x * 3`, the single−step approach is the only way to achieve the desired behavior.

Stopping If a Test Expression Is True

By including a test clause in your *stop* command, you can cause *dbx* to stop if the value of an expression is true. You can use any valid numerical expression as a test. If the result of the expression is nonzero, the expression is true and the test is successful.

The point at which *dbx* evaluates the test expression depends on the command that you use to set the breakpoint:

stop if *expression*

very slow if you choose this type of conditional breakpoint.

`stop at` *line* **`if`** *expression*
Evaluates the expression at the given line.

`stop in` *procedure* **`if`** *expression*
Evaluates the expression at every source line within a given procedure.

Conditional Breakpoints Combining Variable and Test Clauses

You can create conditional breakpoints that combine both variable and test clauses. In these cases, the overall test evaluates to true only if both clauses are true.

The following forms of the *stop* command combine both the variable and test clauses:

`stop` [*expression1*|*variable*] **`if`** *expression2*
Tests both conditions before executing each source line. Stops if both conditions are true.
If *expression1* is of type pointer, look at the data pointed to and watch until it changes.
If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

`stop` [*expression1*|*variable*] **`at`** *line* **`if`** *expression2*
Tests both conditions at the given source line. Stops if both conditions are true.
If *expression1* is of type pointer, look at the data pointed to and watch until it changes.
If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

`stop` [*expression1*|*variable*] **`in`** *procedure* **`if`** *expression2*
Tests both conditions at every source line within a given procedure. Stops if both conditions are true.
If *expression1* is of type pointer, look at the data pointed to and watch until it changes.
If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

Continuing Execution After a Breakpoint

The *cont* command allows you to continue execution after any type of breakpoint. In its simplest form, program execution continues until the end of the program or until another breakpoint is reached. You can also tell *dbx* to continue your program until it reaches a given line or procedure; this is similar to setting a temporary, "one–shot" breakpoint and then continuing.

The syntax of the *cont* command is:

`cont` Continues execution with the current line.

`cont` {**`at`** | **`to`**} *line*
Sets a temporary breakpoint at the specified source line, then resumes execution with the current line. When your program reaches the breakpoint at *line*, *dbx* stops your program and deletes the temporary breakpoint. The keywords *at* and *to* are equivalent.

`cont in` *procedure*
Sets a temporary breakpoint to stop execution upon entering the specified procedure, then resumes execution with the current line. When your program reaches the breakpoint in *procedure*, *dbx* stops your program and deletes the temporary breakpoint.

If your program stopped because *dbx* caught a signal intended for your program, then *dbx* will send that signal to your program when you continue execution. You can also explicitly send a signal to your program when you continue execution. Sending signals to your program upon continuation is discussed in **"Continuing After Catching a Signal".**

When you debug multiprocess programs, the *resume* command can be more helpful than the *cont* command. Refer to **"Resuming a Suspended Process"** for more information about the *resume* command.

Tracing Program Execution

The *trace* command allows you to observe the progress of your program as it executes. With it, you can print:

- values of variables at specific points in your program or whenever variables change value
- parameters passed to and values returned from functions

Each trace is assigned a number when you create it. Use this number to reference the trace in the various commands provided for manipulating traces (for example, *disable*, *enable*, and *delete*, all described in **"Managing Breakpoints, Traces, and Conditional Commands"**).

The syntax of the *trace* command is:

`trace` *variable* Whenever the specified variable changes, *dbx* prints the old and new values of that variable.

`trace` *procedure* Prints the values of the parameters passed to the specified procedure whenever your program calls it. Upon return, *dbx* prints the return value.

`trace` [*expression*|*variable*] **`at`** *line*

Whenever your program reaches the specified line, *dbx* prints the value of the variable if its value has changed.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

`trace` [*expression*|*variable*] **`in`** *procedure*

Whenever the variable changes within the procedure, *dbx* prints the old and new values of that variable.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

`trace` [*expression1*|*variable*] **`at`** *line* **`if`** *expression2*

Prints the value of the variable (if changed) whenever your program reaches the specified line and the given expression is true.

If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

`trace` [*expression1*|*variable*] **`in`** *procedure* **`if`** *expression2*

Whenever the variable changes within the procedure that you specify, *dbx* prints the old and new values of that variable, if the given expression is true.

If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

To examine the parameters passed to and values returned from a function, you can trace that function. For example, if the function name is *foo*, set the trace by entering:

```
(dbx) trace foo
```

When you execute your program, *dbx* prints the values of the parameters passed to *foo* whenever your program calls it. Upon return from *foo*, *dbx* prints the return value:

```
(dbx) run
[3] calling foo(text = 0x10000484 = "Processing...\n", i = 4) from function
 main
[4] foo returning -1 from foo
```

In the example shown above, *foo* receives two parameters: a character string variable named *text* containing the value "Processing...\n" and an integer variable named *i* containing the value 4. The trace also indicates that *foo* returns a value of −1.

You can also examine a variable as it changes values. For example, you can monitor the value of a string variable named *curarg* as you use it to process an argument list. To set the trace, enter:

```
(dbx) trace curarg
Process 2395: [6] trace .test.main.curarg in main
```

When you set a trace on a variable, examine the confirmation that *dbx* prints. If you use the same variable name in multiple functions in your program, *dbx* may not set the trace on the variable that you want. If *dbx* sets the trace on an incorrect variable, delete the trace and set a new trace using a qualified variable format as described in **"Qualifying Variable Names".** For more information on deleting traces, see **"Deleting Breakpoints, Traces, and Conditional Commands".**

So, in this example, if you use the variable *curarg* in both *main* and a function called *arg_process*, and you want to trace the *curarg* in *arg_process*, first delete this trace and then set a new trace:

```
(dbx) delete 6
(dbx) trace arg_process.curarg
Process 2395: [7] trace .test.arg_process.curarg in arg_process
```

When you execute your program, whenever *curarg* changes, *dbx* prints its old and new values:

```
(dbx) run
[7] curarg changed before [arg_process: line 53]:
     new value = (nil);
[7] curarg changed before [arg_process: line 86]:
     old value = 0;
     new value = 0x7fffc7e5 = "-i";
[7] curarg changed before [arg_process: line 86]:
     old value = 2147469285;
     new value = 0x7fffc7eb = "names.out";
[7] curarg changed before [arg_process: line 86]:
     old value = 2147469291;
     new value = 0x7fffc7f5 = "names.in";
```

Writing Conditional Commands

A conditional command created with the *when* command is similar to a breakpoint set with the *stop* command, except that rather than stopping when certain conditions are met, *dbx* executes a list of commands. The command list can consist of any *dbx* commands, separated by semicolons if you include more than one command in the command list. Additionally, you can use the keyword *stop* in the command list to stop execution, just like a breakpoint.

Each conditional command is assigned a number when you create it. You use this number to reference the conditional command in the various commands provided for manipulating conditional commands (for example, *disable*, *enable*, and *delete*, all described in **"Managing Breakpoints, Traces, and Conditional Commands"**).

The syntax of the *when* command is:

when [*expression*|*variable*] **{** *command–list* **}**

Inspects the value before executing each source line. If it has changed, executes the command list.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

when [*expression*|*variable*] **at** *line* **{** *command–list* **}**

Inspects the value at the given source line. If it has changed, executes the command list.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

when [*expression*|*variable*] **in** *procedure* **{** *command–list* **}**

Inspects the value at every source line within a given procedure. If it has changed, executes the command list.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

when if *expression* **{** *command–list* **}**

Evaluates the expression before executing each source line. If it is true, executes the command list. Note that execution is slow if you choose this type of conditional command execution.

when at *line* **if** *expression* {*command–list*}

Evaluates the expression at the given line. If it is true, executes the command list.

when in *procedure* **if** *expression* {*command–list*}

Evaluates the expression at every source line within a given procedure. If it is true, executes the command list.

when [*expression1*|*variable*] **if** *expression2* {*command–list*}

Checks if the value of the variable has changed. If it has changed and the expression is true, executes the command list.

If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

when [*expression1*|*variable*] **at** *line* **if** *expression2* {*command–list*}

Checks if the value of the variable has changed each time the line is executed. If the value has changed and the expression is true, executes the command list.

If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

when [*expression1*|*variable*] **in** *procedure* **if** *expression2* {*command–list*}

Checks if the value of variable has changed at each source line of the given procedure. If the value has changed and the expression is true, executes the command list.

If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

Managing Breakpoints, Traces, and Conditional Commands

dbx provides commands that allow you to disable, enable, delete, and examine the status of the breakpoints, traces, and conditional commands that you set in your programs.

Each breakpoint, trace, and conditional command is assigned a number when you create it. Use these numbers as identifiers in the various commands provided for manipulating these debugging controls.

Listing Breakpoints, Traces, and Conditional Commands

The *status* command lists all of the breakpoints, traces, and conditional commands that you have set and indicates whether they are enabled or disabled.

For example, consider executing the following commands while debugging a program called *test*:

```
(dbx) stop in foo
Process     0: [3] stop in foo
(dbx)   r
Process 22631 (test) started
[3] Process 22631 (test) stopped at [foo:38 ,0x10001050]
    38  r = foo2(i+1);
(dbx)   trace total
Process 22631: [4] trace total in foo
(dbx)   when at 60 {print i,j }
Process 22631: [5] when at "/usr/var/tmp/dbx_examples/test.c":60 { print i, j
 }
```

If you enter *status*, you see the following:

```
(dbx) status
Process 22631: [3] stop in foo
Process 22631: [4] trace total in foo
Process 22631: [5] when at "/usr/var/tmp/dbx_examples/test.c":60 { print i, j
 }
```

Disabling Breakpoints, Traces, and Conditional Commands

The *disable* command allows you to temporarily disable a breakpoint, trace, or conditional command so that it is inoperative and has no effect on program execution. *dbx* remembers all information about a disabled breakpoint, trace, or conditional command, and you may enable it using the *enable* command described in **"Enabling Breakpoints, Traces, and Conditional Commands".**

The syntax of the *disable* command is:

`disable` *item* [, *item* ...]

Disables the specified breakpoint(s), trace(s), or conditional command(s). This command has no effect if the item you specify is already disabled.

For example, to disable the conditional command set in **"Listing Breakpoints, Traces, and Conditional Commands"** enter:

```
(dbx) disable 4
```

If you enter *status*, you see the following:

```
(dbx) status
Process 22631: [3] stop in foo
Process 22631: [4] (disabled) trace total in foo
Process 22631: [5] when at "/usr/var/tmp/dbx_examples/test.c":60 { print i, j
```

Enabling Breakpoints, Traces, and Conditional Commands

The *enable* command reverses the effects of a *disable* command: The breakpoint, trace, or conditional command that you specify is enabled and once again affects the execution of your program. The syntax of the *enable* command is:

`enable` *item* [, *item* ...]

Enables the specified breakpoint(s), trace(s), or conditional command(s).

For example, to enable the conditional command disabled in **"Disabling Breakpoints, Traces, and Conditional Commands"** enter:

```
(dbx) enable 4
```

Executing the *status* command shows that the condition command is now enabled:

```
(dbx) status
Process 22631: [3] stop in foo
Process 22631: [4] trace total in foo
Process 22631: [5] when at "/usr/var/tmp/dbx_examples/test.c":60 { print i, j
```

Deleting Breakpoints, Traces, and Conditional Commands

The *delete* command allows you to delete breakpoints, traces, and conditional commands:

`delete` {*item* [, *item* ...] | **`all`**}

Deletes the item or items specified. If you use the keyword *all* instead of listing individual items, *dbx* deletes all breakpoints, traces, and conditional commands.

For example, to delete the breakpoint and trace set in **"Listing Breakpoints, Traces, and Conditional Commands"** enter:

```
(dbx) delete 3, 4
```

If you enter *status*, you see the following:

```
(dbx) status
Process 22631: [5] when at "/usr/var/tmp/dbx_examples/test.c":60 { print i, j
 }
```

To delete all breakpoints, traces, and conditional commands, enter:

```
(dbx) delete all
```

Using Signal Processing

dbx can detect any signals sent to your program while it is running and, at your option, stop the program.

Catching and Ignoring Signals

With the *catch* command, you can instruct *dbx* to stop your program when it receives any specified signal. The *ignore* command undoes the effects of a *catch* command. The *catch* and *ignore* commands have the following syntax:

`catch` {*signal* | **`all`**}
Instructs *dbx* to stop your program whenever it receives the specified signal. If you use the keyword *all* rather than giving a specific signal, *dbx* catches all signals.

`ignore` {*signal* | **`all`**}
Instructs *dbx* to ignore the specified signal. All ignored signals are passed to your program normally. If you use the keyword *all* rather than giving a specific signal, *dbx* ignores all signals.

`catch` Prints a list of all signals caught.

`ignore` Prints a list of all signals ignored.

You can use the signal names and numbers as listed in the *signal*(2) reference page. You can also abbreviate the signal names by omitting the "SIG" portion of the name. You can use uppercase or lowercase for the signal names.

Note: Because "int" (in lowercase) is a *dbx* keyword, you cannot use it as an abbreviation for the SIGINT signal. You must use uppercase ("INT"), the full signal name ("SIGINT" or "sigint"), or the signal number ("2"). SIGINT is the only signal name with such a restriction.

If you instruct *dbx* to catch a signal, whenever that signal is directed to your program, *dbx* intercepts it and stops your program. Your program does not see this signal until you continue your program with the *cont* command. If your program did not declare a signal handler for a signal, your program does not see the signal when *dbx* continues it.

If you issue a SIGINT signal at the keyboard (usually by pressing **`<Ctrl-C>`**) while you are running an application under *dbx*, what happens depends on the circumstances:

- If the process is in the same IRIX process group as *dbx*, the interrupt signal is sent to both *dbx* and the process. Both *dbx* and the process stop running. You are left at the *dbx* command line.
- If the process was added with **`addproc`**, **`dbx -P`**, or **`dbx -p`**, it is not in the same IRIX process group as *dbx*. In this case, the signal interrupt is sent to *dbx* but not to the process. *dbx* stops running, but the process continues to run. Use the *showproc* command to see whether the process is still running. Then use the *suspend* command to stop the process.

Continuing After Catching a Signal

The *cont* command allows you to continue execution after catching a signal. You can also use the *cont* command to specify a different signal to send to your program than the one that *dbx* caught. Using the same syntax, you can also send a signal to your program when you continue, even if your program did not stop because of a caught signal.

Use the following forms of the *cont* command when handling signals. In each case, if you do not provide a signal, but your program stopped because *dbx* caught a signal intended for your program, then *dbx* sends that signal to your program when you continue execution:

`cont` [*signal*] Continues execution with the current line and sends the specified signal to your program.

`cont` [*signal*] {**`at`** | **`to`**} *line*
Sets a temporary breakpoint at the specified source line, then resumes execution with the current line and sends the specified signal to your program.

`cont` [*signal*] **`in`** *procedure*
Sets a temporary breakpoint to stop execution upon entering the specified procedure, then resumes execution with the current line and sends the specified signal to your program.

For example, if your program stopped because *dbx* caught a SIGINT signal, *dbx* will automatically send

For example, if your program stopped because *dbx* caught a SIGINT signal, *dbx* will automatically send that signal to your program, if you enter:

```
(dbx) cont
```

Suppose you have a procedure called *alarm_handler* to handle an alarm signal sent to your program. If you want to test this procedure by single−stepping through it, you can execute the following command:

```
(dbx) cont SIGALRM in alarm_handler
```

This sets a temporary breakpoint to stop your program upon entering *alarm_handler*, continues execution of your program, and sends a SIGALRM signal to your program. Your program then enters the *alarm_handler* procedure and stops. You can then single−step through the procedure and observe execution.

Debugging a program that attempts to catch signals can be awkward if you also catch the signal in *dbx*. For example, if your program wants to catch SIGFPEs and you issue the command:

```
(dbx) catch sigfpe
```

Then, after *dbx* catches the signal, you must execute the command:

```
(dbx) ignore sigfpe
```

This prevents *dbx* from catching the signal again when you resume execution of your program.

If you want to catch further instances of that signal in your program, you must regain *dbx* control (the best way is to set a breakpoint before executing the *cont*) to re−execute the *catch sigfpe* command.

Stopping at System Calls

Because system calls are part of the operating system and their source is generally not available for debugging purposes, you cannot set breakpoints in system calls using the same method that you use for your program’s procedures. Instead, *dbx* provides the *syscall* command to allow you to stop your program when it executes system calls. With the *syscall* command you can catch (breakpoint) system calls either at the entry to the system call or at the return from the system call.

The syntax of the *syscall* command is:

`syscall catch` {**`call`** | **`return`**} {*system_call* | **`all`**}

Sets a breakpoint to stop execution upon entering (*call*) or returning from (*return*) the specified system call. Note that you can set *dbx* to catch both the call and the return of a system call.

If you use the keyword *all* rather than giving a specific system call, *dbx* catches all system calls.

`syscall ignore` {**`call`** | **`return`**} {*system_call* | **`all`**}

Clears the breakpoint to stop execution upon entering (*call*) or returning from (*return*) the specified system call.

If you use the keyword *all* rather than giving a specific system call, *dbx* clears the breakpoints to stop execution upon entering (*call*) or returning from (*return*) all system calls.

`syscall catch` [{**`call`** | **`return`**}]

Prints a list of all system calls caught upon entry (*call*) or return (*return*). If you provide neither the *call* nor *return* keyword, *dbx* lists all system calls that are caught.

`syscall ignore` [{**`call`** | **`return`**}]

Prints a list of all system calls not caught upon entry (*call*) or return (*return*). If you provide neither the *call* nor *return* keyword, *dbx* lists all system calls that are ignored.

`syscall`

Prints a summary of the catch and ignore status of all system calls. The summary is divided into four sections: 1) caught at call, 2) caught at return, 3) ignored at call, and 4) ignored at return.

Note: The *fork* and *sproc* system calls are treated specially as they invoke new processes. The returns from these system calls are controlled by the *dbx* variables *$promptonfork* and *$mp_program*, not by *syscall*. This is discussed in **"Handling fork System Calls"** and **"Handling sproc System Calls and Process Group Debugging".** The *execv* and *execve* system calls are treated specially as they change a process into a new program. For more information, see **"Handling exec System Calls".**

The system calls are listed in */usr/include/sys.s*. *dbx* ignores the case of the system call names in all *syscall* commands; therefore, you can use uppercase or lowercase in these commands.

A particularly useful setting is:

```
(dbx) syscall catch call exit
```

This stops your program upon entry to *exit*. With your program stopped, you can do a stack trace before the termination to see why *exit* was called.

Stepping Through Your Program

Stepping is a process of executing your program for a fixed number of lines and then automatically returning control to *dbx*. *dbx* provides two commands for stepping through lines of code: *step* and *next*.

For both *step* and *next*, *dbx* counts only those source lines that actually contain code; for the purposes of stepping, *dbx* ignores blank lines and lines consisting solely of comments.

The *next* and *step* commands differ in their treatment of procedure calls. When *step* encounters a procedure call, it usually "steps into" the procedure and continues stepping through the procedure (counting each line of source). On the other hand, when *next* encounters a procedure call, it "steps over" the procedure—executing it without stopping but not counting lines in the procedure—and continues stepping through the current procedure.

The following code fragment illustrates the difference between *step* and *next*:

```
 55 foo( arg1, arg2 )
 56 int arg1, arg2;
 57 {
 58       if ( arg1 < arg2 ) {
 ...        ...
 78       return( 0 );
 79 }
 ...
211 x = foo( i, j );
212 y = 2 * x;
```

In this example, if at line 211 you execute a *step* command to advance one line, *dbx* allows the process to proceed to line 58 (the first code line of the *foo* procedure). However, if you execute a *next* command, *dbx* executes line 211—calling *foo*—and advances the process to line 212.

Stepping Using the *step* Command

The format of the *step* command is:

`step` [*integer*]	Executes the specified number of lines of source code, stepping into procedures. If you do not provide an argument, *step* executes one line. If *step* encounters any breakpoints, it immediately stops execution.

By default, *step* steps into only those procedures that are compiled with the debugging option *-g*. Note that this does not include standard library routines because they are not compiled using debugging options.

You can modify this behavior, even force *dbx* to step into procedures not compiled with full debugging information, by changing the value of the *dbx* variable *$stepintoall*. **Table 4–7** summarizes how the value of *$stepintoall* affects *dbx*'s treatment of the *step* command.

Table 4–7 Effect of *$stepintoall* Variable on the *step* Command

***$stepintoall* value**	**Effect on *step* Command**
0 (default)	*step* steps into only those procedures that are compiled with the debugging option *-g*. step steps over all other procedures.

1 or 2	*step* steps into all procedures. Note that when you debug a source file compiled without symbols or compiled with optimization, the line numbers may sometimes jump erratically. Also note that if *dbx* cannot locate a source file, then it cannot display source lines as you step through a procedure.

Stepping Using the *next* Command

The format of the *next* command is:

`next` [*integer*]	Executes the specified number of lines of source code, stepping over procedures. If you do not provide an argument, *next* executes one line. If *next* encounters any breakpoints, even in procedures that it steps over, it immediately stops execution.

Using the *return* Command

If you step into a procedure and then decide you don't want to step through the rest of it, use *return* to finish executing the procedure and return to the calling procedure.

The format of the *return* command is:

`return`	Continues execution until control returns to the procedure that invoked the *return* command.
`return` *proc*	Continues execution until control returns to the named procedure. Execution continues, unless stopped by a breakpoint, until the latest invocation of the procedure named by *proc* at the time the command was issued is reached. Execution doesn't stop at subsequent invocations of the same procedure. The search for the frame to return to starts with the previous frame, because the current frame is skipped in looking for a frame whose name matches *proc*. If execution is stopped for any reason, this command is cancelled.

Starting at a Specified Line

When you continue your program, you typically do so at the place where it stopped using the *cont* command. However, you can also force your program to continue at a different address by using the *goto* command:

`goto` *line*	Begins execution at the specified line. You may not use the *goto* command to resume execution with a line outside of the current procedure.

Debugging Machine Language Code

This section explains how to debug machine language code by:

- **"Examining and Changing Register Values"**
- **"Examining Memory and Disassembling Code"**
- **"Setting Machine-Level Breakpoints"**
- **"Continuing Execution After a Machine-Level Breakpoint"**
- **"Tracing Execution at the Machine Level"**
- **"Writing Conditional Commands at the Machine Level"**
- **"Stepping Through Machine Code"**

Examining and Changing Register Values

Using *dbx*, you can examine and change the hardware registers during execution of your program. **Table 4–8** lists the machine form of the register names and the alternate software names as defined in the include file *regdef.h*.

Table 4–8 Hardware Registers and Aliases

Register	Software Name	Description
$r0	$zero	Always 0
$r1	$at	Reserved for assembler
$r2... $r3	$v0... $v1	Expression evaluations, function return values, static links

$r4... $r7	$a0... $a3	Arguments
$r8... $r11	$t0... $t7	Temporaries (32 bit)
	$a4... $a7, $ta0... $ta3	Arguments (64 bit)
$r12... $r15	$t4... $t7, $t0... $t3	Temporaries (32 bit)
	$ta0... $ta3	Temporaries (64 bit)
$r16... $r23	$s0... $s7	Saved across procedure calls
$r24... $r25	$t8... $t9	Temporaries
$r26... $r27	$k0... $k1	Reserved for kernel
$r28	$gp	Global pointer
$r29	$sp	Stack pointer
$r30	$s8	Saved across procedure calls
$r31	$ra	Return address
$mmhi		Most significant multiply/divide result register
$mmlo		Least significant multiply/divide result register
$fcsr		Floating point control and status register
$feir		Floating point exception instruction register
$cause		Exception cause register
$d0, $d2, ... $d30		Double precision floating point registers (32 bit)
$d0, $d2, ... $d31		(64 bit)
$f0, $f2, ... $f30		Single precision floating point registers (32 bit)
$f0, $f1, ... $f31		(64 bit)

For registers with alternate names, the *dbx* variable *$regstyle* controls which name is displayed when you disassemble code (as described in **"Examining Memory and Disassembling Code"**). If *$regstyle* is set to 0, then *dbx* uses the alternate form of the register name (for example, "zero" instead of "r0," and "t1" instead of "r9"); if *$regstyle* is anything other than 0, the machine names are used ("r0" through "r31").

Printing Register Values

Use the *printregs* command to print the values stored in all registers.

The base in which the register values are displayed depends on the values of the *dbx* variables *$octints* and *$hexints*. By default, *dbx* prints the register values in decimal. You can set the output base to octal by setting the *dbx* variable *$octints* to a nonzero value. You can set the output base to hexadecimal by setting the *dbx* variable *$hexints* to a nonzero value. If you set both *$octints* and *$hexints* to nonzero values, *$hexints* takes precedence.

To examine the register values in hexadecimal, enter the following:

```
(dbx) set $hexints = 1
(dbx) printregs
r0/zero=0x0      r1/at=0x19050
r2/v0=0x8        r3/v1=0x100120e0
r4/a0=0x4        r5/a1=0xffffffad78
r6/a2=0xffffffad88       r7/a3=0x0
r8/a4=0x10       r9/a5=0x20
r10/a6=0x0       r11/a7=0xfbd5990
r12/t0=0x0       r13/t1=0x0
r14/t2=0x65      r15/t3=0x0
r16/s0=0x1       r17/s1=0xffffffad78
r18/s2=0xffffffad88      r19/s3=0xffffffaf70
r20/s4=0x0       r21/s5=0x0
r22/s6=0x0       r23/s7=0x0
r24/t8=0x0       r25/t9=0x10001034
r26/k0=0x0       r27/k1=0x20
```

```
r28/gp=0x1001a084       r29/sp=0xffffffaca0
r30/s8=0x0      r31/ra=0x1000110c

mdhi=0x0        mdlo=0xe0
cause=0x24      pc=0x10001050
fpcsr=0x0
f0=0.0000000e+00        f1=0.0000000e+00        f2=0.0000000e+00
f3=0.0000000e+00        f4=0.0000000e+00        f5=0.0000000e+00
f6=0.0000000e+00        f7=0.0000000e+00        f8=0.0000000e+00
f9=0.0000000e+00        f10=0.0000000e+00       f11=0.0000000e+00
f12=0.0000000e+00       f13=0.0000000e+00       f14=0.0000000e+00
f15=0.0000000e+00       f16=0.0000000e+00       f17=0.0000000e+00
f18=0.0000000e+00       f19=0.0000000e+00       f20=0.0000000e+00
f21=0.0000000e+00       f22=0.0000000e+00       f23=0.0000000e+00
f24=0.0000000e+00       f25=0.0000000e+00       f26=0.0000000e+00
f27=0.0000000e+00       f28=0.0000000e+00       f29=0.0000000e+00
f30=0.0000000e+00       f31=0.0000000e+00
d0=0.000000000000000e+00        d1=0.000000000000000e+00
d2=0.000000000000000e+00        d3=0.000000000000000e+00
d4=0.000000000000000e+00        d5=0.000000000000000e+00
d6=0.000000000000000e+00        d7=0.000000000000000e+00
d8=0.000000000000000e+00        d9=0.000000000000000e+00
d10=0.000000000000000e+00       d11=0.000000000000000e+00
d12=0.000000000000000e+00       d13=0.000000000000000e+00
d14=0.000000000000000e+00       d15=0.000000000000000e+00
d16=0.000000000000000e+00       d17=0.000000000000000e+00
d18=0.000000000000000e+00       d19=0.000000000000000e+00
d20=0.000000000000000e+00       d21=0.000000000000000e+00
d22=0.000000000000000e+00       d23=0.000000000000000e+00
d24=0.000000000000000e+00       d25=0.000000000000000e+00
d26=0.000000000000000e+00       d27=0.000000000000000e+00
d28=0.000000000000000e+00       d29=0.000000000000000e+00
d30=0.000000000000000e+00       d31=0.000000000000000e+00
```

(Note that there are twice as many floating point registers with 64–bit programs.) You can also use the value of a single register in an expression by typing the name of the register preceded by a dollar sign ($).

For example, to print the current value of the program counter (the *pc* register), enter:

```
(dbx) printx $pc
0x10001050
```

Changing Register Values

In the same way you change the values of program variables, you can use the *assign* command to change the value of registers:

assign *register* **=** *expression*

Assigns the value of *expression* to *register*. You must precede the name of the register with a dollar sign ($).

For example:

```
(dbx) assign $f0 = 3.14159
3.1415899999999999
(dbx) assign $t3 = 0x5a
0x5a
```

By default, the *assign register* command changes the register value in the current activation level, which is a typical operation. To force the hardware register to be updated regardless of the current activation level, use the *$ set $framereg* command.

Examining Memory and Disassembling Code

The forward slash (*/*) and question mark (*?*) commands allow you to examine the contents of memory. Depending on the format you specify, you can display the values as numbers, characters, or disassembled machine code. Note that all common forms of *address* are supported. Some unusual expressions may not be accepted unless enclosed in parentheses, as in (*address*)/*count format.*

The commands for examining memory have the following syntax:

address **/** *count format*	Prints the contents of the specified address, or disassembles the code for the instruction at the specified address. Repeat for a total of *count* addresses in increasing address—in other words, an "examine forward" command. The format codes are listed in **Table 4−9** .
address **?** *count format*	Prints the contents of the specified address or, disassembles the code for the instruction at the specified address. Repeat for a total of *count* addresses in decreasing address—in other words, an "examine backward" command. The format codes are listed in **Table 4−9** .
address **/** *count* **L** *value mask*	Examines *count* 32−bit words in increasing addresses; prints those 32−bit words which, when ORed with *mask*, equals *value*. This command searches memory for specific patterns.
`./`	Repeats the previous examine command with increasing address.
`.?`	Repeats the previous examine command with decreasing address.

Table 4−9 Memory Display Format Codes

Format Code	Displays Memory in the Format
i	print machine instructions (disassemble)
d	print a 16−bit word in signed decimal
D	print a 32−bit word in signed decimal
dd	print a 64−bit word in signed decimal
o	print a 16−bit word in octal
O	print a 32−bit word in octal
oo	print a 64−bit word in octal
x	print a 16−bit word in hexadecimal
X	print a 32−bit word in hexadecimal
xx	print a 64−bit word in hexadecimal
v	print a 16−bit word in unsigned decimal
V	print a 32−bit word in unsigned decimal
vv	print a 64−bit word in unsigned decimal
L	like X but use with val mask
b	print a byte in octal
c	print a byte as character
s	print a string of characters that ends in a null byte
f	print a single−precision real number
g	print a double−precision real number

For example, to display 10 disassembled machine instructions starting at the current address of the program counter, enter:

```
(dbx) $pc/10i
*[main:26, 0x400290]  sw   zero,28(sp)
 [main:27, 0x400294]  sw  zero,24(sp)
 [main:29, 0x400298]  lw  t1,28(sp)
 [main:29, 0x40029c]  lw  t2,32(sp)
 [main:29, 0x4002a0]  nop
 [main:29, 0x4002a4]  slt  at,t1,t2
 [main:29, 0x4002a8]  beq  at,zero,0x4002ec
 [main:29, 0x4002ac]  nop
 [main:31, 0x4002b0]  lw  t3,28(sp)
```

```
[main:31, 0x4002b4] nop
```

To disassemble another 10 lines, enter:

```
(dbx) ./
[main:31, 0x4002b8] addiu t4,t3,1
[main:31, 0x4002bc] sw  t4,28(sp)
[main:32, 0x4002c0] lw  t5,24(sp)
[main:32, 0x4002c4] lw  t6,28(sp)
[main:32, 0x4002c8] nop
[main:32, 0x4002cc] addu t7,t5,t6
[main:32, 0x4002d0] sw  t7,24(sp)
[main:33, 0x4002d4] lw  t8,28(sp)
[main:33, 0x4002d8] lw  t9,32(sp)
[main:33, 0x4002dc] nop
```

To examine ten 32−bit words starting at address 0x7fffc754, and print those whose least significant byte is hexadecimal 0x19, enter:

```
(dbx) 0x7fffc754 / 10 L 0x19 0xff
7fffc758: 00000019
```

Consider a single−precision floating point array named *array*. You can examine the six consecutive elements, beginning with the fifth element, by entering:

```
(dbx) &array[4] / 6f
7fffc748: 0.2500000 0.2000000 0.1666667 0.1428571
7fffc758: 0.1250000 0.1111111
```

Setting Machine−Level Breakpoints

dbx allows you to set breakpoints while debugging machine code just as you can while debugging source code. You set breakpoints at the machine code level using the *stopi* command.

The conditional and unconditional versions of the *stopi* commands work in the same way as the *stop* command described in **"Setting Breakpoints"** with these exceptions:

- The *stopi* command checks its breakpoint conditions on a machine−instruction level instead of a source−code level.
- The *stopi at* command requires an address rather than a line number.

Each breakpoint is assigned a number when you create it. Use this number to reference the breakpoint in the various commands provided for manipulating breakpoints (for example, *disable*, *enable*, and *delete* , all described in **"Managing Breakpoints, Traces, and Conditional Commands"**).

Syntax of the *stopi* Command

The syntax of the *stopi* command is:

stopi at	Sets an unconditional breakpoint at the current instruction.
stopi at *address*	Sets an unconditional breakpoint at the specified address.
stopi in *procedure*	Sets an unconditional breakpoint to stop execution upon entering the specified procedure.
stopi [*expression*|*variable*]	Inspects the value before executing each machine instruction and stops if the value has changed. If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).
stopi [*expression*|*variable*] **at** *address*	Inspects the value when the program is at the given address and stops if the value has changed (for machine−level debugging).

changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

stopi [*expression*|*variable*] **in** *procedure*

Inspects the value at every machine instruction within a given procedure and stops if the value has changed.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

stopi if *expression*

Evaluates the expression before executing each instruction and stops if the expression is true. Note that execution is very slow if you choose this type of conditional breakpoint.

stopi at *address* **if** *expression*

Evaluates the expression at the given address and stops if the expression is true.

stopi in *procedure* **if** *expression*

Evaluates the expression at every instruction within a given procedure and stops if the expression is true.

stopi [*expression1*|*variable*] **if** *expression2*

Tests both conditions before executing each machine instruction. Stops if both conditions are true.

If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

stopi [*expression1*|*variable*] **at** *address* **if** *expression2*

Tests both conditions at the given address (for machine−level debugging). Stops if both conditions are true.

If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

stopi [*expression1*|*variable*] **in** *procedure* **if** *expression2*

Tests the expression each time that the given variable changes within the given procedure.

If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

Note: When you stop execution because of a machine−level breakpoint set by one of the *stopi in* commands, a *where* command at the point of stop may yield an incorrect stack trace. This is because the stack for the function is not completely set up until several machine instructions have been executed. *dbx* attempts to account for this, but is sometimes unsuccessful.

Linking With DSOs

If you link with a DSO, be careful when you use the *stopi at* command. For example, suppose you enter:

```
dbx() stopi at functionx
```

The breakpoint at *functionx* is hit only if the *gp_prolog* instruction is executed. (*gp_prolog* is a short sequence of instructions at the beginning of the routine.) To avoid this problem, use the *stopi in* command:

```
dbx() stopi in functionx
```

If you really want to use *stopi at*, a safe alternative is to disassemble *functionx* and put the breakpoint after the *gp_prolog*. For more information on *gp_prolog*, see the *MIPSpro Assembly Language Programmer's Guide*.

The *tracei at*, *wheni at*, and *conti at* commands described in the following sections also follow this pattern. Use their "*in*" versions to ensure that the function breakpoint is hit.

Continuing Execution After a Machine–Level Breakpoint

The *conti* command continues executing assembly code after a breakpoint. As with the *cont* command, if your program stops because *dbx* catches a signal intended for your program, then *dbx* sends that signal to your program when you continue execution. You can also explicitly send a signal to your program when you continue execution. Signal processing and sending signals to your program is discussed in **"Using Signal Processing".**

The syntax of the *conti* command is:

`conti` [*signal*] Continues execution with the current instruction.

`conti` [*signal*] {`at` | `to`} *address*

Sets a temporary breakpoint at the specified address, then resumes execution with the current instruction. When your program reaches the breakpoint at *address*, *dbx* stops your program and deletes the temporary breakpoint.

`conti` [*signal*] `in` *procedure*

Sets a temporary breakpoint to stop execution upon entering the specified procedure, then resumes execution with the current instruction. When your program reaches the breakpoint in *procedure*, *dbx* stops your program and deletes the temporary breakpoint.

See also **"Linking With DSOs"** for a description on using the *conti in* and *conti at* commands with DSOs.

Tracing Execution at the Machine Level

The *tracei* command allows you to observe the progress of your program while debugging machine code, just as you can with the *trace* command while debugging source code. The *tracei* command traces in units of machine instructions instead of in lines of code.

Each trace is assigned a number when you create it. Use this number to reference the breakpoint in the various commands provided for manipulating breakpoints (for example, *disable*, *enable*, and *delete*, all described in **"Managing Breakpoints, Traces, and Conditional Commands"**).

The syntax of the *tracei* command is:

`tracei` [*expression*|*variable*]

Whenever the specified variable changes, *dbx* prints the old and new values of that variable. (For machine–level debugging.) Note that execution is very slow if you choose this type of trace.
If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

`tracei` *procedure*

This command is equivalent to entering `trace` *procedure*. *dbx* prints the values of the parameters passed to the specified procedure whenever your program calls it. Upon return, *dbx* prints the return value.

`tracei` [*expression*|*variable*] `at` *address*

Prints the value of the variable whenever your program reaches the specified address. (For machine–level debugging.)
If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

`tracei` [*expression*|*variable*] `in` *procedure*

Whenever the variable changes within the procedure that you specify, *dbx* prints the old and new values of that variable. (For machine–level debugging.)
If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

`tracei` [*expression1*|*variable*] `at` *address* `if` *expression2*

Prints the value of the variable whenever your program reaches the specified address and the given expression is true. (For machine–level debugging.)
If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the

`tracei` [*expression1*|*variable*] **`in`** *procedure* **`if`** *expression2*

Whenever the variable changes within the procedure that you specify, *dbx* prints the old and new values of that variable, if the given expression is true. (For machine−level debugging.)
If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

See also **"Linking With DSOs"** for a description on using the *tracei in* and *tracei at* commands with DSOs.

Writing Conditional Commands at the Machine Level

Use the *wheni* command to write conditional commands for use in debugging machine code. The *wheni* command works in the same way as the *when* command described in **"Writing Conditional Commands".** The command list is a list of *dbx* commands, separated by semicolons. When the specified conditions are met, the command list is executed. If one of the commands in the list is *stop* (with no operands), then the process stops when the command list is executed.

`wheni if` *expression* **{***command−list***}**

Evaluates the expression before executing each machine instruction. If the expression is true, executes the command list.

`wheni at` *address* **`if`** *expression* **{***command−list***}**

Evaluates the expression at the given address. If the expression is true, executes the command list.

`wheni in` *procedure* **`if`** *expression* **{***command−list***}**

Evaluates the expression in the given procedure. If the expression is true, executes the command list.

`wheni` *variable* **`at`** *address* **`if`** *expression* **{***command−list***}**

Tests both conditions at the given address. If the conditions are true, executes the command list. (For machine−level debugging.)
If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

`wheni` *variable* **`in`** *procedure* **`if`** *expression* **{***command−list***}**

Tests both conditions at every machine instruction within a given procedure. If they are true, executes the command list.

See also **"Linking With DSOs"** for a description on using the *wheni in* and *wheni at* commands with DSOs.

Stepping Through Machine Code

The stepi and nexti commands allow you to step through machine code in much the same way as you can with the *step* and *next* commands while debugging source code. The *step* and *next* commands step in units of machine instructions instead of in lines of code.

The formats of the *nexti* and *stepi* commands are:

`nexti` [*integer*] Executes the specified number of machine instructions, stepping over procedures. If you do not provide an argument, *nexti* executes one instruction. If *nexti* encounters any breakpoints, even in procedures that it steps over, it immediately stops execution.

`stepi` Single steps one machine instruction, stepping into procedures (as called by *jal* and *jalr*). If *stepi* encounters any breakpoints, it immediately stops execution.

`stepi` [*n*] Executes the specified number of machine instructions, stepping into procedures (as called by *jal* and *jalr*).

The values of the *dbx* variable *$stepintoall* affects the *stepi* and *nexti* commands just as they do the *step* and *next* commands. See **"Stepping Through Your Program"** for a discussion of these variables.

For the *stepj* command, *dbx* determines the next jump by reading through the instructions until it finds a *jump* or *jal* machine instruction. Because it ignores conditional branches, it does not necessarily follow the sequence of the program's execution to find the "next" jump. This means that it may not stop where you expect.

Multiple Process Debugging

This section explains multiprocess debugging procedures, including:

- **"Listing Available Processes"**
- **"Adding a Process to the Process Pool"**
- **"Deleting a Process From the Process Pool"**
- **"Selecting a Process"**
- **"Suspending a Process"**
- **"Resuming a Suspended Process"**
- **"Waiting for a Resumed Process"**
- **"Waiting for Any Running Process"**
- **"Killing a Process"**
- **"Handling fork System Calls"**
- **"Handling exec System Calls"**
- **"Handling sproc System Calls and Process Group Debugging"**

Processes

dbx supports debugging multiprocess applications, including processes spawned with either the *fork*(2) or *sproc*(2) system calls. You can attach child processes automatically to *dbx*. You also can perform process control operations on a single process or on all processes in a group.

dbx provides commands specifically for seizing, stopping, and debugging currently running processes. When *dbx* seizes a process, it adds it to a pool of processes available for debugging. Once you select a process from the pool of available processes, you can use all the *dbx* commands normally available.

Once you are finished with the process, you can terminate it, return it to the pool, or return it to the operating system.

Using the pid Clause

Many *dbx* commands allow you to append the clause **`pid`** *pid* (where *pid* is a numeric process ID or a debugger variable holding a process ID). Using the **`pid`** *pid* clause means you can apply a command to any process in the process pool even though it is not the active process.

For example, to set a breakpoint at line 97 of the process whose ID is 12745, enter:

```
(dbx) stop at 97 pid 12745
Process 12745: [3] stop at "/usr/demo/test.c":97
```

Commands that accept the **`pid`** *pid* clause include:

```
active          edit    resume   wait
addproc         file    return    whatis
assign          func         showpoc       when, when[i]
catch           goto         status        where
cont, cont[i]   ignore       step, step[i}  whereis
delete          kill         stop, stop[i]  which
delproc         next         suspend
directory       print        trace, trace[i]
down            printf       up
dump            printregs    use
```

Using the *pgrp* Clause

Many *dbx* commands allow the *pgrp* clause as a way to apply a command to several processes. For information, see **"Handling sproc System Calls and Process Group Debugging".**

Using Scripts

Additionally, *dbx* provides two variables that you can use when writing scripts to debug multiprocess programs:

$lastchild	Always set to the process ID of the last child process created by a *fork* or *sproc*.
$pid0	Always set to the process ID of the process started by the *run* command.

See the *dbx* online help file section on *hint_mp_debug* for sample multiprocessing debugging scripts.

Listing Available Processes

Use the *showproc* command to list the available processes:

showproc	Shows processes already in the *dbx* process pool or processes that *dbx* can control. Without any arguments, *dbx* lists the processes it already controls.
showproc all	Lists all the processes it controls as well as all those processes it could control but that are not yet added to the process pool.
showproc *pid*	Shows the status of the process ID.

For example, to display all processes in the process pool, enter:

```
(dbx) showproc
Process 12711 (test) Trace/BPT trap [main:14 ,0x40028c]
Process 12712 (test) Trace/BPT trap [main:18 ,0x4002b4]
```

To display only process 12712, enter:

```
(dbx) showproc 12712
Process 12712 (test) Trace/BPT trap [main:18 ,0x4002b4]
```

To display all processes that *dbx* can control, enter:

```
(dbx) showproc all
Process 12711 (test) Trace/BPT trap [main:14 ,0x40028c]
Process 12055 (tcsh)
Process 12006 (clock)
Process 12054 (tcsh)
Process 12673 (zipxgizmo)
Process 12672 (zip)
Process 11974 (4Dwm)
Process 12712 (test) Trace/BPT trap [main:18 ,0x4002b4]
Process 12708 (dbx)
Process 12034 (xlock)
```

Adding a Process to the Process Pool

The *addproc* command adds one or more specified processes to the *dbx* process pool. This allows you to debug a program that is already running. The syntax of the *addproc* command is:

addproc *pid* [...]
addproc *var*

For example:

```
(dbx) addproc 12924
Reading symbolic information of Process 12924 . . .
Process 12924 (loop_test) added to pool
Process 12924 (loop_test) running
```

Equivalently, you can enter either of the following:

```
(dbx) set $foo = 12924
(dbx) addproc $foo
```

Deleting a Process From the Process Pool

The *delproc* command removes a process or variable from the process pool, freeing it from *dbx* control.

The *delproc* command removes a process or variable from the process pool, freeing it from *dbx* control. When you delete a process from the process pool, *dbx* automatically returns the process to normal operation. The syntax of the *delproc* command is:

`delproc` *pid* [...]
`delproc` *var*

For example:

```
(dbx) delproc 12924
Process 12924 (loop_test) deleted from pool
```

Equivalently, you can enter either of the following:

```
(dbx) set $foo = 12924
(dbx) delproc $foo
```

Selecting a Process

The *dbx* command has the ability to control multiple processes. However, *dbx* commands (by default) apply to only one process at a time, the *active process*. To select a process from the process pool to be the active process, use the *active* command:

`active` [*pid*]	Selects a process, *pid*, from *dbx* process pool as the active process. If you do not provide a process ID, *dbx* prints the currently active process ID.

For example, to determine which process is currently active, enter:

```
(dbx) active
Process 12976 (test1) is active
```

To then select process 12977 as the active process, enter:

```
(dbx) active 12977
Process 12977 (test1) after fork [.fork.fork:15 +0x8,0x4005e8]
```

Suspending a Process

The *suspend* command allows you to stop a process in the *dbx* process pool:

`suspend`	Suspends the active process if it is running. If it is not running, this command does nothing.
`suspend all`	Suspends all the processes.
`suspend pid` *pid*	Suspends the process *pid* if it is in the *dbx* process pool. If it is not running, this command does nothing.
`suspend` *pgrp*	Suspends all the processes in the *pgrp*.

For example, to stop the active process, enter:

```
(dbx) suspend
Process 12987 (loop_test) requested stop [main:10 +0x8,0x400244]
 10 i = i % 10;
```

Then to stop process 12988, enter:

```
(dbx) suspend pid 12988
Process 12988 (test3) requested stop [main:29 +0x4,0x400424]
 10 j = k / 10.0;
```

Resuming a Suspended Process

To resume execution of a suspended *dbx* controlled process, you can use either the *cont* command or the *resume* command. If you use *cont*, you do not return to the *dbx* command interpreter until the program encounters an event (for example, a breakpoint). On the other hand, the *resume* command program encounters an event (for example, a breakpoint). On the other hand, the *resume* command returns immediately to the *dbx* command interpreter.

The syntax of the *resume* command is:

`resume`	Resumes execution of the program, and returns immediately to the *dbx* command interpreter.
`resume` [*signal*]	Resumes execution of the process, sending it the specified signal, and returns immediately to the *dbx* command interpreter.

Because the *resume* command returns you to the *dbx* command interpreter after restarting the process, it is more useful than using the *cont* command when you're debugging multiple processes. With *resume*, you are free to select and debug a process while another process is running.

If any *resume*d process modifies the terminal modes (for example if it uses *curses*(3X)), *dbx* can't correctly control the modes. Intercept programs using *curses* by typing **`dbx -p`** (or **`dbx -P`**).

For example, if you are debugging multiple processes and want to resume the active process, enter:

```
(dbx) resume
```

dbx restarts the active process and returns the *dbx* prompt. You can then continue debugging, for example by switching to another process.

To resume all the processes in *pgrp 2* and send a *SIGINT* signal to the process when *dbx* resumes, enter:

```
(dbx) resume SIGINT 2
```

Waiting for a Resumed Process

To wait for a process to stop for an event (such as a breakpoint), use the *wait* command. This is useful after a *resume* command. Also refer to the description of the *waitall* command, described in **"Waiting for Any Running Process".**

The syntax of the *wait* command is:

`wait`	Waits for the active process to stop for an event.
`wait pid` *pid*	Waits for the process *pid* to stop for an event.

For example, assume that you want to wait until process 14280 stops, perhaps at a breakpoint you have set. To do so, enter:

```
(dbx) wait pid 14280
```

After you enter this command, *dbx* waits until process 14280 stops, at which point it displays the *dbx* prompt.

Waiting for Any Running Process

To wait for any process currently running to breakpoint or stop for any reason, use the *waitall* command. It causes *dbx* to wait until a running process in the process list stops, at which point it returns you to the *dbx* command interpreter.

Note: When you return to the *dbx* command interpreter after a *waitall* command, *dbx* does *not* make the process that stopped the active process. You must use the *active* command to change the active process.

For example, to wait until one of your processes under *dbx* control stops, enter:

```
(dbx) waitall
```

After you enter this command, *dbx* waits until a process stops, at which point it indicates which process stopped and displays the *dbx* prompt. For example:

```
Process 14281 (loop_test) Terminated [main:10 +0x8,0x400244]
 10 i = i % 10;
(dbx)
```

Killing a Process

To kill a process in the process pool while running *dbx*, use the *kill* command:

`kill`	Kills the active process.
`kill` *pid* [...]	Kills the specified process(es).

For example, to kill process 14257, enter:

```
(dbx) kill 14257
Process 14257 (fork_test) terminated
Process 14257 (fork_test) deleted from pool
```

Handling *fork* System Calls

When a program executes a *fork* system call and starts another process, *dbx* allows you to add that process to the process pool. (See also **"Stopping at System Calls".**)

The *dbx* variable *$promptonfork* determines how *dbx* treats forks. **Table 4–10** summarizes its effects.

Table 4–10 How the *$promptonfork* Variable Affects *dbx*'s Treatment of Forks

$promptonfork Value	Effect on *dbx*'s Treatment of Forks
0 (default)	*dbx* does not add the child process to the process pool. Both the child process and the parent process continue to run.
1	*dbx* stops the parent process and asks if you want to add the child process to the process pool. If you answer yes, then *dbx* adds the child process to the pool and stops the child process; if you answer no, *dbx* allows the child process to run and does not place it in the process pool.
2	*dbx* automatically stops both the parent and child processes and adds the child process to the process pool.

Note: **"Handling sproc System Calls and Process Group Debugging"** provides additional information on debugging multiprocessing programs; some of the material in that section can apply also to programs that use the *fork* system call.

Consider a program named *fork* that contains these lines:

```
main(argc, argv)
int argc;
char *argv;
{
     int pid;
     if ((pid = fork()) == -1)
        perror("fork");
     else if (pid == 0)
        printf("child\n");
     else { printf("parent\n");
}
```

If you set *$promptonfork* to 1 and run the program, *dbx* prompts you whether it should add the child process to the process pool:

```
(dbx) set $promptonfork = 1
(dbx) run
Process 22661 (fork) started
Process 22662 (fork) has executed the "fork" system call

Add child to process pool (n if no)?y
Process 22662 (fork) added to pool
Process 22662 (fork) stopped on sysexit fork [_fork:28 ,0x40643a4]
```

```
Process 22661 (fork) stopped on sysexit fork [_fork:28 ,0x40643a4]
           Source (of /shamu/lib/libc/libc_64/proc/fork.s) not available for
 Process 22661
```

Handling *exec* System Calls

The *exec* system call executes another program. During an *exec*, the first program gives up its process number to the program it executes. When a program using DSOs executes an *exec()* call, *dbx* runs the new program to *main*. When a program linked with a non−shared library executes an *exec()* call, *dbx* reads the symbolic information for the new program and then stops program execution. In either case, you can continue by entering a *cont* or *resume* command.

For example, consider the programs *exec1.c* and *exec2.c*:

```
/* exec1.c */
main()
{
     printf("in exec1\n");

     /* Invoke the "exec2" program */

     execl("exec2", "exec2", 0);

     /* We'll only get here if execl() fails */

     perror("execl");
}
/* exec2.c */
main()
{
     printf("in exec2\n");
}
```

You can enter *cont* to continue executing *exec2*. For example:

```
(dbx) cont
in exec2
Process 14409 (exec2) finished
```

Handling *sproc* System Calls and Process Group Debugging

The process group facility allows a group of processes to be operated on simultaneously by a single *dbx* command. This is more convenient to use when dealing with processes created with the *sproc* system call than issuing individual *resume*, *suspend*, or breakpoint setting commands. This facility was created to deal more conveniently with parallel programs created, for example, by the Power Fortran Accelerator (PFA).

The *dbx* variable *$mp_program* determines how *dbx* treats *sproc* system calls. **Table 4−11** summarizes its effects.

Table 4−11 How the *$mp_program* Variable Affects *dbx*'s Treatment of *sproc*s

***$mp_program* Value**	**Effect on *dbx*'s Treatment of *sproc***
0 (default)	*dbx* treats calls to *sproc* in the same way as it treats calls to *fork*.
1	Child processes created by calls to *sproc* are allowed to run; they block on multiprocessor synchronization code emitted by mp Fortran or C code. When you set *$mp_program* to 1, multiprocess Fortran or C code is easier to debug.

Whenever a process executes a *sproc*, if *dbx* adds the child to the process pool, *dbx* also adds the parent and child to the *group list*. The group list is simply a list of processes. If you set the *dbx* variable *$groupforktoo* to 1, then forked processes are added to the group list automatically just as *sproc*ed processes are. (By default, *$groupforktoo* is 0.)

You can explicitly add one or more processes to the group list with the *addpgrp* command (you can add only processes in the process pool to the group list):

`addpgrp` *pid* [...]

You can remove processes from the group list with the *delpgrp* command:

`delpgrp` *pid* [...]

The *showpgrp* command displays information about the group list. The *showpgrp* command shows the process group numbers and all the *stop*, *trace*, or *when* events in each. These events are created by *stop[i]*, *when[i]* ... *pgrp* (which create multiple *stop*, *trace*, or *when* events) and by *delete pgrp* commands, which delete them.

The following example shows the output of the *showpgrp* command with two processes in the group list:

```
(dbx) showpgrp
2 processes in group:
   14559 14558
```

Once you add processes to the group list (by adding the keyword *pgrp* to the end of certain *dbx* commands), you can apply that command to all processes in the group. The commands to which you can append *pgrp* are: *delete*, *next[i]*, *resume*, *stop[i]*, *status*, *suspend*, *trace[i]*, and *when[i]*.

The breakpoints and traces set by the *stop[i]*, *trace[i]*, and *when* commands, when used with the *pgrp* keyword, are also added to the *group history*. This group history is displayed as a numbered list when you execute *showpgrp*.

For example, to delete the history entry 7 for the process group, enter:

```
delete 7 pgrp
```

Thus, you can delete breakpoints from multiple processes with a single command.

Breakpoints set on the process group are recorded both in the group and in each process. Deleting breakpoints individually (even if set by a group command) is allowed.

For example, the following command sets a breakpoint at line 10 in all processes in the group list:

```
(dbx) stop at 10 pgrp
Process 14558: [6] stop at "/usr/demo/pgrp_test.c":10
Process 14559: [7] stop at "/usr/demo/pgrp_test.c":10
```

If you now enter a *status* command, only those breakpoints associated with the active process are displayed:

```
(dbx) status
Process 14559: [7] {pgrp 269011340} stop at "/usr/demo/pgrp_test.c":10
```

By appending the keyword *pgrp*, you can display the breakpoints for all processes in the group list:

```
(dbx) status pgrp
Process 14558: [6] {pgrp 269011276} stop at "/usr/demo/pgrp_test.c":10
Process 14559: [7] {pgrp 269011340} stop at "/usr/demo/pgrp_test.c":10
```

Use the *showpgrp* command to display the group history:

```
(dbx) showpgrp
2 processes in group:
   14559 14558
Group history number: 10
   Process 14558 Process 14558: [6] stop at "/usr/demo/pgrp_test.c":10
   Process 14559 Process 14559: [7] stop at "/usr/demo/pgrp_test.c":10
```

You can delete the breakpoints in both processes by deleting the associated group history entry. For example, enter:

```
(dbx) delete 10 pgrp
```

```
(dbx) showpgrp
2 processes in group:
   14559 14558
```

dbx Commands

All *dbx* commands are listed below along with a brief description. For more information about a command, refer to its description in previous sections of this chapter.

;	Use the semicolon (;) as a separator to include multiple commands on the same command line.
****	Use the backslash (\\) at the end of a line of input to *dbx* to indicate that the command is continued on the next line.
./	Repeats the previous examine command by incrementing the address.
/[*reg_exp*]	Searches forward through the current source file from the current line for the regular expression *reg_exp*. If *dbx* reaches the end of the file without finding the regular expression, it wraps around to the beginning of the file. *dbx* prints the first source line containing a match of the search expression. If you do not supply *reg_exp*, *dbx* searches forward, based on the last regular expression you searched for.
.?	Repeats the previous examine command by decrementing the address.
?[*reg_exp*]	Searches backward through the current source file from the current line for the regular expression *reg_exp*. If *dbx* reaches the beginning of the file without finding the regular expression, it wraps around to the end of the file. *dbx* prints the first source line containing a match of the search expression. If you do not supply a regular expression, *dbx* searches backward, based on the last regular expression you searched for.
!!	Repeats the previous command. If the value of the *dbx* variable *$repeatmode* is set to 1, then entering a carriage return at an empty line is equivalent to executing **!!**. By default, *$repeatmode* is set to 0.
!*string*	Repeats the most recent command that starts with the specified *string*.
!*integer*	Repeats the command associated with the specified *integer* in the history list.
!-*integer*	Repeats the command that occurred *integer* times before the most recent command. Entering **!-1** executes the previous command, **!-2** the command before that, and so forth.
active [*pid*]	Selects a process, *pid*, from *dbx* process pool as the active process. If you do not provide a process ID, *dbx* prints the currently active process ID.
addpgrp *pid* [...]	Adds the process IDs specified to the group list. Only processes in the process pool can be added to the group list.
addproc *pid* [...]	Adds the specified process(es) to the pool of *dbx* controlled processes.
address **/** *count format*	Prints the contents of the specified address or disassembles the code for the instruction at the specified address. Repeats for a total of *count* addresses in increasing address—in other words, an *examine forward* command. The format codes are listed in **Table 4–9** .
address **?** *count format*	Prints the contents of the specified address or disassembles the code for the instruction at the specified address. Repeats for a total of *count* addresses in decreasing address—in other words, an "examine backward" command. The format codes are listed in **Table 4–9** .
address **/** *count* **L** *value mask*	Examines *count* 32–bit words in increasing address and print those 32–bit words which, when ORed with *mask*, equal *value*. This command searches memory for specific patterns.
alias	Lists all existing aliases.
alias *name*	Lists the alias definition for *name*.

alias *name command*	Defines *name* as an alias for *command*.
alias *name "string"*	Defines *name* as an alias for *string*. With this form of the *alias* command, you can provide command arguments in the alias definition.
alias *name*(*arg1* [, ... *argN*]) *"string"*	Defines *name* as an alias for *string*. *arg1* through *argN* are arguments to the alias, appearing in the *string* definition. When you use the alias, you must provide values for the arguments, which *dbx* then substitutes in *string*.
assign *register* = *expression*	Assigns the value of *expression* to *register*. You must precede the name of the register with a dollar sign ($).
assign *variable* = *expression*	Assigns the value of *expression* to the program variable, *variable*.
catch	Prints a list of all signals caught.
catch {*signal* | **all**}	Instructs *dbx* to stop your program whenever it receives the specified signal. If you use the keyword *all* rather than giving a specific signal, *dbx* catches all signals.
ccall *func*(*arg1*, *arg2*, ... , *argn*)	Calls a function with the given arguments.
clearcalls	Clears all stopped interactive calls.
cont	Continues execution with the current line.
cont {**at** | **to**} *line*	Sets a temporary breakpoint at the specified source line, then resumes execution with the current line. When your program reaches the breakpoint at *line*, *dbx* stops your program and deletes the temporary breakpoint. The keywords *at* and *to* are equivalent.
cont in *procedure*	Sets a temporary breakpoint to stop execution upon entering the specified procedure, then resumes execution with the current line. When your program reaches the breakpoint in *procedure*, *dbx* stops your program and deletes the temporary breakpoint.
cont [*signal*]	Continues execution with the current line and sends the specified signal to your program. If you do not provide a signal, but your program stopped because *dbx* caught a signal intended for your program, then *dbx* sends that signal to your program when you continue execution.
cont [*signal*] {**at** | **to**} *line*	Sets a temporary breakpoint at the specified source line, then resumes execution with the current line and sends the specified signal to your program. If you do not provide a signal, but your program stopped because *dbx* caught a signal intended for your program, then *dbx* sends that signal to your program when you continue execution.
cont [*signal*] **in** *procedure*	Sets a temporary breakpoint to stop execution upon entering the specified procedure, then resumes execution with the current line and sends the specified signal to your program. If you do not provide a signal, but your program stopped because *dbx* caught a signal intended for your program, then *dbx* sends that signal to your program when you continue execution.
conti [*signal*]	Continues execution with the current instruction. If you specify a signal, *dbx* sends the signal to your program. If you do not provide a signal, but your program stopped because *dbx* caught a signal intended for your program, then *dbx* sends that signal to your program when you continue execution.
conti [*signal*] {**at** | **to**} *address*	Sets a temporary breakpoint at the specified address, then resumes execution with the current instruction. When your program reaches the breakpoint at *address*, *dbx* stops your program and deletes the temporary breakpoint. If you specify a signal, then *dbx* sends the signal to your program. If you do not provide a signal, but your program stopped because *dbx* caught a signal intended for your program, then *dbx* sends that signal to your program when you continue execution.

conti [*signal*] **in** *procedure*	Sets a temporary breakpoint to stop execution upon entering the specified procedure, then resumes execution with the current instruction. When your program reaches the breakpoint in *procedure*, *dbx* stops your program and deletes the temporary breakpoint. If you specify a signal, then *dbx* sends the signal to your program. If you do not provide a signal, but your program stopped because *dbx* caught a signal intended for your program, then *dbx* sends that signal to your program when you continue execution.
corefile [*file*]	If you provide a filename, *dbx* uses the program data stored in the core dump *file*. If you do not provide a filename, *dbx* displays the name of the current core file.
delete {*item* [, *item* ...] | **all**}	Deletes the item or items specified. If you use the keyword *all* instead of listing individual items, *dbx* deletes all breakpoints, traces, and conditional commands.
delpgrp *pid* [...]	Deletes the process IDs specified from the group list.
delproc *pid* [...]	Deletes the specified process(es) from the pool of *dbx* controlled processes.
dir [*dir* ...]	If you provide one or more directories, *dbx* adds those directories to the end of the source directory list. If you do not provide any directories, *dbx* displays the current source directory list.
disable *item* [, *item* ...]	Disables the item or items listed. The specified breakpoint(s), trace(s), or conditional command(s) no longer affect program execution. This command has no effect if the item you specify is already disabled.
down [*num*]	Moves down the specified number of activation levels in the stack. The default is one level.
dump	Prints information about the variables in the current procedure.
dump *procedure*	Prints information about the variables in the specified procedure. The procedure must be active.
dump.	Prints information about the variables in all procedures currently active.
edit [*file* | *procedure*]	Edits a file. If you set the *dbx* variable *$editor* to the name of an editor, the *edit* command invokes that editor on the source file. If you do not set the *dbx* variable *$editor*, *dbx* checks whether you have set the environment variable *EDITOR* and, if so, invokes that editor. If you do not set either the *dbx* variable or the environment variable, *dbx* invokes the *vi* editor. When you exit the editor, you return to the *dbx* prompt. If you supply a filename, *edit* invokes the editor on that file. If you supply the name of a procedure, *edit* invokes the editor on the file that contains the source for that procedure. If you do not supply a filename or a procedure name, *edit* invokes the editor on the current source file.
edit pid *pid*	Edits the process ID *pid* clause.
enable *item* [, *item* ...]	Enables the item or items specified. This command activates the specified breakpoint(s), trace(s), or conditional command(s), reversing the effects of a *disable* command, so that they affect program execution.
file [*file*]	Changes the current source file to *file*. The new file becomes the current source file, on which you can search, list, and perform other operations.
func	Displays the name of the procedure corresponding to the current activation level.
func {*activation_level* | *procedure*}	Changes the current activation level. If you specify an activation level by number, *dbx* changes to that activation level. If you specify *procedure*, *dbx* changes to the activation level of that procedure. If you specify a procedure name and that procedure has called itself recursively, *dbx* changes to the most recently called instance of that procedure. If you specify *procedure*, *dbx* changes the current source file to be that procedure, even if the procedure is not active.
givenfile [*file*]	

	executable code and debugging information found in *file*. If you do not provide a filename, *dbx* displays the name of the program that it is currently debugging.
goto *line*	Begins execution at the specified line. You may not use the *goto* command to resume execution with a line outside of the current procedure.
hed	Edits only the last line of the history list (the last command executed).
hed *num1*	Edits line *num1* in the history list.
hed *num1***,** *num2*	Edits the lines in the history list from *num1* to *num2*.
hed all	Edits the entire history list.
help	Shows the list of available help sections.
help all	Displays the entire *dbx* help file. *dbx* displays the file using the command name given by the *dbx* *$pager* variable. The *dbx* help file is large and can be difficult to use if you use a simple paging program like *more*(1). A useful technique is to set the *$pager* variable to a text editor like *vi*(1).
help help	Explains how to display the help file in your favorite editor.
help *section*	Shows this help section. *dbx* displays the file using the command name given by the *dbx* *$pager* variable. (By default, it uses *more*.) A useful technique is to set the *$pager* variable to a text editor like *vi*(1).
history	Prints the commands in the history list.
ignore	Prints a list of all signals ignored.
ignore {*signal* | **all**}	Instructs *dbx* to ignore the specified signal. All ignored signals are passed to your program normally. If you use the keyword *all* rather than giving a specific signal, *dbx* ignores all signals.
kill	Kills the active process.
kill *pid* ...	Kills the active process(es) whose PIDs are specified.
list	Lists *$listwindow* lines beginning at the current line.
list *exp*	Lists *$listwindow* lines starting with the line number given by the expression *exp*. The expression may be any valid expression that evaluates to an integer value.
list *exp1***:** *exp2*	Lists *exp2* lines, beginning at line *exp1*.
list *exp1***,** *exp2*	Lists all source between line *exp1* and line *exp2* inclusive.
list *func*	Lists *$listwindow* lines starting at procedure *func*.
list *func*: *exp*	Lists *exp2* lines, beginning at *func*.
list *func*, *exp*	Lists all source between *func* and *exp*, inclusive.
listobj	Lists dynamic shared objects being used. The base application is first in the list.
next [*n*]	Executes the specified number of lines of source code, stepping over procedures. If you do not provide an argument, *next* executes one line. If *next* encounters any breakpoints, even in procedures that it steps over, it immediately stops execution.
nexti [*n*]	Executes the specified number of machine instructions, stepping over procedures. If you do not provide an argument, *nexti* executes one line. If *nexti* encounters any breakpoints, even in procedures which it steps over, it immediately stops execution.
playback input [*file*]	Executes the commands from *file*. The default file is the current temporary file created for the *record input* command. If the *dbx* variable *$pimode* is nonzero, the commands are printed out as they are played back.
playback output [*file*]	Prints the commands from *file*. The default file is the current temporary file created for the *record output* command.
print [*exp1* [, *exp2*, ...]]	Prints the value(s) of the specified expression(s).
printd [*exp1* [, *exp2*, ...]]	Prints the value(s) of the specified expression(s) in decimal.
printf *string* [, *exp1* [, *exp2*, ...]]	Prints the value(s) of the specified expression(s) in the format specified by the string, *string*. The *printf* command supports all formats except "%s". For a list of formats, see the *printf*(3S) reference page.
printo [*exp1* [, *exp2*, ...]]	

printregs	Prints all register values.
printx [*exp1* [, *exp2*, ...]]	Prints the value(s) of the specified expression(s) in hexadecimal.
quit	Quits *dbx*.
record	Displays the current input and output recording sessions.
record input [*file*]	Records everything you type to *dbx* in *file*. The default file is a temporary *dbx* file in the */tmp* directory. The name of the temporary file is stored in the *dbx* variable *$defaultin*.
record output [*file*]	Records all *dbx* output in *file*. The default file is a temporary *dbx* file in the */tmp* directory. The name of the temporary file is stored in the *dbx* variable *$defaultout*. If the *dbx* variable *$rimode* is nonzero, *dbx* also records the commands you enter.
rerun *run−arguments*	Without any arguments, repeats the last *run* command, if applicable. Otherwise, *rerun* is equivalent to the *run* command without any arguments.
resume	Resumes execution of the program, and returns immediately to the *dbx* command interpreter.
resume [*signal*]	Resumes execution of the process, sending it the specified signal, and returns immediately to the *dbx* command interpreter.
return	Continues execution until control returns to the next procedure on the stack.
return *proc*	Continues execution until control returns to the named procedure.
run *run−arguments*	Starts your program and passes to it any arguments that you provide. All shell processing is accepted, such as unglobbing of * and ? in filenames. Redirection of the program's standard input and standard output, and/or standard error is also done by the shell. In other words, the *run* command does exactly what typing *target run−arguments* does. You can specify a target, either on *dbx* invocation or in a prior *givenfile* command. *dbx* passes *./target* as *argv[0]* to *target* when you specify it as a relative pathname. You can specify *target* either on *dbx* invocation or in a prior *givenfile* command. *dbx* passes *./target* as *argv[0]* to *target* when you specify it as a relative pathname. A *run* command must appear on a line by itself and cannot be followed by another *dbx* command. Terminate the command line with a return (new−line). Note that you cannot include a *run* command in the command list of a *when* command.
set	Displays a list of predefined and user defined variables.
set *var* **=** *exp*	Defines (or redefines) the specified *dbx* variable, setting its value to that of the expression you provide.
sh	Invokes a subshell. To return to *dbx* from the subshell, enter **exit** at the command line, or otherwise terminate the subshell.
sh *com*	Executes the specified shell command. *dbx* interprets the remainder of the line as a command to pass to the spawned shell process, unless you enclose the command in double−quotes or you terminate your shell command with a semicolon (;).
showpgrp	Shows the group process list and the group history.
showproc [*pid* | **all**]	Shows processes already in the *dbx* process pool or processes that *dbx* can control. If you provide no arguments, *dbx* lists the processes it already controls. If you provide a *pid*, *dbx* displays the status of the specified process. If you use argument "all," *dbx* lists all the processes it controls as well as all those processes it could control but that are not yet added to the process pool.
source [*file*]	Executes *dbx* commands from *file*.
status	Displays all breakpoints, traces, and conditional commands.
step [*n*]	Executes the specified number of lines of source code, stepping into procedures. If you do not provide an argument, *step* executes one line. If *step* encounters any breakpoints, it immediately stops execution.
stepi	Single steps one machine instruction, stepping into procedures (as called by *jal* and *jalr*). If *stepi* encounters any breakpoints, it immediately stops execution.
stepi [*n*]	Executes the specified number of machine instructions, stepping into procedures (as called by *jal* and *jalr*).
stop at	Set a breakpoint at the current source line.

stop at *line*	Sets a breakpoint at the specified source line.
stop *expression*	Inspects the expression. If the expression is type pointer, checks the data being pointed at. Otherwise, checks the 32 bits at the address given by the expression.
stop in *procedure*	Sets a breakpoint to stop execution upon entering the specified procedure.
stop [*expression|variable*]	Inspects the value before executing each source line. If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).
stop [*expression|variable*] **at** *line*	Inspects the value at the given source line. Stops if the value has changed. If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).
stop [*expression|variable*] **in** *procedure*	Inspects the value at every source line within a given procedure. Stops if the value has changed. If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).
stop if *expression*	Evaluates the expression before executing each source line. Stops if the expression is true.
stop at *line* **if** *expression*	Evaluates the expression at the given source line. Stops if the expression is true.
stop in *procedure* **if** *expression*	Evaluates the expression at every source line within a given procedure. Stops if the expression is true.
stop [*expression1|variable*] **if** *expression2*	Tests both conditions before executing each source line. Stops if both conditions are true. If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).
stop [*expression1|variable*] **at** *line* **if** *expression2*	Tests both conditions at the given source line. Stops if both conditions are true. If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).
stop [*expression1|variable*] **in** *procedure* **if** *expression2*	Tests both conditions at every source line within a given procedure. Stops if both conditions are true. If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).
stopi at	Sets an unconditional breakpoint at the current machine instruction.
stopi at *address*	Sets an unconditional breakpoint at the specified address (for machine−level debugging).
stopi in *procedure*	Sets an unconditional breakpoint to stop execution upon entering the specified procedure (for machine−level debugging).
stopi [*expression|variable*]	Inspects the value before executing each machine instruction and stops if the value has changed.

changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

stopi [*expression*|*variable*] **at** *address*

Inspects the value when the program is at the given address and stops if the value has changed (for machine−level debugging).

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

stopi [*expression*|*variable*] **in** *procedure*

Inspects the value at every machine instruction within a given procedure and stops if the value has changed.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

stopi if *expression*

Evaluates the expression before executing each machine instruction and stops if the expression is true.

stopi at *address* **if** *expression*

Evaluates the expression at the given address and stops if the expression is true (for machine−level debugging).

stopi in *procedure* **if** *expression*

Evaluates the expression at every machine instruction within a given procedure and stops if the expression is true.

stopi [*expression1*|*variable*] **if** *expression2*

Tests both conditions before executing each machine instruction. Stops if both conditions are true.

If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

stopi [*expression1*|*variable*] **at** *address* **if** *expression2*

Tests both conditions at the given address (for machine−level debugging). Stops if both conditions are true.

If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

stopi [*expression1*|*variable*] **in** *procedure* **if** *expression2*

Tests the expression each time that the given variable changes within the given procedure.

If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

suspend Suspends the active process if it is running. If it is not running, this command does nothing. If you use the keyword *all*, suspends all active processes.

suspend *pgrp*

Suspends all the processes in *pgrp*.

suspend pid *pid*

Suspends the process *pid* if it is in the *dbx* process pool. If it is not running, this command does nothing.

syscall Prints a summary of the catch and ignore status of all system calls. The summary is divided into four sections: 1) caught at call, 2) caught at return, 3) ignored at call, and 4) ignored at return.

syscall catch [{**call** | **return**}]

Prints a list of all system calls caught upon entry (*call*) or return (*return*). If you provide neither the *call* nor *return* keyword, *dbx* lists all system calls that are caught.

syscall ignore [{**call** | **return**}]

Prints a list of all system calls not caught upon entry (*call*) or return (*return*). If you provide neither the *call* nor *return* keyword, *dbx* lists all system calls that are ignored.

syscall catch {**call** | **return**} {*system_call* | **all**}

Sets a breakpoint to stop execution upon entering (*call*) or returning from (*return*) the specified system call. Note that you can set *dbx* to catch both the call and the return of a system call.

If you use the keyword *all* rather than giving a specific system call, *dbx* catches all system calls.

`syscall ignore` {**`call`** | **`return`**} {*system_call* | **`all`**}

Clears the breakpoint to stop execution upon entering (*call*) or returning from (*return*) the specified system call.

If you use the keyword *all* rather than giving a specific system call, *dbx* clears the breakpoints to stop execution upon entering (*call*) or returning from (*return*) all system calls.

`tag` *procedure* Searches the tag file for the given procedure.

`trace` *variable* Whenever the specified variable changes, *dbx* prints the old and new values of that variable.

`trace` *procedure*Prints the values of the parameters passed to the specified procedure whenever your program calls it. Upon return, *dbx* prints the return value.

`trace` [*expression*|*variable*] **`at`** *line*

Whenever your program reaches the specified line, *dbx* prints the value of the variable if its value has changed.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

`trace` [*expression*|*variable*] **`in`** *procedure*

Whenever the variable changes within the procedure, *dbx* prints the old and new values of that variable.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

`trace` [*expression1*|*variable*] **`at`** *line* **`if`** *expression2*

Prints the value of the variable (if changed) whenever your program reaches the specified line and the given expression is true.

If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

`trace` [*expression1*|*variable*] **`in`** *procedure* **`if`** *expression2*

Whenever the variable changes within the procedure that you specify, *dbx* prints the old and new values of that variable, if the given expression is true.

If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

`tracei` [*expression*|*variable*]

Whenever the specified variable changes, *dbx* prints the old and new values of that variable. (For machine−level debugging.)

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

`tracei` *procedure*

This command is equivalent to entering **`trace`** *procedure*. (For machine−level debugging.)

`tracei` [*expression*|*variable*] **`at`** *address*

Prints the value of the variable whenever your program reaches the specified address. (For machine−level debugging.)

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

`tracei` [*expression*|*variable*] **`in`** *procedure*

Whenever the variable changes within the procedure that you specify, *dbx* prints the old and new values of that variable. (For machine−level debugging.)

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address

tracei [*expression1*|*variable*] **at** *address* **if** *expression2*

Prints the value of the variable whenever your program reaches the specified address and the given expression is true. (For machine−level debugging.)

If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

tracei [*expression1*|*variable*] **in** *procedure* **if** *expression2*

Whenever the variable changes within the procedure that you specify, *dbx* prints the old and new values of that variable, if the given expression is true. (For machine−level debugging.)

If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

unalias *alias* Removes the specified alias.

unrecord *session1* [**,** *session2* ...]

Turns off the specified recording session(s) and closes the file(s) involved.

unrecord all Turns off all recording sessions and closes all files involved.

unset *var* Removes the specified *dbx* variable.

up [*num*] Moves up the specified number of activation levels in the stack. The default is one level.

use [*dir* ...] If you provide one or more directories, *dbx* replaces the source directory list with the directories that you provide.

If you do not provide any directories, *dbx* displays the current source directory list.

wait Waits for the active process to stop for an event.

wait pid *pid* Waits for the process *pid* to stop for an event.

waitall Waits for any process currently running to breakpoint or stop for any reason.

whatis *variable* Prints the type declaration for the specified variable or procedure.

when [*expression*|*variable*] **{** *command−list* **}**

Inspects the value before executing each source line. If it has changed, executes the command list.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

when [*expression*|*variable*] **at** *line* **{** *command−list* **}**

Inspects the value at the given source line. If it has changed, executes the command list.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

when [*expression*|*variable*] **in** *procedure* **{** *command−list* **}**

Inspects the value at every source line within a given procedure. If it has changed, executes the command list.

If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

when if *expression* **{** *command−list* **}**

Evaluates the expression before executing each source line. If it is true, executes the command list.

when at *line* **if** *expression* **{** *command−list* **}**

Evaluates the expression at the given source line. If it is true, executes the command list.

when in *procedure* **if** *expression* **{** *command−list* **}**

Evaluates the expression at every source line within a given procedure. If it is true, executes the command list.

when [*expression1*|*variable*] **if** *expression2* **{** *command−list* **}**

Checks if the value of the variable has changed. If it has changed and the expression is true, executes the command list.

If *expression1* is of type pointer, look at the data pointed to and watch until it changes.

expression evaluates to an address).

when [*expression1*|*variable*] **at** *line* **if** *expression2* **{***command–list***}**
Checks if the value of the variable has changed each time the line is executed. If the value has changed and the expression is true, executes the command list.
If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

when [*expression1*|*variable*] **in** *procedure* **if** *expression2* **{***command–list***}**
Checks if the value of variable has changed at each source line of the given procedure. If the value has changed and the expression is true, executes the command list.
If *expression1* is of type pointer, look at the data pointed to and watch until it changes. If *expression1* is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

wheni if *expression* **{***command–list***}**
Evaluates the expression before executing each machine instruction. If the expression is true, executes the command list.

wheni at *address* **if** *expression* **{***command–list***}**
Evaluates the expression at the given address. If the expression is true, executes the command list. (For machine–level debugging.)

wheni in *procedure* **if** *expression* **{***command–list***}**
Evaluates the expression in the given procedure. If the expression is true, executes the command list. (For machine–level debugging.)

wheni *variable* **at** *address* **if** *expression* **{***command–list***}**
Tests both conditions at the given address. If the conditions are true, executes the command list. (For machine–level debugging.)
If the expression is of type pointer, look at the data pointed to and watch until it changes. If the expression is not of type pointer, look at the 32 bits at that address (assume the expression evaluates to an address).

wheni *variable* **in** *procedure* **if** *expression* **{***command–list***}**
Tests both conditions at every machine instruction within a given procedure. If they are true, executes the command list.

where Print a stack trace.

which *variable* Prints the fully qualified name of the active version of the specified variable.

whichobj *variable*
Lists the dynamic shared objects that contain the named variable.

Predefined Aliases

Table 4–12 lists all predefined *dbx* aliases. You can override any predefined alias by redefining it with the *alias* command or by removing it with the *unalias* command.

Table 4–12 Predefined Aliases

Alias	Definition	Description
a	assign	Assigns the specified expression to the specified program variable or register.
b	stop at	Sets a breakpoint at the specified line.
bp	stop in	Sets a breakpoint in the specified procedure.
c	cont	Continues program execution after a breakpoint.
d	delete	Deletes the specified item from the status list.
dir	directory	Displays the current source directory list. If you specify one or more directories, those directories are added to the end of the source directory list.
e	file	Displays the name of the currently selected source file. If you specify a file, this command makes the specified file the currently selected source file.
f	func	Moves to the specified procedure (activation level) on the stack. If you specify no procedure or expression, *dbx* prints the current activation level.

g	goto	Goes to the specified source line.
h	history	Lists all the items currently in the history list.
j	status	Lists all the currently set *stop*, *trace*, and *when* commands.
l	list	Lists the next *$listwindow* lines of source code beginning at the current line.
li	$curpc/10i; \ set $curpc=$curpc+40	Lists the next 40 bytes of machine instructions (approximately 10 instructions).
n	next	Executes the specified number of lines of source code, stepping over procedures. If you do not provide an argument, *dbx* executes only one line.
ni	nexti	Executes the specified number of lines of machine code, stepping over procedures. If you do not provide an argument, *dbx* executes only one instruction.
p	print	Prints the value of the specified variable or expression.
pd	printd	Prints the value of the specified variable in decimal.
pi	playback input	Replays *dbx* commands saved in the specified file. If you do not specify a file, *dbx* uses the temporary file specified by *$defaultin*.
po	printo	Prints the value of the specified variable or expression in octal.
pr	printregs	Prints values contained in all registers.
px	printx	Prints the value of the specified variable or expression in hexadecimal.
q	quit	Quits *dbx*.
r	rerun	Runs the program again using the arguments specified for the last *run* command executed.
ri	record input	Records to the specified file all the input you give to *dbx*. If you do not specify a file, *dbx* creates a temporary file. The name of the file is specified by *$defaultin*.
ro	record output	Records all *dbx* output to the specified file. If no file is specified, records output to a temporary file. The name of the file is specified by *$defaultout*.
s	step	Executes the specified number of lines of source code, stepping into procedures. If you do not provide an argument, *dbx* executes only one line.
S	next	Executes the specified number of lines of source code, stepping over procedures. If you do not provide an argument, *dbx* executes only one line.
si	stepi	Executes the specified number of lines of machine code, stepping into procedures. If you do not provide an argument, *dbx* executes only one instruction.
Si	nexti	Executes the specified number of lines of machine code, stepping over procedures. If you do not provide an argument, *dbx* executes only one instruction.
source	playback input (pi)	Replays *dbx* commands saved in the specified file. If no file is specified, *dbx* uses the temporary file specified by *$defaultin*.
t	where	Does a stack trace to show the current activation levels.
u	list $curline−9:10	Lists a window of source code showing the nine lines before the current code line and the current code line. This command does not change the current code line.

w	list $curline−5:10	Lists a window of source code around the current line. This command shows the four lines before the current code line, the current code line, and five lines after the current code line. This command does not change the current code line.
W	list $curline−10:20	Lists a window of source code around the current line. This command shows the nine lines before the current code line, the current code line, and 10 lines after the current code line. This command does not change the current code line.
wi	$curpc−20/10i	Lists a window of assembly code around the program counter.

Predefined *dbx* Variables

Predefined dbx variables are listed in **Table 4−13** . The predefined variable names begin with "$" so that they do not conflict with variable, command, or alias names.

Table 4−13 Predefined *dbx* Variables

Variable	Default	Description
$addrfmt	"0x%x"	Specifies the format for addresses. This can be set to any format valid for the C language *printf*(3S) function.
$addrfmt64	"0x%x"	Specifies the format for 64−bit addresses. This can be set to any format valid for the C language *printf*(3S) function.
$assignverify		Prints the new value of a variable after an *assign*.
$casesense	2	If 0, symbol names are case sensitive. If 1, symbol names are not case sensitive. If 2, the case sensitivity of symbol names depends on the case sensitivity of the language in which the symbol was defined.
$ctypenames	1	If 1, the words "unsigned," "short," "long," "int," "char," "struct," "union," and "enum" are keywords usable only in type casts. If 0, "struct," "union," and "enum" are ordinary words with no predefined meaning (in C modules, the others are still known as C types).
$curevent		The last event number as seen by the *status* command.
$curline		The current line in the source code being executed.
$curpc		The current program counter.
$cursrcline		The current source listing line plus one.
	0	If the value is 1, *dbx* allows you to set traps in *rld* code, so you can debug bugs in run−time initialization of C++ global variables and the like.
$defaultin		The name of the file that *dbx* uses when the *record input* or the *playback input* command is executed with no argument.
$defaultout		The name of the file that *dbx* uses when the *record output* or the *playback output* command is executed with no argument.
$editor	*vi*	The name of the editor to invoke (with the *edit* command). Default value is set to the value of the *EDITOR* environment variable. If *EDITOR* missing, it defaults to *vi*.
$fp_precise	0	(Systems with floating−point precise mode only.) A nonzero value specifies that the *run* command instructs the operating system to run the program in floating−point precise mode (a nondebug mode). The debugged process performs SGI_SET_FP_PRECISE

		and SGI_SET_FP_PRESERVE *syssgi* system calls.
$framereg	1	If 1, all references to registers are to the registers of the current activation level. If 0, all references are to the hardware registers (the registers of activation level 0).
$groupforktoo	0	If 0, adds only processes created with the *sproc*(2) system call to the process group list automatically. If 1, then adds processes created with either the *fork*(2) or *sproc* system calls to process group list.
$hexchars	0	If nonzero, outputs characters in hexadecimal, using C format "%x". This affects char type variables, including those in structures. It does not affect arrays of characters, which are printed using the "%.*s" format.
	0	If nonzero, float and double values print as normal, with a trailing output of `hex ffffffff` and `hex ffffffff 00000000`, respectively. These are hex representations of the bits of the float and double.
$hexin	0	If nonzero, input constants are assumed to be in hexadecimal. This overrides *$octin*.
$hexints	0	If nonzero, outputs integers in hexadecimal format. This overrides *$octints*.
$hexstrings	0	If nonzero, outputs strings and arrays in hexadecimal. For character arrays, if nonzero, the null byte is not taken as a terminator. Instead, prints the entire array (or *$maxlen* values, whichever is less). If 0, then a null byte in a C or C++ character array is taken as the end of the array (the length of the array and *$maxstrlen* can terminate the array printing before a null byte is found).
$historyevent		The current history line number.
$lastchild		The process ID of the last child process created by a *fork* or *sproc* system call.
$lines	100	The number of lines in the history list.
$listwindow	10	Specifies how many lines the *list* command lists.
$maxstrlen	128	Maximum length printed for zero−terminated char strings and arrays. Prints char arrays for array−length, *$maxstrlen* bytes, or up to a null byte, whichever comes first (see *$hexstrings*).
$mp_program	0	If 0, *dbx* treats calls to *sproc* in the same way as it treats calls to *fork*. If 1, child processes created by calls to *sproc* are allowed to run; they block on multiprocessor synchronization code emitted by mp Fortran code. When you set *$mp_program* to 1, mp Fortran code is easier to debug.
$newevent		After every command creating an event, this variable is set to the event's number. The *$newevent* variable is useful in writing scripts that do not use hard−coded event numbers.
$newpgrpevent		Displays the value of the latest *pgrp* event created by *stop[i]*, *trace[i]*, and *when[i]*... *pgrp*. Useful when writing scripts that then do *deletepgrp#* *pgrp* with a symbolic (*dbx* variable) *pgrp#*.
$octin	0	If nonzero, assumes input constants are in octal (*$hexin* overrides *$octin*).
$octints	0	If nonzero, outputs integers in octal format (*$hexints* takes precedence).
$page	1	Specifies whether or not to page when *dbx* output

		scrolls information off the current screen. A nonzero value turns on paging; a 0 turns it off.
$pager	*more*	The name of the program used to display output from *dbx*.
$pagewidth	80	The width of the window in characters (assumes a fixed−width font). Used by *dbx* to calculate how many screen lines are output. *dbx* never inserts newlines; the window software wraps the lines.
$pagewindow	23	Specifies how many lines print when information is longer than one screen. This can be changed to match the number of lines on any terminal. If set to 0, 1 is used.
	0	Uses no pending traps. If nonzero, allows traps that cannot be satisfied immediately to pend until they can be satisfied. This is useful for debugging programs that use *dlopen()* to load symbols as it allows setting breakpoints before the *dlopen()* call. Similar to *$debugrld*, but *$pendingtraps* does not allow setting breakpoints in *rld*. When set to nonzero, mistyped procedure names cause a pending trap to be set.
$piaddtohist	1	If 1, adds commands read from files using the *playback input* command to the command history. If 0, does not add the commands to the history.
$pid		The current process for kernel debugging (**−k**).
$pid0		Set by *dbx* to the process ID of the running process (also called the object file).
$pimode	0	If 1, *dbx* prints the commands read from files using the *playback input* command. If 0, *dbx* does not print the commands. In either case, *dbx* prints the output resulting from such commands.
$printdata	0	Used when disassembling. If 1, prints register contents alongside disassembled instructions. If 0, just prints disassembled instructions.
	0	If 0, prints only the next line to be executed. If nonzero, prints each line that is executed while it single steps.
$printwide	0	If 0, prints arrays one element per line. If nonzero, prints arrays compactly (wide).
		This variable applies only if you invoke *dbx* with the **−k** option (that is, it is not available unless you are doing kernel debugging). Whenever *$pid* is set, *dbx* sets *$procaddr* to the address of the process table entry for that process.
$prompt	`dbx`	The prompt for *dbx*.
$promptonfork	0	If 0, *dbx* does not add the child process to the process pool. Both the child process and the parent process continue to run. If 1, *dbx* stops the parent process and asks if you want to add the child process to the process pool. If you answer yes, then *dbx* adds the child process to the pool and stops the child process; if you answer no, *dbx* allows the child process to run and does not place it in the process pool. If 2, *dbx* automatically stops both the parent and child processes and adds the child process to the process pool.
$regstyle	0	If 0, *dbx* uses the alternate form of the register name (for example, "zero" instead of "r0" and "t1" instead of "r9"). If nonzero, *dbx* uses the machine name ("r0"

		"r9"). If nonzero, *dbx* uses the machine name ("r0" through "r31").
$repeatmode	0	If nonzero, entering a null line (entering a newline on an empty line) repeats the last command. If 0, *dbx* performs no action.
$rimode	0	If 1, *dbx* records commands you enter in addition to output when using the *recordoutput* command. If 0, *dbx* does not copy the commands.
$showbreakaddrs	0	If nonzero, show the address of each breakpoint placed in the code each time it is placed. Removal of the breakpoints is not shown. If multiple breakpoints are placed at one location, only one of the placements is shown. Since breakpoints are frequently placed and removed by *dbx*, the volume of output can be annoying when tracing.
$stacktracelimit		Limits to the depth of the stack trace.
$stepintoall	0	If 0, *step* steps into only those procedures that are compiled with the debugging option **–g**. step steps over all other procedures. If 1 or 2, *step* steps into all procedures. Note that when you debug a source file compiled without symbols or compiled with optimization, the line numbers may sometimes jump erratically. Also note that if *dbx* cannot locate a source file, then it cannot display source lines as you step through a procedure.
$tagfile	*tags*	The name of a file of tags, as created by *ctags*(1). Used by the *tag* command.

Chapter 5

Using the Performance Tools

This chapter explains how to use performance tools to reduce the execution time of your programs. For information about the compiler optimization facilities, see **Chapter 6, "Optimizing Program Performance."**

This chapter covers the following topics:

- **"Overview of Profiling"** explains how profiling can help you to analyze your data.
- **"Profiling With prof"** describes how to run the profiler, *prof* and lists its options.
- **"pc Sampling"** explains how to use *prof* to obtain program counter (pc) sampling.
- **"Basic Block Counting"** covers how to use *prof* and *pixie* perform basic block counting.
- **"Profiling Multiprocessed Executables"** describes how to profile executables that use *sproc* and *sprocsp* system calls.

Although it may be possible to obtain short−term speed increases by relying on unsupported or undocumented quirks of the compiler system, it’s a bad idea to do so. Any such "features" may break in future releases of the compiler system. The best way to produce efficient code that can be trusted to remain efficient is to follow good programming practices; in particular, choose good algorithms and leave the details to the compiler.

The techniques described in this manual comprise only a part of performance tuning. Other areas that you can tune, but are outside the scope of this document, include graphics, I/O, the kernel, system parameters, memory, and real−time system calls.

Overview of Profiling

Profiling is a three−step process that consists of compiling the source program, executing the program, and then running the profiler, *prof*, to analyze the data.

The compiler system provides two types of profiling:

- *Program counter (pc) sampling*, which measures the amount of execution time spent in various parts of the program. This statistical data is obtained by periodically sampling the program counter. By default, *prof* generates pc sampling data.
- *Basic block counting*, which counts the execution of basic blocks (a basic block is a sequence of instructions that is entered only at the beginning and exits only at the end). It produces an exact count of the number of times each basic block is executed.

Profiling With *prof*

Profiling produces detailed information about program execution. You can use profiling tools to find the areas of code where most of the execution time is spent. In a typical program, a large part of the execution time is spent in relatively few sections of code. It is a good idea to concentrate on improving code efficiency in those sections first.

The topics covered below include:

- **"Running the Profiler"**
- **"prof Options"**

Running the Profiler

The profiler program, *prof*, analyzes raw profiling information and produces a printed report. The program analyzes either pc sampling or basic block counting data.

prof Syntax

```
prof [options] [prog_name] [profile_filename ...]
```

options	One of the keywords or keyword abbreviations shown in **Table 5–1** . (Specify either the entire name or the initial character of the option, as indicated in the table.)
prog_name	Specifies the name of the program whose profile data is to be profiled.
profile_filename	Specifies one or more files containing the profile data gathered when the profiled program executed (defaults are explained below). If you specify more than one file, *prof* sums the statistics in the resulting profile listings.

prof Defaults

The *prof* program has these defaults:

- If you do not specify *-pixie*, *prof* assumes pc–sampling data is being analyzed. If you do not specify *profile_filename*, the profiler looks for a *mon.out* file. If this file does not exist in the current directory, *prof* looks for profile input data files in the directory specified by the PROFDIR environment variable (see **"Creating Multiple Profile Data Files"** for information on PROFDIR).
  You may want to use the *-merge* option when you have more than one profile data file. This option merges the data from several profile files into one file.
- If you specify *-pixie* and do not specify *profile_filename*, then *prof* looks for *prog_name.Counts* and provides basic block count information if this file is present.
- If you specify *profile_filename*(s), *prof* determines the file type based on its content: a *prof*– or *pixie* –mode file.

prof Options

Table 5–1 lists *prof* options. Options that apply to basic block counting are indicated as such.

Table 5–1 Options for *prof*

Name	Result
-c[*lock*] *n*	A basic–block–counting option. This option lists the number of seconds spent in each routine, based on the CPU clock frequency *n*, expressed in megahertz.
-d[*is*]	Disassembles and annotates the analyzed object code with cycle times or number of pc samples.
-dso [*dso_name*]	Applies *prof* analysis to only the named DSO. If you don't specify *dso_name*, *prof*

	prints a list of applicable DSO names.
-e[*xclude*] *procedure_name*	Excludes information on the procedures specified by *procedure_name*. If you specify uppercase *-E*, *prof* also omits that procedure from the base upon which it calculates percentages.
-h[*eavy*]	A basic−block−counting option. Same as the *-lines* option, but sorts the lines by their frequency of use.
-i[*nvocations*]	A basic−block−counting option. Lists the number of times each procedure is invoked. The *-exclude* and *-only* options described below apply to callees, but not to callers.
-l[*ines*]	Lists statistics for each line of source code.
-m[*erge*] *filename*	Merges the input files into *filename* (the default is *mon.out*), allowing you to specify the name of the merged file (instead of several filenames) on subsequent profiler runs. This option is useful when using multiple input files of profile data.
-o[*nly*] *procedure_name*	Reports information on only the procedure specified by *procedure_name* rather than the entire program. You can specify more than one *-o* option. If you specify uppercase *-O*, *prof* uses only the named procedures, rather than the entire program, as the base upon which it calculates percentages.
-pcsample	Tells *prof* that the data to be analyzed is from pc sampling. This is the default. This option and *-pixie* are mutually exclusive.
-pixie	A basic−block−counting option. Indicates that information is to be generated on basic block counting, and that the *prog_name.Counts* files produced by *pixie* are to be used by default. This option and *-pcsample* are mutually exclusive.
-p[*rocedures*]	Lists the time spent in each procedure.
-q[*uit*] *n*	Condenses output listings by truncating unwanted lines. You can specify *n* in three ways: *n*, an integer, truncates everything after *n* lines; *n%*, an integer followed by a percent sign, truncates everything after the line containing *n%* calls in the *%calls* column; *ncum%*, an integer, followed by *cum%*, truncates everything after the line containing *ncum%* calls in the *cum%* column.
-t[*estcoverage*]	A basic−block−counting option. Lists line numbers that contain code that is never executed.
-z[*ero*]	A basic−block−counting option. Lists the procedures that are never invoked.

pc Sampling

Program counter (pc) sampling reveals the amount of execution time spent in various parts of a program. The count includes:

- CPU time and memory access time
- Time spent in user routines

pc sampling does not count time spent swapping or time spent accessing external resources.

This section explains how to obtain pc sampling and provides examples showing the use of various *prof* options. Specifically, this section covers:

- **"Obtaining pc Sampling"**
- **"Creating Multiple Profile Data Files"**
- **"pc Sampling Frequency"**
- **"Examples of Using prof to Obtain pc Sampling"**

Obtaining pc Sampling

Obtain pc sampling information by linking the desired source modules using the *-p* option and then executing the resulting program object, which generates raw profile data.

Use the procedure below to obtain pc sampling information. Also refer to **Figure 5−1** , which illustrates how pc sampling works.

1. Compile the program using the appropriate compiler. For example, to compile a C program

myprog.c:

```
% cc -c myprog.c
```

2. Link the object file created in Step 1.

```
% cc -p -o myprog myprog.o
```

Note: You must specify the *-p* profiling option during this step to obtain pc sampling information.

3. Execute the profiled program (just as you would execute an unprofiled program).

```
% myprog
```

During execution, profiling data is saved in the file *mon.out*. You can run the program several times, altering the input data, to create multiple profile data files. You can also use the environment variable PROFDIR as explained in **"Creating Multiple Profile Data Files."**

4. Run the profile formatting program *prof*.

```
% prof -pcsample myprog mon.out
```

prof extracts information from *mon.out* and prints it in an easily readable format. If *mon.out* exists, it is overwritten. Therefore, rename each *mon.out* to save its output. For more information, see the *prof* (1) reference page.

Include or exclude information on specific procedures within your program by using the *-only* or *-exclude* profiler options (refer to **Table 5–1**).

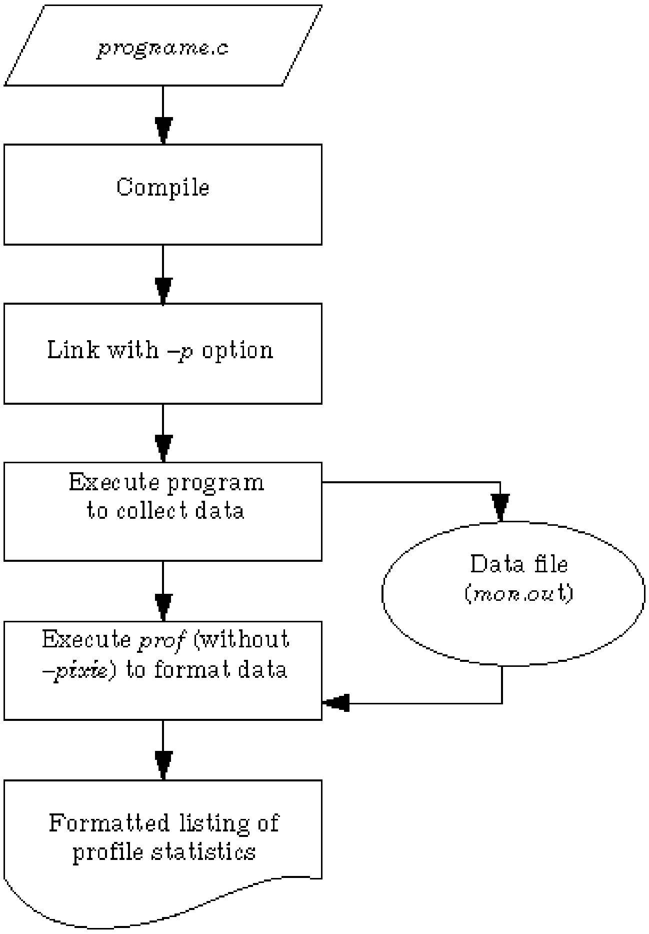

Figure 5-1 How pc Sampling Works

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*. To collect profile data in several files, or to specify a different name for the profile data file, set the environment variable PROFDIR, using the appropriate method from **Table 5-2** .

Table 5-2 Setting a PROFDIR Environment Variable

C shell and tcsh	Bourne shell and Korn shell
setenv PROFDIR *dirname*	PROFDIR=*dirname*; export PROFDIR

Setting the environment variable puts the raw profile data of each invocation of *progname* in files named *dirname/progname.mon.pid*. (You must create a directory named *dirname* before you run the program.) *pid* is the process ID of the executing program; *progname* is the name of the program when invoked.

pc Sampling Frequency

The default frequency of pc sampling is 10 milliseconds. You can change the pc sampling to 1 millisecond by setting the environment variable:

```
PROF_SAMPLING=1
```

Since pc sampling is statistical, this provides more accurate profiling data. However, be aware that considerable kernel overhead is incurred for every process executing on the system while the profiled program is running.

Examples of Using prof to Obtain pc Sampling

The examples in this section illustrate how to use *prof* and its options to obtain pc sampling data.

Example Using prof -pcsample

The following partial listing is an example of pc sampling output from a profiled version of the program *espresso*.

```
Profile listing generated Fri May 13 14:19:12 1994
      with:       prof -pcsmaple espresso
-----------------------------------------------------------------------------
--

samples   time    CPU    FPU   Clock   N-cpu  S-interval Countsize
     4160    42s  R8000  R8010  75.0MHz   1     10.0ms     4(bytes)

Each sample covers 4 bytes for every 10.0ms ( 0.02% of 41.6000sec)

-----------------------------------------------------------------------------
--
    -p[rocedures] using pc sampling.
    Sorted in descending order by the number of samples in each procedure.
    Unexecuted procedures are excluded.
-----------------------------------------------------------------------------
--

samples   time(%)      cum time(%)      procedure (file)

      847   8.5s( 20.4)  8.5s( 20.4)  massive_count
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
      410   4.1s(  9.9)   13s( 30.2)       cofactor
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
```

```
      341   3.4s(  8.2)   16s( 38.4)   setp_implies
(espresso:/usr/people/guest/enjoy/008.espresso/set.c)
      325   3.2s(  7.8)   19s( 46.2)  elim_lowering
(espresso:/usr/people/guest/enjoy/008.espresso/expand.c)
      303     3s(  7.3)   22s( 53.5)    essen_parts
(espresso:/usr/people/guest/enjoy/008.espresso/expand.c)
   ...
```

In the above listing:

- The `samples` column reports the number of samples in each procedure, sorted in descending order. For example, there were `847` samples for the procedure `massive_count`.
- The `time(%)` column lists the number of seconds and percentage of execution time spent in each procedure. For example, there were `8.5` seconds (`20.4`% of execution time) spent in `massive_count`.
- The `cum time(%)` column lists the percentage of the total execution time spent in each procedure. For example, there were `16` seconds (`38.4`% of total execution time) were spent cumulatively in the `massive_count`, `cofactor`, and `setp_implies` procedures. Note that this does not imply that these routines called each other; they may have executed sequentially.
- The `procedure (file)` column prints the procedure name and its source file. For example, the source file containing the `massive_count` procedure is `cofactor.c`.

Example Using prof -pixie -dis -pcsample

You can use the *-dis* option to *prof* to disassemble the analyzed object code and see the number of cycles it takes to execute an instruction. Check the disassembled code for stalls (wasted cycles) and the number of instructions per cycle. For example, partial output looks like this:

```
-----------------------------------------------------------------------------
--
Profile listing generated Fri May 13 14:37:07 1994
      with:       prof -pixie -dis espresso
-----------------------------------------------------------------------------
--

Total cycles    Total Time  Instructions Cycles/inst    Clock
    1573247788        20.98s    2093722528       0.751  75.0MHz
...
/usr/people/guest/enjoy/008.espresso/main.c
main: <0x1001c90c-0x1001e238>
      219 total cycles(0.00%) invoked 1 times, average 219 cycles/invocation
    [20] 0x1001c90c       0x67bdfe40      daddiu  sp,sp,-448      # 0
                  <1 cycle stall for following instruction>
    [20] 0x1001c910       0xffbf0170      sd      ra,368(sp)      # 2
    [20] 0x1001c914       0xffb00158      sd      s0,344(sp)      # 3
    ...
    [152] 0x1000b178 0x1540000d bne t2,zero,0x1000b1b0 # 23
    [152] 0x1000b17c 0x016c5823 subu t3,t3,t4 # 23
    Preceding branch executed 1000 times, taken 0 times.
    ^--- 3000 total cycles(0.01%) executed 1000 times, average 3 cycles.---^
    ...
```

The previous listing shows statistics about the procedure `main` in the file `main.c` and lists the beginning and ending addresses of `main`: <0x1001c90c–0x1001e238>

- The first column lists the line number of the instruction: [20]
- The second column lists the beginning address of the instruction: 0x1001c90c.
- The third column shows the instruction in hexadecimal: 0x67bdfe40.
- The next column reports the assembler form (mnemonic) of the instruction: daddiu sp,sp,–448.
- The last column reports the cycle in which the instruction executed: `# 2`

Other information includes:

- The number of times an above branch was executed and taken:

Preceding branch executed 1000 times, taken 0 times

- The total number of cycles in a basic block and the percentage of the total cycles for that basic block, the number of times the branch terminating that basic block was executed, and the number of cycles for one execution of that basic block:
  300 total cycles(0.01%) executed 1000 times, average3 cycles.
- Any cycle stalls (cycles that were wasted)

For information on cycle stalls and what causes them, see the *MIPS Microprocessor Chip Set User's Guide* for your architecture.

Basic Block Counting

Basic block counting, obtained using the program *pixie*, measures the execution of basic blocks. A basic block is a sequence of instructions that is entered only at the beginning and exits only at the end. This section covers:

- **"Using pixie"**
- **"Obtaining Basic Block Counts"**
- **"Summing Basic Block Count Results"**
- **"Profiling Multiprocessed Executables"**

Using pixie

Use *pixie* to measure the frequency of code execution. *pixie* reads an executable program, partitions it into basic blocks, and writes (instruments) an equivalent program containing additional code that counts the execution of each basic block.

Note that the execution time of an instrumented program is two−to−five times longer than an uninstrumented one. This timing change may alter the behavior of a program that deals with a graphical user interface (GUI), or depends on events such as SIGALARM that are based on an external clock.

pixie Syntax

```
pixie prog_name [options]
```

prog_name	Name of the input program.
options	One of the keywords listed in **Table 5−3** .

pixie Options

Table 5−3 lists *pixie* options. For a complete list of options refer to the *pixie*(1) reference page.

Table 5−3 Options for *pixie*

Name	**Result**
-pixie_file out_file	Specifies a name for the instrumented output file. The default is to remove any leading directory names from the input filename and append *.pixie*.
-counts_file file	Specifies a name for the counts file that is generated while running the instrumented program. The default is to remove any leading directory names from the input filename and append *.Counts*.
-[*no*]*autopixie*	Permits (or prevents) a recursive instrumenting all dynamic shared libraries used by the input file during run time. *pixie* keeps the timestamp and checksum from the original executable. Thus, before instrumenting a shared library, *pixie* checks any *lib.pixie* files that it finds matching the *lib* it is to instrument. If the fields match, they are not instrumented. *pixie* cannot detect shared libraries opened with *dlopen* (and hence does not instrument them). All used DSOs need to be instrumented for the *a.out* to work. The default behavior with shared libraries is *-noautopixie,* and can be overridden with *-autopixie*, which is the default in all other cases.
-[*no*]*liblist*	Prevents (or permits) printing the names and paths of dynamic shared libraries used by the input program during run time. This uses the same default search

	path as *rld* and *prof*. This list is useful to build a dependency list for *makefiles* and shell scripts. *pixie* removes any leading directory names from the input filename and appends *.liblist*. *pixie* cannot detect libraries opened with *dlopen*. The default is *-noliblist*.
-[*no*]*pids*	Appends the process ID number on the end of the *.Counts* file. This is useful if you want to run the program instrumented with *pixie* through a variety of tests. This option is only needed for the *main* program. It will be transferred automatically to the instrumented DSOs during run time. The default is *-nopids*.
-[*no*]*verbose*	Suppresses (or prints) messages summarizing the binary−to−binary translation process. The default is *-noverbose*.

Obtaining Basic Block Counts

Use this procedure to obtain basic block counts. Also refer to **Figure 5−2** , which illustrates how basic block counting works.

1. Compile and link your program. Do not use the *-p* option. The following example uses the input file *myprog.c*.

   ```
   %   cc −o myprog myprog.c
   ```

   The *cc* compiler compiles *myprog.c* into an executable called *myprog*.

2. Run *pixie* to generate the equivalent program containing basic−block−counting code.

   ```
   %   pixie myprog
   ```

   pixie takes *myprog* and writes an equivalent program, *myprog.pixie*, containing additional code that counts the execution of each basic block. *pixie* also writes an equivalent program for each shared object used by the program (in the form: *libname.so.pixie*), containing additional code that counts the execution of each basic block. For example, if *myprog* uses *libc.so.1*, pixie generates *libc.so.1.pixie*.

3. Set the path for your *.pixie* files. *pixie* uses the *rld* search path for libraries (see *rld*(1) for the default paths). If the *.pixie* files are in your local directory, set the path as:

   ```
   % setenv LD_LIBRARY_PATH .
   ```

4. Execute the file(s) generated by *pixie* (*myprog.pixie*) in the same way you executed the original program.

   ```
   %   myprog.pixie
   ```

   This program generates a list of basic block counts in files named *myprog.Counts*. If the program executes a *fork/sproc*, a process ID is appended to the end of the filename (for example, *myprog.Counts.345*) for each process.

5. Run the profile formatting program *prof* specifying the *-pixie* option and the name of the original program.

   ```
   %   prof −pixie myprog myprog.Counts
   ```

   prof extracts information from *myprog.Counts* and prints it in an easily readable format. If multiple *.Counts* files exist, you can use the wildcard character (*) to specify all of the files.

   ```
   %   prof −pixie myprog myprog.Counts*
   ```

Note: Specifying *myprog.Counts* is optional; *prof* searches by default for names having the form *prog_name.Counts*.

You can run the program several times, altering the input data, to create multiple profile data files. See **"Example Using prof -pixie -procedures -clock"** later in this section for an example.

The time computation assumes a "best case" execution; actual execution may take longer. This is because the time includes predicted stalls within a basic block, but not actual stalls that may occur entering a basic block. Also it assumes that all instructions and data are in cache (for example, it excludes the delays due to cache misses and memory fetches and stores).

The complete output of the *-pixie* option is often extremely large. Use the *-quit* option with *prof* to restrict the size of the output. Refer to **"Running the Profiler"** for details about *prof* options.

Include or exclude information on specific procedures within your program by using the *prof* options *-only* or *-exclude* (explained in **Table 5–1**). *prof* timings reflect only time spent in a specific procedure, not time spent including procedures called by that procedure. The *CASEVision/WorkShop* toolset, an optional software product, can show an estimate of inclusive times.

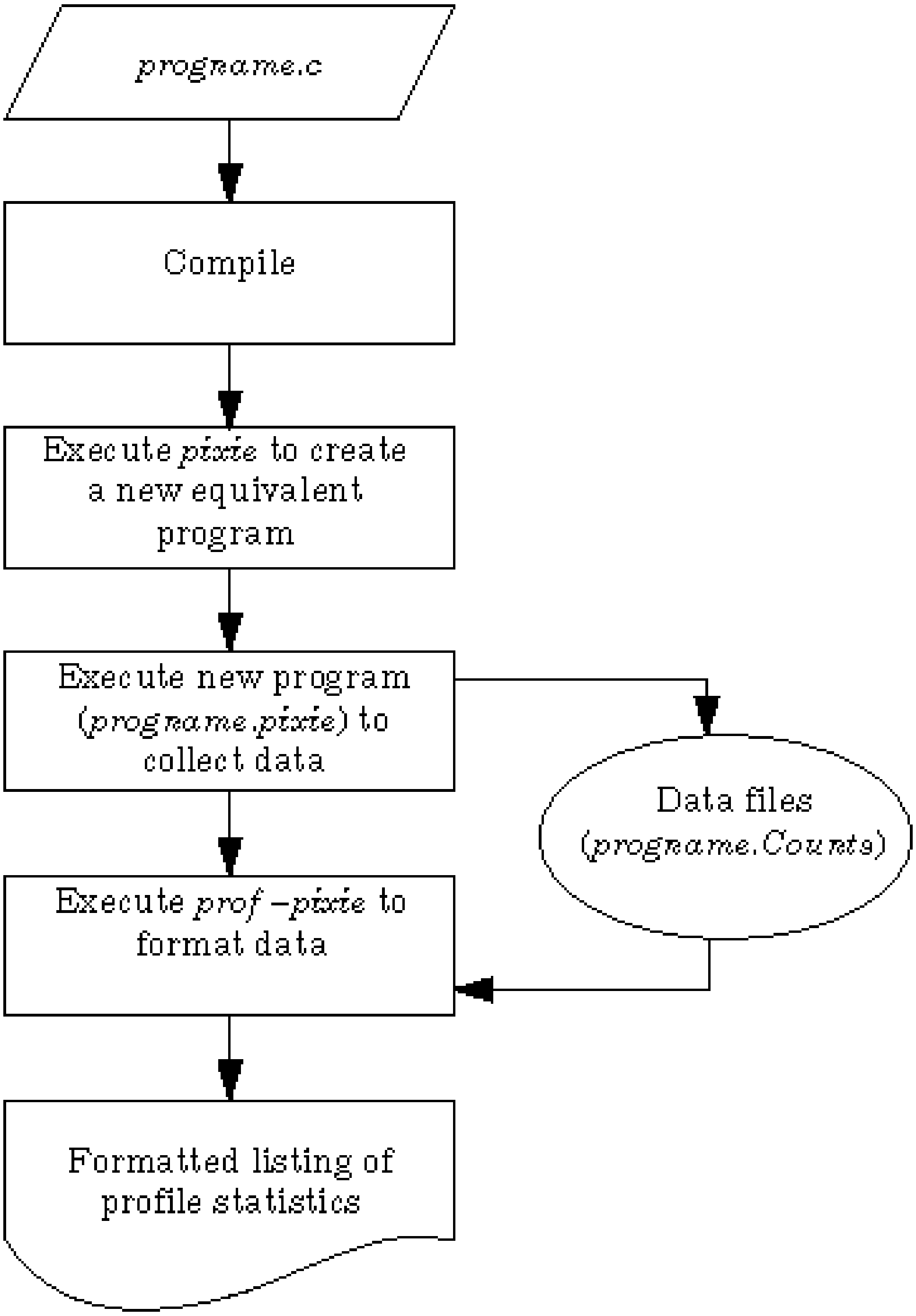

Figure 5–2 How Basic Block Counting Works

Examples of Basic Block Counting

The examples in this section illustrate how to use *prof -pixie* to obtain basic block counting information from a profiled version of a C file, *espresso*.

Example Using prof -pixie -invocations

The partial listing below illustrates the use of the -*i*[*nvocations*] option. For each procedure, *prof* reports the number of times it was invoked from each of its possible callers and lists the procedure(s) that called it.

```
% prof -pixie -i espresso

-----------------------------------------------------------------------------
--
Profile listing generated Fri May 13 14:25:19 1994
      with:       prof -pixie -i espresso
...
*     Sorted in descending order by number of calls per procedure.
*     Unexecuted procedures are excluded.
*     The inst column is a static number of instructions.
*     %coverage column contains the percent inst executed.
-----------------------------------------------------------------------------
--
Total procedure invocations: 12113082

           calls(%)     cum%      inst %coverage procedure (file)

     3055229(25.22)   25.22        26     25.00 full_row
(espresso:/usr/people/guest/enjoy/008.espresso/setc.c)
      966541( 7.98)   33.20        26     25.00 set_or
(espresso:/usr/people/guest/enjoy/008.espresso/set.c)
      772942( 6.38)   39.58        26     25.00 cleanfree
(espresso:/work/irix/lib/libc/libc_64/gen/malloc.c)
      611793( 5.05)   44.63        26     25.00 setp_implies
...
```

The above listing shows the total procedure invocations (calls) during the run: `12113082`.

- The `calls(%)` column reports the number of calls (and the percentage of total calls) per procedure. For example, there were `3055229` calls (or `25.22`% of the total) spent in the procedure *full_row*.
- The `cum%` column shows the cumulative percentage of calls. For example, `25.22`% of all calls were spent in `full_row`.
- The `inst` column shows the number of instructions executed for a procedure. For example, there were `26` instructions in the procedure `full_row`.
- The `%coverage` column reports the percentage of instructions executed. For example, `25.00`% of instructions were executed in `full_row`.
- The `procedure (file)` column lists the procedure and its file. For example, the first line reports statistics for the procedure `full_row` in the file `setc.c`.

Example Using prof -pixie -heavy

The following partial listing shows the source code lines responsible for the largest portion of execution time produced with the *-heavy* option.

```
% prof -pixie -heavy espresso

-----------------------------------------------------------------------------
--
Profile listing generated Fri May 13 14:28:56 1994
      with:       prof -pixie -heavy espresso
-----------------------------------------------------------------------------
```

```
-----------------------------------------------------------------------------
--
...
* -h[eavy] using basic block counts.
*      Sorted in descending order by number of cycles per line.
*      Unexecuted lines are excluded.
*      The insts column contains distinct executed instructions for this line.

-----------------------------------------------------------------------------
--

          cycles(%)   cum %   line insts procedure (file)
    77948528( 4.95%)   4.95%     57    40 cofactor
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
    73800963( 4.69%)   9.65%    213    67 essen_parts
(espresso:/usr/people/guest/enjoy/008.espresso/expand.c)
    53399667( 3.39%)  13.04%     48    29 full_row
(espresso:/usr/people/guest/enjoy/008.espresso/setc.c)
    44723520( 2.84%)  15.88%    137    22 massive_count
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
    38032848( 2.42%)  18.30%    257    39 taut_special_cases
(espresso:/usr/people/guest/enjoy/008.espresso/irred.c)
...
```

The previous partial listing shows basic block counts sorted in descending order. The most heavily used line (`57`) was in procedure `cofactor`.

- The `cycles(%)` column shows the total number of program cycles (and percentage of the total number). For example, there were `77948528` cycles (`4.95`% of the total number) for the procedure `cofactor`.
- The `cum%` column shows the cumulative percentage of the total program cycles. For example, `4.95` % of all program cycles were spent in `cofactor`. The first three procedures used `13.04%` of the total cycles.
- The `line` column lists the line number of the procedure: `57`.
- The `insts` column reports the number of distinct instructions that were executed at least once. For example, line `57` had `40` instructions.

Example Using prof -pixie -lines

The following partial listing, produced using the *-lines* option, shows the execution time spent on each line of code, grouped by procedure.

```
% prof -pixie -lines espresso

-----------------------------------------------------------------------------
--
Profile listing generated Fri May 13 14:30:49 1994
      with:       prof -pixie -lines espresso
-----------------------------------------------------------------------------
--
...
*  -l[ines] using basic-block counts.
*       Grouped by procedure.
*       Major sort on cycles executed per procedure.
*       Minor sort on line numbers within procedure.
*       Unexecuted procedures are execluded.
-----------------------------------------------------------------------------
--
          cycles(%)   cum %   line insts procedure (file)
      856768( 0.05%)   0.05%    121    12 massive_count
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
```

```
    25235712( 1.60%)   1.66%    128    12 massive_count
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
      934656( 0.06%)   1.72%    134    16 massive_count
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
     5963136( 0.38%)   2.10%    135     7 massive_count
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
    20870976( 1.33%)   3.42%    136    13 massive_count
(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
...
```

In the above listing:

- The `cycles(%)` column lists the number of program cycles (and the percentage of the total cycles) for each procedure. For example, there were `856768` program cycles (`0.05`% of the total) for `massive_count`.
- The `cum%` column shows the cumulative percentage of the total program cycles. For example, `0.05` % of all program cycles were spent in `massive_count`.
- The `line` and `insts` columns report the procedure's line number, and number of distinct instructions. For example, for the procedure `massive_count`, lines `121` and `128` each generated `12` instructions that were executed at least once, and line `134` generated `16` instructions that were executed at least once.

Example Using prof -pixie -quit

You can limit the output of *prof* to information on only the most time−consuming parts of the program by specifying the *-quit* option. You can instruct *prof* to quit after a particular number of lines of output, after listing the elements consuming more than a certain percentage of the total, or after the portion of each listing whose cumulative use is a certain amount.

Consider the following sample listing:

```
          cycles(%)  cum %  seconds   cycles bytes procedure(file)
                                     /call /line

   360331656(22.90)  22.90     4.80     4626    93
massive_count(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
   174376925(11.08)  33.99     2.33    15479   108
cofactor(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
   157700711(10.02)  44.01     2.10    43817   123
elim_lowering(espresso:/usr/people/guest/enjoy/008.espresso/expand.c)
   155670642( 9.89)  53.91     2.08    49216   156
essen_parts(espresso:/usr/people/guest/enjoy/008.espresso/expand.c)
    66835754( 4.25)  58.15     0.89      691    76
scofactor(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
    66537017( 4.23)  62.38     0.89       21   156
full_row(espresso:/usr/people/guest/enjoy/008.espresso/setc.c)
    57747597( 3.67)  66.05     0.77     1670    87
taut_special_cases(espresso:/usr/people/guest/enjoy/008.espresso/irred.c)
```

Any one of the following commands eliminates everything from the line starting with 66835754 to the end of the listing:

```
prof −quit 4
prof −quit 5%
prof −quit 53cum%
```

Example Using prof -pixie -procedures

The following partial listing, produced with the *-procedures* option, shows the percentage of execution time spent in each procedure.

```
% prof −pixie −procedures espresso
```

```
--------------------------------------------------------------------------
--
Profile listing generated Fri May 13 14:33:00 1994
      with:       prof -pixie -procedures espresso
--------------------------------------------------------------------------
--
...
*  -p[rocedures] using basic-block counts.
*     Sorted in descending order by the number of cycles executed in each
*     procedure. Unexecuted procedures are not listed.
--------------------------------------------------------------------------
--
          cycles(%)  cum %  seconds   cycles bytes procedure(file)
                                     /call /line

   360331656(22.90)  22.90     4.80     4626    93
massive_count(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
   174376925(11.08)  33.99     2.33    15479   108
cofactor(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
   157700711(10.02)  44.01     2.10    43817   123
elim_lowering(espresso:/usr/people/guest/enjoy/008.espresso/expand.c)
   155670642( 9.89)  53.91     2.08    49216   156
essen_parts(espresso:/usr/people/guest/enjoy/008.espresso/expand.c)
    66835754( 4.25)  58.15     0.89      691    76
scofactor(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
```

In the above listing:

- The `cycles(%)` column lists the number of program cycles (and percentage of the total) used. For example, `massive_count` used `360331656` program cycles (`22.90%` of the total cycles).
- The `cum %` column reports the cumulative total of all cycles used. For example, `massive_count`, `cofactor`, and `elim_lowering` used `44.01%` of the cycles.
- The `seconds` column lists the time used by the procedure (the clock rate, 75 megahertz, is omitted in this example). For example, there were `4.80` seconds used by `massive_count`.
- The `cycles/call` and `bytes/line` columns report the average cycles per call, and the bytes of code per line of source text, respectively. For example, `massive_count` used an average of `4626` cycles per call, and had `93` bytes of generated code per line of source text.
- The `procedure (file)` column lists the procedure, `massive_count`, and its source file, `cofactor.c`.

Example Using prof -pixie -procedures -clock

You can add absolute time information to the output by specifying the clock rate, in megahertz, with the *-clock* option. Partial output looks like this:

```
% prof -pixie -procedures -clock 20 espresso

--------------------------------------------------------------------------
--
Profile listing generated Fri May 13 14:34:55 1994
      with:       prof -pixie -procedures -clock 20 espresso
--------------------------------------------------------------------------
--
...
*  -p[rocedures] using basic-block counts.
*     Sorted in descending order by the number of cycles executed in each
*     procedure. Unexecuted procedures are not listed.
--------------------------------------------------------------------------
--
```

```
          cycles(%)  cum %  seconds    cycles bytes procedure(file)
                                      /call /line

   360331656(22.90)  22.90    18.02     4626    93
massive_count(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
   174376925(11.08)  33.99     8.72    15479   108
cofactor(espresso:/usr/people/guest/enjoy/008.espresso/cofactor.c)
   157700711(10.02)  44.01     7.89    43817   123
elim_lowering(espresso:/usr/people/guest/enjoy/008.espresso/expand.c)
...
```

In the previous listing, the `seconds` column lists the number of seconds spent in each procedure. For example `18.02` seconds were spent in the procedure `massive_count`. The time (computed by the profiler), in seconds, is based on the machine speed specified with the *-clock* option (in megahertz). In this example, the speed specified is `20` megahertz.

Summing Basic Block Count Results

Sometimes a single run of a program does not produce the results you require. You can repeatedly run the version of a program created by *pixie* and vary the input with each run, then use the resulting *.Counts* files to produce a consolidated report.

Use the following procedure to average *prof* results:

1. Compile and link the input file. Do not use the *-p* option. For example, consider the input file *myprog.c*:

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

   The *cc* compiler compiles *myprog.c* and saves the executable as *myprog*.

2. Run the profiling program *pixie*.

   ```
   % pixie myprog -pids
   ```

   pixie generates the modified program *myprog.pixie*.

3. Run the profiled program as many times as desired. Each time you run the program, *pixie* creates a *myprog.Counts.pid* file, with the process ID appended.

   ```
   % myprog.pixie < input1 > output1
   % myprog.pixie < input2 > output2
   % myprog.pixie < input3 > output3
   ```

4. Create the report.

   ```
   % prof -pixie myprog myprog.Counts*
   ```

   prof sums the basic block data in the *.Counts* files to produce the profile report.

Profiling Multiprocessed Executables

You can gather either basic block and pc sampling profile data from executables that use the *sproc* and *sprocsp* system calls, such as those executables generated by POWER Fortran and POWER C. Prepare and run the job using the same method as for uniprocessed executables. For multiprocessed executables, each thread of execution writes its own separate profile data file. View these data files with *prof* like any other profile data files.

The only difference between multiprocessed and regular executables is the way in which the data files are named. When using pc sampling, the data files for multiprocessed executables are named *process_id.prog_name*. When using *pixie*, the data files are named *prog_name.Counts.process_id*. This naming convention avoids the potential conflict of multiple threads attempting to write simultaneously to the same file.

Chapter 6

Optimizing Program Performance

This chapter describes the compiler optimization facilities and their benefits, and explains the major optimizing techniques. Topics covered include:

- **"Optimization Overview"**
- **"Optimization Options"**
- **"Improving Global Optimization"**
- **"Improving Other Optimization"**
- **"Register Allocation"**

Optimization Overview

This section explains optimization benefits and optimization and debugging.

Benefits of Optimization

The primary benefits of optimization are faster running programs and often smaller object code size. However, the optimizer can also speed up development time. For example, you can reduce coding time by leaving it up to the optimizer to relate programming details to execution time efficiency. You can 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. The optimizer may move operations around so that the object code does not correspond in an obvious way to the source code. These changed sequences of code can create confusion when using a debugger.

Using the Optimization Options

This section describes the optimization options and includes:

- **"Optimization Options"**
- **"Controlling Floating Point Optimization"**
- **"Controlling Miscellaneous Optimizations"**
- **"Controlling Global Code Motion"**
- **"Controlling Software Pipelining"**
- **"Controlling the Target Architecture"**
- **"Controlling the Target Environment"**

Optimization Options

Invoke the optimizer by specifying a compiler, such as *cc*(1), with any of the options listed in **Table 6−1** .

Table 6−1 Optimization Options

Option	Result
-O0	Performs no optimization that may complicate debugging. No attempt is made to minimize memory access by keeping values in registers, and little or no attempt is made to eliminate redundant code. This is the default.
-O1	Performs as many local optimizations as possible without affecting compile−time performance. No attempt is made to minimize memory access by keeping values in registers, but much of the locally redundant code is removed.
-O2, -O	Performs extensive global optimization. The optimizations at this level are generally conservative in the sense that they: (1) provide code improvements commensurate with the compile time spent (2) are almost always beneficial (3) avoid changes that affect such things as floating point results

-O3	Performs aggressive optimization. The additional optimization at this level focuses on maximizing code quality even if that requires extensive compile time or relaxing language rules. *-O3* is more likely to use transformations that are usually beneficial but can hurt performance in isolated cases. This level may cause noticeable changes in floating point results due to relaxing expression evaluation rules (see the discussion of floating point optimization and the *-OPT:roundoff* option below).

Refer to the applicable compiler reference page for details on the options.

Controlling Floating Point Optimization

Floating point numbers (the Fortran REAL*n, DOUBLE PRECISION, and COMPLEX*n, and the C float, double, and long double) are inexact representations of ideal real numbers. The operations performed on them are also necessarily inexact. However, the MIPS processors conform to the IEEE 754 floating point standard, producing results as precise as possible given the constraints of the IEEE 754 representations, and the MIPSpro compilers generally preserve this conformance. Note, however, that 128−bit floating point (that is, the Fortran REAL*16 and the C long double) is not precisely IEEE−compliant. In addition, the source language standards imply rules about how expressions are evaluated.

Most code that has not been written with careful attention to floating point behavior does not require precise conformance to either the source language expression evaluation standards or to IEEE 754 arithmetic standards. Therefore, the MIPSpro compilers provide a number of options that trade off source language expression evaluation rules and IEEE 754 conformance against better performance of generated code. These options allow transformations of calculations specified by the source code that may not produce precisely the same floating point result, although they involve a mathematically equivalent calculation.

Two of these options are the preferred controls:

- *-OPT:roundoff* deals with the extent to which language expression evaluation rules are observed, generally affecting the transformation of expressions involving multiple operations.
- *-OPT:IEEE_arithmetic* deals with the extent to which the generated code conforms to IEEE 754 standards for discrete IEEE−specified operations (for example, a divide or a square root).

These options are described below.

-OPT:roundoff=n

The *-OPT:roundoff* option provides control over floating point accuracy and overflow/underflow exception behavior relative to the source language rules.

The *roundoff* option specifies the extent to that optimizations are allowed to affect floating point results, in terms of both accuracy and overflow/underflow behavior. The roundoff value, *n*, has a value in the range 0...3. Roundoff values are described below.

roundoff=0	Do no transformations that could affect floating point results. This is the default for optimization levels *-O0* to *-O2*.
roundoff=1	Allow transformations with limited effects on floating point results. For roundoff, limited means that only the last bit or two of the mantissa is affected. For overflow (underflow), it means that intermediate results of the transformed calculation may overflow within a factor of two of where the original expression may have overflowed (underflowed). Note that effects may be less limited when compounded by multiple transformations.
roundoff=2	Allow transformations with more extensive effects on floating point results. Allow associative rearrangement, even across loop iterations, and distribution of multiplication over addition/subtraction. Disallow only transformations known to cause cumulative roundoff errors or overflow/underflow for operands in a large range of valid floating point values. Reassociation can have a substantial effect on the performance of software pipelined loops by breaking recurrences. This is therefore the default for optimization level *-O3*.

roundoff=3 Allow any mathematically valid transformation of floating point expressions. This allows floating point induction variables in loops, even when they are known to cause cumulative roundoff errors, and fast algorithms for complex absolute value and divide, which overflow (underflow) for operands beyond the square root of the representable extremes.

-OPT:IEEE_arithmetic=n

The *-OPT:IEEE_arithmetic* option controls conformance to IEEE 754 arithmetic standards for discrete operators.

The *-OPT:IEEE_arithmetic* option specifies the extent to which optimizations must preserve IEEE floating point arithmetic. The value *n* must be in the range 1...3. Values are described below.

-OPT:IEEE_arithmetic=1
No degradation: do no transformations that degrade floating point accuracy from IEEE requirements. The generated code may use instructions like *madd*s, which provide greater accuracy than required by IEEE 754. This is the default.

-OPT:IEEE_arithmetic=2
Minor degradation: allow transformations with limited effects on floating point results, as long as exact results remain exact. This option allows use of the MIPS IV *recip* and *rsqrt* operations.

-OPT:IEEE_arithmetic=3
Conformance not required: allow any mathematically valid transformations. For instance, this allows implementation of `x/y` as `x*recip(y)`, or `sqrt(x)` as `x*rsqrt(x)`.

As an example, consider optimizing the Fortran code fragment:

```
INTEGER i, n
REAL sum, divisor, a(n)
sum = 0.0
DO i = 1,n
      sum = sum + a(i)/divisor
END DO
```

At *roundoff=0* and *IEEE_arithmetic=1*, the generated code must do the *n* loop iterations in order, with a divide and an add in each.

Using *IEEE_arithmetic=3*, the divide can be treated like `a(i)*(1.0/divisor)`. On the MIPS R8000, the reciprocal can be done with a *recip* instruction. But more importantly, the reciprocal can be calculated once before the loop is entered, reducing the loop body to a much faster multiply and add per iteration, which can be a single *madd* instruction on the R8000.

Using *roundoff=2*, the loop may be reordered. The original loop takes at least 4 cycles per iteration on the R8000 (the latency of the *add* or *madd* instruction). Reordering allows the calculation of several partial sums in parallel, adding them together after loop exit. With software pipelining, a throughput of nearly 2 iterations per cycle is possible on the R8000, a factor of 8 improvement. (Refer to the description of *-SWP:interleave_reductions* below for another example.)

Consider another example:

```
INTEGER i,n
COMPLEX c(n)
REAL r
DO i = 1,n
       r = 0.1 * i
       c(i) = CABS ( CMPLX(r,r) )
END DO
```

Mathematically, `r` can be calculated by initializing it to 0.0 before entering the loop and adding 0.1 on each iteration. But doing so causes significant cumulative errors because the representation of 0.1is not exact. The complex absolute value is mathematically equal to `SQRT(r*r + r*r)`. However, calculating it this way causes an overflow if `2*r*r` is greater than the maximum REAL value, even though a representable result can be calculated for a much wider range of values of *r* (at greater cost). Both of these transformations are forbidden for *roundoff=2*, but enabled for *roundoff=3*.

Other Options to Control Floating Point Behavior

Other options exist that allow finer control of floating point behavior than is provided by *-OPT:roundoff*. The options may be used to obtain finer control, but they may disappear or change in future compiler releases.

-OPT:div_split[=(*ON*|*OFF*)]

Enable/disable the calculation of `x/y as x*(1.0/y)`, normally enabled by *IEEE_arithmetic=3*. See *-OPT:recip*.

-OPT:fast_complex[=(*ON*|*OFF*)]

Enable/disable the fast algorithms for complex absolute value and division, normally enabled by *roundoff=3*.

-OPT:fast_exp[=(*ON*|*OFF*)]

Enable/disable the translation of exponentiation by integers or halves to sequences of multiplies and square roots. This can change *roundoff*, and can make these functions produce minor discontinuities at the exponents where it applies. Normally enabled by *roundoff>0* for Fortran, or for C if the function *exp()* is labelled intrinsic in *<math.h>* (the default in *-xansi* and *-cckr* modes).

-OPT:fast_sqrt[=(*ON*|*OFF*)]

Enable/disable the calculation of square root as `x*rsqrt(x)` for MIPS IV and above, normally enabled by *IEEE_arithmetic=3*.

-OPT:fold_reassociate[=(*ON*|*OFF*)]

Enable/disable transformations that reassociate or distribute floating point expressions, normally enabled by *roundoff>1*.

-OPT:IEEE_comparisons[=*ON*]

Force comparisons to yield results conforming to the IEEE 754 standard for *NaN* and *Inf* operands, normally disabled. Setting this option disables certain optimizations like assuming that a comparison `x==x` is always TRUE (since it is FALSE if *x* is a *NaN*). It also disables optimizations that reverse the sense of a comparison, for example, turning `"x < y" into "! (x >= y)"`, since both `"x<y"` and `"x>=y"` may be FALSE if one of the operands is a *NaN*.

-OPT:recip[=(*ON*|*OFF*)]

Allow use of the MIPS IV reciprocal instruction for `1.0/y`, normally enabled by *IEEE_arithmetic>=2*. See *-OPT:div_split*.

-OPT:rsqrt[=(*ON*|*OFF*)]

Allow use of the MIPS IV reciprocal square root instruction for `1.0/sqrt(y)`, normally enabled by *IEEE_arithmetic>=2*.

-TARG:madd[=(*ON*|*OFF*)]

The MIPS IV architecture supports fused multiply−add instructions, which add the product of two operands to a third, with a single roundoff step at the end. Because the product is not separately rounded, this can produce slightly different (but more accurate) results than a separate multiply and add pair of instructions. This is normally enabled for *-mips4*.

Debugging Floating−Point Problems

The options above can change the results of floating point calculations, causing less accuracy (especially *-OPT:IEEE_arithmetic*), different results due to expression rearrangement (*-OPT:roundoff*), or NaN/Inf results in new cases. Note that in some such cases, the new results may not be worse (that is, less accurate) than the old, they just may be different. For instance, doing a sum reduction by adding the terms in a different order is likely to produce a different result. Typically, that result is not less accurate, unless the original order was carefully chosen to minimize roundoff.

If you encounter such effects when using these options (including *-O3*, which enables *-OPT:roundoff=2* by default), first attempt to identify the cause by forcing the safe levels of the options: *-OPT:IEEE_arithmetic=1:roundoff=0*.

When you do this, *do not* have the following options explicitly enabled:

-OPT:div_split
-OPT:fast_complex
-OPT:fast_exp
-OPT:fast_sqrt
-OPT:fold_reassociate
-OPT:recip
-OPT:rsqrt

If using the safe levels works, you can either use the safe levels or, if you are dealing with performance−critical code, you can use the more specific options (for example, *div_split*, *fast_complex*, and so forth) to selectively disable optimizations. Then you can identify the source code that is sensitive and eliminate the problem. Or, you can avoid the problematic optimizations.

Controlling Miscellaneous Optimizations

The following *-OPT* options allow control over a variety of optimizations.

-OPT:space The MIPSpro compilers normally make optimization decisions based strictly on the expected execution time effects. If code size is more important, use this option. One of its effects is to cause most subprogram exits to go though a single exit path, with a single copy of register restores, result loads, and so forth.

-OPT:alias=name

The compilers must normally be very conservative in optimization of memory references involving pointers (especially in C), since aliases (that is, different ways of accessing the same memory) may be very hard to detect. This option may be used to specify that the program being compiled avoids aliasing in various ways. *alias* options are:

1.

alias=any The compiler assumes that any pair of memory references may be aliased unless it can prove otherwise. This is the default.

alias=typed The compiler assumes that any pair of memory references that reference distinct types in fact reference distinct data. For example, consider the code:

```
void dbl ( int *i, float *f ) {
       *i = *i + *i;
       *f = *f + *f;
}
```

The compiler assumes that `i` and `f` point to different memory, and produces an overlapped schedule for the two calculations.

alias=unnamed The compiler assumes that pointers never point to named objects. For example, consider the code:

```
float g;
void dbl ( float *f ) {
       g = g + g;
       *f = *f + *f;
}
```

The compiler assumes that `f` cannot point to `g`, and produces an overlapped schedule for the two calculations.

This option also implies the *alias=typed* assumption. Note that this is the default assumption for the pointers implicit in Fortran dummy arguments according to the ANSI standard.

alias=restrict The compiler assumes a very restrictive model of aliasing, where no two pointers ever point to the same memory area. For example, consider the code:

```
void dbl ( int *i, int *j ) {
       *i = *i + *i;
       *j = *j + *j;
}
```

The compiler assumes that `i` and `j` point to different memory, and produces an

overlapped schedule for the two calculations.
Although this is a very dangerous option to use in general, it may produce significantly better code when used for specific well−controlled cases where it is known to be valid.

Controlling Loop Unrolling

The following options control loop unrolling in the MIPSpro optimizer (that is, making multiple copies of a loop body to minimize the loop overhead or to expose more instruction parallelism). Unrolling is subject to a number of limits in the optimizer, intended to balance the run−time benefits against code expansion. These options allow you to modify those limits when they can be improved. Note that loops expected to be software pipelined are subject to similar options in the *-SWP* group.

-OPT:unroll_times_max=n
The optimizer normally unrolls loops at most 2 times (*-mips4*) or 4 times (*-mips3*), unless it can unroll them completely. This option modifies the default limit.

-OPT:unroll_size=n
The optimizer normally unrolls loops only to the extent that the resulting unrolled loop body contains at most 320 instructions. This option modifies the default limit.

-OPT:unroll_bblimit=n
The optimizer normally unrolls loops containing up to 10 basic blocks. This option modifies the default limit. (A basic block is a sequence of code between branches and labels; it has single entry at the top and a single exit at the bottom.)

Controlling Global Code Motion

An important optimization performed by the MIPSpro compilers is Global Code Motion (GCM). It is intended to improve the overall execution time of programs by redistributing the instructions among the basic blocks along an execution path to improve instruction parallelism and make better use of the machine resources.

Traditional global optimizers avoid moving instructions in cases that may cause them to be executed along control flow paths where they would not have been in the original program. However, GCM performs such motion, called *speculative code motion* because the instructions moved are executed based on speculation that they actually prove useful. By default, GCM is very conservative in its speculations. Most of the options in the *-GCM:* group control the sorts of speculation that are to be allowed.

Valid speculative code motion must normally avoid moving operations that may cause run−time traps. As a result, turning off certain traps at run time enables more motion. See the target environment option *-TENV:X=n* for general control over the exception environment.

-GCM:aggressive_speculation[=(*ON*|*OFF*)]
GCM normally tries not to move instructions to basic blocks that are already using most of the instruction execution resources available, since doing so will likely extend the execution time of the block. This option minimizes that bias, which often helps floating point intensive code.

-GCM:array_speculation[=(*ON*|*OFF*)]
A form of speculation that is often very effective is called *bottom loading*. It involves moving instructions from the top of a loop body to both the block before the loop (for the first iteration) and to the end of the loop body (which executes them at the end of one iteration so that they are ready early for the next iteration). Bottom loading, however, means that the instructions are executed one or more extra times in the last iteration(s) of the loop. If the instructions moved are loading elements of an array, this may cause extra accesses beyond the end of the array.
This option enables such accesses a small distance beyond the end of the array. The compiler attempts to pad arrays to guarantee that such accesses won't cause memory faults, but it may not always be able to do so, so this option must be used with care.

-GCM:pointer_speculation[=(*ON*|*OFF*)]
This option allows speculative motion of loads of two kinds, both involving pointer usage. The first is to allow motion of loads through pointers that may be NULL. In order to prevent this from causing memory faults, the compiler causes page zero of

references from causing faults, avoid this option during debugging.
The second form involves moving a reference like `*(p+n)`, for a small integer `n`, to a block that already contains a reference to `*p`. The assumption involved is that if `p` is a valid address, `p+n` is too.
Consider an example:

```
...
if ( p->next != NULL ) {
       sum += p->next->val;
} else {
       sum += p->final_val;
}
...
```

If this option is set, the load of `p->next->val` can be moved before the `if` (it is through a potentially NULL pointer), as can the load of `p->final_val` (it is offset by a small amount from the `p->next` reference).

-GCM:static_load_speculation[=(*ON*|*OFF*)]

Allow the speculative motion of loads from static data areas. Since such areas are known to be allocated, such motion cannot cause new faults except page faults or cache misses. As a result, it is generally safe, although it may hurt performance if it causes unnecessary page faults or cache misses.

-GCM:prepass[=(*ON*|*OFF*)]
-GCM:postpass[=(*ON*|*OFF*)]

GCM is implemented in two passes, one before instruction scheduling and register allocation, and one after, currently enabled at optimization *-O2* and above. These options allow either or both of the passes to be enabled or disabled. The two passes are complementary in effect, often improving code more in combination than the sum of their individual improvements, so enabling just one is not recommended.

Controlling Software Pipelining

Software pipelining (SWP), an important optimization for the inner loops of programs, can cause dramatic improvement by rearranging the loops to overlap calculations from multiple iterations. This iterative process searches for an effective schedule and then for a workable allocation of registers, and then retries if either step fails. Some of the options in the *-SWP* group control that process. Other options control how the loop body is prepared for the attempt, for example, by unrolling.

Many important loop preparation transformations involve reassociation of floating point values. See the discussion of floating point optimization above, especially the **"-OPT:roundoff=n"** option.

SWP must normally be careful during the initial and final iterations of a loop to not perform extra operations that may cause run−time traps. It must be similarly careful if early exits from a loop (that is, before the initially calculated trip count is reached) are possible. Turning off certain traps at run time can give it more flexibility, producing better schedules and/or simpler wind−up/wind−down code. See the target environment option *-TENV:X=n* for general control over the exception environment.

-SWP:=(*ON*|*OFF*)

Enable/disable SWP (normally enabled at *-O3*).

-SWP:back_substitution[=(*ON*|*OFF*)]

The iteration interval (II) of a pipelined loop, that is the frequency at which new iterations are started, is constrained by circular data dependencies across iterations, called recurrences. This option, ON by default, allows transformations that make recurrences less severe by substituting the expression that defines a variable for the variable. For example, consider the code:

```
DO i=1,n
a[i] = a[i-1] + 5.0
END DO
```

Without back−substitution, each iteration must wait for the previous iteration's add to complete, yielding a best case 2 of 4 cycles per iteration on the R8000.
Back−substitution can transform the loop to code equivalent to:

```
DO i=1,n
a[i] = a[i-8] + 40.0
END DO
```

With appropriate initialization, this version can achieve an effective II of nearly 0.5 cycles.

-SWP:backtracks=n

SWP often backtracks and tries again when it fails to find a workable schedule. This option controls the limit on how many times it does so. Increasing the limit improves its chances of success; decreasing it may improve the compilation time required.

-SWP:body_ins_count_max=n

SWP is not attempted for loop bodies containing more than *n* instructions (the default is 100, 0 for no limit). Larger loop bodies are less likely to be successfully pipelined, and take more compilation time in the attempt, so this is another tradeoff of (potential) code improvement vs. compile time.

Loop bodies are also normally unrolled in preparation for SWP. This also limits the unrolling, since loops are not unrolled to more than *n* instructions in the unrolled body. Unrolling is also constrained by the *unroll_times_max* option described below. (Unrolling of loop bodies not expected to be software pipelined is controlled separately by *-OPT:unroll_size* and *-OPT:unroll_times_max*.)

-SWP:fix_recurrences[=(*ON*|*OFF*)]

This option controls both of the transformations controlled by *back_substitution* and *interleave_reductions*. See their descriptions.

-SWP:if_conversion[=(*ON*|*OFF*)]

SWP generally works much better on loop bodies without internal branches caused by conditional execution. This option causes conditional branches to be removed when possible by using conditional move instructions (MIPS IV) and equivalents. For example, consider the code:

```
DO i=1,n
       IF ( a(i) .LT. b(i) ) THEN
          c(i) = a(i)
       ELSE
          c(i) = b(i)
       END IF
END DO
```

The loop body is compiled for MIPS IV as:

```
ldc1 $f0,a(i)
ldc1 $f1,b(i)
c.lt.s cc,$f0,$f1
movf.s $f0,$f1,cc
sdc1 $f0,c(i)
```

Note that no conditional branches occur in the code. This option is ON by default for MIPS IV targets only.

-SWP:interleave_reductions[=(*ON*|*OFF*)]

This option, ON by default, has the same motivation as back–substitution. It allows transformations that make recurrences arising from reductions less severe by interleaving multiple threads of the reduction and then piecing them together at the end of the loop. For example, consider the code to sum an array:

```
DO i=1,n
       sum = sum + a(x)
END DO
```

Without interleaving, each iteration must wait for the previous iteration’s add to complete, yielding a best case II of 4 cycles per iteration on the R8000. Interleaving can transform the loop to something equivalent to:

```
DO i=1,n,8
```

```
        sum1 = sum1 + a(i)
        sum2 = sum2 + a(i+1)
        sum3 = sum3 + a(i+2)
        sum4 = sum4 + a(i+3)
        sum5 = sum5 + a(i+4)
        sum6 = sum6 + a(i+5)
        sum7 = sum7 + a(i+6)
        sum8 = sum8 + a(i+7)
END DO
sum = sum1+sum2+sum3+sum4+sum5+sum6+sum7+sum8
```

This version can achieve an effective II of nearly 0.5 cycles. These transformations generally require *-OPT:roundoff=2* or better.

-SWP:trip_count_min=n

SWP is not attempted for loops with trip counts known to be smaller than *n* (the default is 5). The limit is applied via a run−time test for cases where the trip count is not known at compile time. Sometimes, a longer loop body can be profitably pipelined even with a smaller trip count, enabled by this option.

-SWP:unroll_times_max=n

This option controls the maximum number of times inner loop bodies are unrolled before attempting pipelining. The default is 2 for MIPS IV, and 1 for MIPS I−III. Unrolling is also constrained by the *body_ins_count_max* option described above. (Unrolling of loop bodies not expected to be software pipelined is controlled separately by *-OPT:unroll_size* and *-OPT:unroll_times_max*.)

The *-S* option to the compiler provides information about software pipelining, the loops it works on, and the results. See **"General Options for Compiler Drivers"** for more information.

Controlling the Target Architecture

Some options control the target architecture for which code is generated; most of the options are in the *-TARG*: option group. The options are described below.

-32 or *-64*

This option determines the base ABI to be assumed, either the 32-bit ABI for MIPS I/II targets, or the new 64−bit ABI for MIPS III/IV targets.

-mips[*1234*] or *-TARG:isa=mips*[*1234*]

This option identifies the instruction set architecture (ISA) to be used for generated code. It defaults to MIPS I for 32−bit compilations, and to MIPS III for 64−bit compilations. However, you can specify higher ABIs for either, at the cost of producing an executable that will not run on the earlier architectures. The linker (*ld*) does not understand the *-TARG* group, so you must use the short form if linking is to be invoked as part of the compilation.

-TARG:madd[=(*ON*|*OFF*)]

This option enables/disables use of the multiply/add instructions for MIPS IV targets. These instructions multiply two floating point operands and then add (or subtract) a third with a single roundoff of the final result. They are therefore slightly more accurate than the usual discrete operations, and may cause results not to match baselines from other targets. This option may be used to determine whether observed differences are due to *madd*s. This option is ON by default for MIPS IV targets; otherwise, it is ignored.

-TARG:processor=(*r3000*|*r4000*|*r8000*)

This option identifies the probable execution target of the code to be generated; it will be scheduled for optimal execution on that target. This option does not affect the ABI and/or ISA selected by the options described above (that is, it has no effect on which target processors can correctly execute the program, but just on how well they do so).

The command below produces MIPS III code conforming to the 64−bit ABI (and therefore executable on any MIPS III or above processor), which is optimized to run on the R8000.

```
cc −64 −TARG:proc=r8000 ...
```

Controlling the Target Environment

Generated code is affected by a number of assumptions about the target software environment. The *-TENV* options tell the compiler what assumptions it can make, and sometimes what assumptions it should enforce.

The first *-TENV* options are concerned with the shared code model.

-TENV:large_GOT[=(*ON*|*OFF*)]
-TENV:small_GOT[=(*ON*|*OFF*)]

Shared code and dynamic shared objects (DSOs) require the use of a global offset table (GOT) containing addresses of static data and subprograms at run time. A dedicated register (*$gp*) points to the GOT at run time, and the code can load these addresses from the GOT without being dependent on its actual virtual address. If the GOT is less than 64KB in size, those loads can all be single instructions; otherwise they require adding a constructed offset to *$gp*. These options choose one of these cases (the default is *small_GOT*).

As mentioned in the GCM and SWP discussions above, being able to execute instructions speculatively can make a significant difference in the quality of generated code. What instructions can be executed speculatively depends on the trap enable state at run time.

-TENV:X=n

Specify the level (0 to 5, the default is 1) of enabled traps that are assumed (and enforced) for purposes of performing speculative code motion. At level 0, no speculation is done. At level 1, only safe speculative motion may be done, assuming that the IEEE 754 underflow and inexact traps are disabled. At level 2, all IEEE 754 floating point traps are disabled except divide by zero. At level 3, divide by zero traps are disabled. At level 4, memory traps may be disabled or dismissed by the operating system. At level 5, any traps may be disabled or ignored.

Use nondefault levels with great care. Disabling traps eliminates a useful debugging tool, since the problems that cause them are detected later (often much later) in the execution of the program. In addition, many memory traps can’t be avoided outright, but must be dismissed by the operating system after they occur. As a result, level 4 or 5 speculation can actually slow down a program significantly if it causes frequent traps.

Disabling traps in one module requires disabling them for the entire program. Programs that make use of level 2 or above should not attempt explicit manipulation of the hardware trap enable flags. Furthermore, libraries (including DSOs) being built for incorporation in many client programs should generally avoid using this option, since using it makes debugging of the client programs difficult, and can prevent them from safely making use of the hardware trap enables.

The last set of environment options control the alignment of data. These options include:

-align8 | *-align16* | *-align32* | *-align64*

The MIPS architectures perform memory references much more efficiently if the data referenced is naturally aligned (that is, if 4–byte objects are at 4–byte–aligned address, and so forth). By default, the compilers allocate well–aligned data, a requirement of the ABI for C. However, code ported from other architectures without alignment constraints may require less restricted alignment. The ANSI Fortran standard essentially requires maximum alignment of 4 bytes (32 bits), although it is unusual for code to actually depend on this.

The *-align* options specify a maximum alignment (in bits) to be forced in allocating data. The MIPSpro compilers default to *-align32* for MIPS I Fortran, to *-align64* for MIPS II–IV Fortran, and to ABI alignment (up to 128 bits for long double) for C.

-TENV:align_aggregates=n

The ABI specifies that aggregates (that is, *struct*s and *array*s) be aligned according to the strictest requirements of their components (fields or elements). Thus, an array of *short int*s (or a struct containing only *short int*s or *char*s) normally is 2–byte aligned. However, some code (non–ANSI conforming) may reference such objects assuming greater alignment, and some code (for example, *struct* assignments) may be more efficient if the objects are better aligned. This option specifies that any aggregate of size at least *n* is at least *n*–byte aligned. It does not affect the alignment of aggregates, which are themselves components of larger aggregates.

-TENV:misalignment=n

assume that the memory references it generates are well aligned. If *n=1*, it assumes well–aligned references unless it can determine otherwise (except in circumstances like struct copies where the ABI alignment requirements don't normally imply alignment).
If *n=2*, it always analyzes alignment based on the actual expression used, and trusts casts in the source code to reflect actual alignment. For example, consider the expression:

```
*(type_a *) &b
```

The declared type of `b` implies a certain minimum alignment, as does the cast to `type_a`; the compiler uses the maximum of the two.
If `n=3`, the compiler also analyzes alignment, but does not trust casts. In the above example, it assumes the alignment of the declared type of `b`.

Improving Global Optimization

This section describes the global optimizer and contains coding hints. Specifically this section includes:

- **"Overview of the Global Optimizer"**
- **"Optimizing C and Fortran Programs"**
- **"Improving Other Optimization"**
- **"Register Allocation"**

Overview of the Global Optimizer

The global optimizer is part of the compiler back end. It improves the performance of object programs by transforming existing code into more efficient coding sequences. The optimizer distinguishes between C, C++, and Fortran programs to take advantage of the various language semantics.

Optimizing C and Fortran Programs

The following suggestion applies to both C and Fortran programs.

Do not use indirect calls (that is, calls via function pointers, including those passed as subprogram arguments). Indirect calls may cause unknown side effects (for instance, changing global variables) reducing the amount of optimization possible.

Optimizing C Programs

The following suggestions apply to C programs only.

Return values. Use functions that return values instead of pointer parameters.

Do while. Use *do while* when possible, instead of *while* or *for*. For *do while*, the optimizer does not have to duplicate the loop condition in order to move code from within the loop to outside the loop.

Unions. Avoid unions that cause overlap between integer and floating point data types. The optimizer can not assign such fields to registers.

Local variables. Use local variables and void 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. Pass parameters by value instead of passing by reference (pointers) or using global variables. Reference parameters have the same performance–degrading effects as the use of pointers (see below).

Pointers and aliasing. *Aliases* occur when there are multiple ways to reference the same data object. For instance, when the address of a global variable is passed as a subprogram argument, it may be referenced either using its global name, or via the pointer. The compiler must be conservative when dealing with objects that may be aliased, for instance keeping them in memory instead of in registers, and carefully retaining the original source program order for possibly aliased references.

Pointers in particular tend to cause aliasing problems, since it is often impossible for the compiler to identify their target objects. Therefore, you can help the compiler avoid possible aliases by introducing local variables to store the values obtained from dereferenced pointers. Indirect operations and calls affect dereferenced values, but do not affect local variables. Therefore, local variables can be kept in registers. The following example shows how the proper placement of pointers and the elimination of aliasing produces better code.

Example of Pointer Placement and Aliasing

In the example below, if `len>10` (for instance because it is changed in another function before calling zero), `*p++ = 0` will eventually modify len. Therefore, the compiler cannot place `len` in a register for optimal performance. Instead, the compiler must load it from memory on each pass through the loop.

Consider the following source code:

```
char a[10];
int len = 10;
void
zero()
{
     char *p;
     for (p= a; p != a + len; ) *p++ = 0;
}
```

The generated assembly code looks like this:

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

You can increase the efficiency of this example by using subscripts instead of pointers, or by using local variables to store unchanging values.

Using subscripts instead of pointers. Using subscripts in the procedure `azero` (as shown below) eliminates aliasing. The compiler keeps the value of `len` in a register, saving two instructions. It still uses a pointer to access `a` efficiently, even though a pointer is not specified in the source code. For example, consider the following source code:

```
char a[10];
int len = 10;
void azero()
{
      int i;
      for ( i = 0; i != len; i++ )
      a[i] = 0;
}
```

The generated assembly code looks like this:

```
         for (i = 0; i != len; i++ ) a[i] = 0;
         move   $2, $0          # i = 0
         beq    $4, 0, $37      # len != a
         la     $5, a
$36:
```

```
          sb      $0, 0($5)        # *a = 0
          addu    $2, $2, 1        # i++
          addu    $5, $5, 1        # a++
          bne     $2, $4, $36      # i != len
$37:
```

Using local variables. Using local (automatic) variables or formal arguments instead of static or global prevents aliasing and allows the compiler to allocated them in registers.

For example, in the following code fragment, the variables `loc` and `form` are likely to be more efficient than `ext*` and `stat*`.

```
extern int ext1;
static int stat1;

void p ( int form )
{
     extern int ext2;
     static int stat2;
     int loc;
     ...
}
```

Write straightforward code. For example, do not use **++** and **– –** operators within an expression. Using these operators for their values, rather than for their side–effects, often produces bad code.

Addresses. Avoid taking and passing addresses (& values). Using addresses creates aliases, makes the optimizer store variables from registers to their home storage locations, and significantly reduces optimization opportunities that would otherwise be performed by the compiler.

VARARG/STDARG. Avoid functions that take a variable number of arguments. The optimizer saves all parameter registers on entry to VARARG or STDARG functions. If you must use these functions, use the ANSI standard facilities of *stdarg.h*. These produce simpler code than the older version of *varargs.h*

Improving Other Optimization

The global optimizer processes programs only when you specify the *-O2* or *-O3* option at compilation. However, the code generator phase of the compiler always perform certain optimizations.

This section contains coding hints that increase optimization for the other passes of the compiler.

C and Fortran Programs

The following suggestions apply to both C and Fortran programs:

- Use tables rather than *if–then–else* or *switch* statements. For example:

  ```
  typedef enum { BLUE, GREEN, RED, NCOLORS } COLOR;
  ```

  Instead of:

  ```
  switch ( c ) {
       case CASE0: x = 5; break;
       case CASE1: x = 10; break;
       case CASE2: x = 1; break;
  }
  ```

  Use:

  ```
  static int Mapping[NCOLORS] = { 5, 10, 1 };
  ...
  x = Mapping[c];
  ```

- As an optimizing technique, the compiler puts the first eight parameters of a parameter list into registers where they may remain during execution of the called routine. Therefore, always declare, as the first eight parameters, those variables that are most frequently manipulated in the called

routine.
- Use word−size scalar variables instead of smaller ones. This practice can take more data space, but produces more efficient code.

C Programs Only

The following suggestions apply to C programs only:
- Rely on *libc.so* functions (for example, *strcpy*, *strlen*, *strcmp*, *bcopy*, *bzero*, *memset*, and *memcpy*). These functions are carefully coded for efficiency.
- Use a signed data type, or cast to a signed data type, for any variable that does not require the full unsigned range and must be converted to floating−point. For example:

  ```
  double d;
  unsigned int u;
  int i;
  /* fast */ d = i;
  /* fast */ d = (int)u;
  /* slow */ d = u;
  ```

  Converting an unsigned type to floating−point takes significantly longer than converting signed types to floating−point; additional software support must be generated in the instruction stream for the former case.
- Use signed *int*s in 64−bit code if they may appear in mixed type expressions with *long int*s. Since the hardware automatically sign−extends the results of most 32−bit operations, this may avoid explicit zero−extension code. For example:

  ```
  unsigned int ui;
  signed int i;
  long int li;
  /* fast */ li += i;
  /* fast */ li += (int)ui;
  /* slow */ li += ui;
  ```

Register Allocation

The MIPS architecture emphasizes the use of registers. Therefore, register usage has a significant impact on program performance.

The optimizer allocates registers for the most suitable data items, taking into account their frequency of use and their locations in the program structure. In addition, the optimizer assigns values to registers in such a way as to minimize shifting around within loops and during procedure invocations.

Chapter 7

Coding for 64−Bit Programs

This chapter provides information about ways to write/update your code so that you can take advantage of the Silicon Graphics implementation of the IRIX 64−bit operating system. Specifically, this chapter describes:
- **"Porting Code to 64−Bit Silicon Graphics Systems"**
- **"Coding Assumptions to Avoid"**
- **"Guidelines for Writing Code for 64−Bit Silicon Graphics Platforms"**

Porting Code to 64−Bit Silicon Graphics Systems

This section covers porting code to 64−bit Silicon Graphics systems, including:
- **"Using Data Types"**
- **"Using Predefined Types"**
- **"Using Typedefs"**

You can find additional information about porting to 64−bit Silicon Graphics systems in the *Application*

Application Porting and Transition Guide for Power Challenge Systems.

Using Data Types

Data types and sizes are listed in **Table 7−1** .

Table 7−1 Data Types and Sizes

Data Type	32 Bit	64 Bit
char	8	8
short	16	16
int	32	32
long	32	64
long long	64	64
pointer	32	64
float	32	32
double	64	64
long double*	64	128
void*	32	64

Note that in 64−bit mode, types *long* and *int* have different sizes and ranges; a *long* always has the same size as a *pointer*. A *pointer* (or address) has 64−bit representation in 64−bit mode and 32−bit representation in 32−bit mode. An *int* object has a smaller range than a *pointer* type in 64−bit mode.

Characteristics of integral types and floating point types are defined in the standard files *limits.h* and *float.h*.

Using Predefined Types

The *cc*, *CC*, and *as* compiler drivers produce predefined macros listed in **Table 7−2** . These macros are used in *sys/asm.h*, *sys/regdef.h*, and *sys/fpregdef.h*.

Table 7−2 Predefined Macros

32−Bit Executables	64−Bit Executables
-D_MIPS_FPSET=16	-D_MIPS_FPSET=32
-D_MIPS_ISA=_MIPS_ISA_MIPS1	-D_MIPS_ISA=_MIPS_ISA_MIPS3
-D_MIPS_SIM=_MIPS_SIM_ABI32	-D_MIPS_SIM=_MIPS_SIM_ABI64
-D_MIPS_SZINT=32	-D_MIPS_SZINT=32
-D_MIPS_SZLONG=32	-D_MIPS_SZLONG=64
-D_MIPS_SZPTR=32	-D_MIPS_SZPTR=64

_MIPS_FPSET describes the number of floating point registers. The 64−bit compilation mode makes use of the extended floating point registers available on the R4000 and R8000 processors.

MIPS_ISA determines the MIPS Instruction Set Architecture. MIPS_ISA_MIPS1 and MIPS_ISA_MIPS3 are the defaults for 32 bits and 64 bits, respectively. For example:

```
/* Define a parameter for the integer register size: */
#if (_MIPS_ISA == _MIPS_ISA_MIPS1 || _MIPS_ISA == _MIPS_ISA_MIPS2)
#define SZREG          4
#else
#define SZREG          8
#endif
```

MIPS_SIM determines the MIPS Subprogram Interface Model, which describes the subroutine linkage convention and register naming/usage convention.

_MIPS_SZINT, _MIPS_SZLONG, and _MIPS_SZPTR define the size of types *int*, *long*, and *pointer*, respectively.

The 64−bit MIPSpro compiler drivers generate 64−bit *pointer*s and *long*s and 32−bit *int*s. Therefore, assembler code that uses either *pointer* or *long* types must be converted to use *double−word* instructions for MIPS III code(*-64*), and must continue to use *word* instructions for MIPS I and MIPS II code (*-32*).

Also, new subroutine linkage conventions and register naming conventions exist. The compiler predefined macro _MIPS_SIM enables macros in *sys/asm.h* and *sys/regdef.h*.

Eight argument registers exist: *$4* through *$11*. Four additional argument registers replace the *temp* registers in *sys/regdef.h*. These *temp* registers are not lost, however, as the argument registers can serve also as scratch registers, with certain constraints.

In the _MIPS_SIM_ABI64 model, registers *t4* through *t7* are not available, so any code using these registers does not compile. Similarly, registers *a4* through *a7* are not available under the _MIPS_SIM_ABI32 model.

If you are converting assembler code, the new registers *ta0*, *ta1*, *ta2*, and *ta3* are available under both _MIPS_SIM models. These alias with registers *t4* through *t7* in 32−bit mode, and with registers *a4* through *a7* in 64−bit mode.

Note that the caller no longer has to reserve space for a called function in which to store its arguments. The called routine allocates space for storing its arguments on its own stack, if desired. The NARGSAVE macro in *sys/asm.h* facilitates this.

Using Typedefs

This section describes *typedef*s that you can use to write portable code for a range of target environments, including 32− and 64−bit workstations as well as 16− and 32−bit PCs. These *typedef*s are enabled by compiler−predefined macros (listed in **Table 7−2**), and are in the file *inttypes.h*. (This discussion applies to C, although the same macros are predefined by the C++ compiler.)

Portability problems exist because an *int* (32 bits) is no longer the same size as a *pointer* (64 bits) and a *long* (64 bits) in 64−bit programs. *Typedef*s free you from having to know the underlying compilation model or worry about type sizes. In the future, if that model changes, the code should still work.

Typically, you want source code that you can compile either in 32− or 64−bit mode. (In this discussion, 32−bit mode means *-mips1* or *-mips2*; 64−bit mode means *-mips3* or *-mips4*.)

The following *typedef*s are defined in *inttypes.h*:

```
typedef signed char   int8_t;
typedef unsigned char   uint8_t;
typedef signed short  int16_t;
typedef unsigned short  uint16_t;
typedef signed int  int32_t;
typedef unsigned int  uint32_t;
typedef signed long long int  int64_t;
typedef unsigned long long int  uint64_t;
typedef signed long long int  intmax_t;
typedef unsigned long long int  uintmax_t;
typedef signed long int  intptr_t;
typedef unsigned long int  uintptr_t;
```

intmax_t and *uintmax_t* are guaranteed to be the largest integer type supported by this implementation. Use them in code that must be able to deal with any integer value.

intptr_t and *uintptr_t* are guaranteed to be exactly the size of a *pointer*.

Coding Assumptions to Avoid

Most porting problems come from assumptions, implicit or explicit, about either absolute or relative sizes of the *int*, *long int*, or *pointer* types in code.

To avoid porting problems, examine code that assumes:

- **"sizeof(int) == sizeof(void *)"**
- **"sizeof(int) == sizeof(long)"**
- **"sizeof(long) == 4"**
- **"sizeof(void *) == 4"**
- **"Implicitly Declared Functions"**

- **"Constants With the High−Order Bit Set"**
- **"Arithmetic with long Types"** (including shifts involving mixed types and code that may overflow 32 bits)

sizeof(int) == sizeof(void *)

An assumption may arise from casting pointers to *int*s to do arithmetic, from unions that implicitly identify *int*s and *pointer*s, or from passing *pointer*s as actual arguments to functions where the corresponding formal arguments are declared as *int*s. Any of these practices may result in inadvertently truncating the high−order part of an address.

The compilers generally detect the first case and provide warnings. Also given ANSI C function prototypes, the compilers generally detect the last case. No diagnostic messages are provided for unions that implicitly identify *int*s and *pointer*s.

You can declare an integer variable that is required to be the size of a pointer with the type *ptrdiff_t* in the standard header file *stddef.h*, or with the types *__psint_t* and *__psunsigned_t* in the header file *inttypes.h*.

Also note that a cast of an *int* to a *pointer* may result in sign−extension, if the sign bit of the *int* is set when a *-64* compilation occurs.

sizeof(int) == sizeof(long)

Data that fits in an *int* or *long* on 32−bit systems will fit in an *int* on 64−bit systems. Expansion, in this case, has no visible effect. Problems may occur, however, where an unsigned *int* actual parameter is passed to a *long* (signed or unsigned) formal parameter without benefit of an ANSI prototype. In this case, the unsigned value is implicitly sign−extended in the register, and therefore is misinterpreted in the callee if the *sign* bit was set.

sizeof(long) == 4

A problem may occur in cases where long *int*s are used to map fields in data structures defined externally to be 32 bits, or where *union*s attempt to identify a *long* with four *char*s.

sizeof(void *) == 4

Problems with this code are similar to those encountered with `sizeof(long)==4`. However, mappings to external data structures are seldom a problem, since the external definition also assumes 64−bit pointers.

Implicitly Declared Functions

It is always risky to call a function without an explicit declaration in scope. Furthermore, be sure to declare with a compatible prototype any function defined with a prototype. Problems arise when mixing prototype and nonprototype declarations for the same function. For example, suppose you call a function (defined with a prototype to take a variable number of arguments) in a scope without a prototyped declaration. You may get unexpected results if a floating point argument is passed to it. This is a typical problem with calls to *printf* and after *stdio.h* routines. Therefore, always include *stdio.h* in any context where you use *stdio.h* facilities.

Constants With the High−Order Bit Set

A change in type sizes may yield some problems related to constants. Be careful about using constants with the high−order (*sign*) bit set. For instance, the hex constant *0xffffffff* yields different results in the expression:

```
long x;
... ( (long) ( x + 0xffffffff ) ) ...
```

In both modes, the constant is interpreted as a 32−bit unsigned *int*, with value 4,294,967,295. In 32−bit mode, the addition results in a 32−bit unsigned *long*, which is cast to type *long* and has value x−1 because of the truncation to 32 bits. In 64−bit mode, the addition results in a 64−bit *long* with value x+4,294,967,295, and the cast is redundant.

Arithmetic with *long* Types

Code that does arithmetic (including shifting), and code that may overflow 32 bits and assumes particular treatment of the overflow (for example, truncation), can exhibit different behavior, depending on the mix of types involved (including signedness).

Similarly, implicit casting in expressions that mix *int* and *long* values may produce unexpected results due to *sign/zero* extension. An *int* constant is sign− or zero−extended when it occurs in an expression with *long* values.

Solving Porting Problems

Once you identify porting problems, solve them by

- changing the relevant declaration to one that has the desired characteristics in both target environments
- adding explicit type casts to force the correct conversions
- using function prototypes or using type suffixes (such as *l* or *u*) on constants to force the correct type

Guidelines for Writing Code for 64−Bit Silicon Graphics Platforms

The key to revising existing code and writing new code that is compatible with all of the major C data models is to avoid the assumptions described above in **"Coding Assumptions to Avoid."** Since all of the assumptions described sometimes represent legitimate attributes of data objects, you need to tailor declarations to the target machines' data models.

The following guidelines help you to produce portable code. Use these guidelines when you are developing new code or as you identify portability problems in existing code.

1. In a header file that you include in each of the program's source files, define (*typedef*) a type for each of the following functions:
   - For each specific integer data size required, that is, where exactly the same number of bits is required on each target, define a signed and unsigned type. For example:

     ```
     typedef signed char int8_t
     typedef unsigned char uint8_t
     ...
     typedef unsigned long long uint64_t
     ```

   - If you require a large scaling integer type, that is, one that is as large as possible while remaining efficiently supported by the target, define another pair of types. For example:

     ```
     typedef signed long intscaled_t
     typedef unsigned long uintscaled_t
     ```

   - If you require integer types of at least a particular size, but chosen for maximally efficient implementation on the target, define another set of types, similar to the first but defined as larger standard types where appropriate for efficiency. The typedefs referred to above exist in the file *inttypes.h* (see **"Using Typedefs"**).

   After you construct the above header file, use the new *typedef* types instead of the standard C type names. You may need a distinct copy of this header file (or conditional code) for each target platform supported.

   If you provide libraries or interfaces to be used by others, be careful to use these types (or similar application−specific types) chosen to match the specific requirements of the interface. Also, carefully choose the actual names used to avoid namespace conflicts with other libraries. Thus, your clients should be able to use a single set of header files on all targets. However, you will always need to provide distinct libraries (binaries) for the 32−bit compatibility mode and the 64−bit native mode on 64−bit Silicon Graphics platforms, although the sources can be identical.
2. Be sure to specify constants with an appropriate type specifier so that they will have the size required by the context with the values expected. Bit masks can be particularly troublesome in this

regard: avoid using constants for negative values. For example, 0xffffffff may be equivalent to a −1 on 32−bit systems, but may be interpreted as 4,294,967,295 (signed or unsigned, depending on the mode and context) on most 64−bit systems. The *inttypes.h* header file provides *cpp* macros to facilitate this conversion. Defining constants that are sensitive to type sizes in a central header file may help in modifying them when a new port is done.

3. Where *printf/scanf* are used for objects whose types are *typedef*ed differently among the targets you must support, you may need to define constant format strings for each of the types defined in step 1. For example:

```
#define _fmt32 "%d"
#define _fmt32u "%u"
#define _fmt64 "%lld"
#define _fmt64u "%llu"
```

The *inttypes.h* header file also defines *printf/scanf* format extensions to standardize these practices. They are implemented by the first release of the 64−bit compilers and related tools.