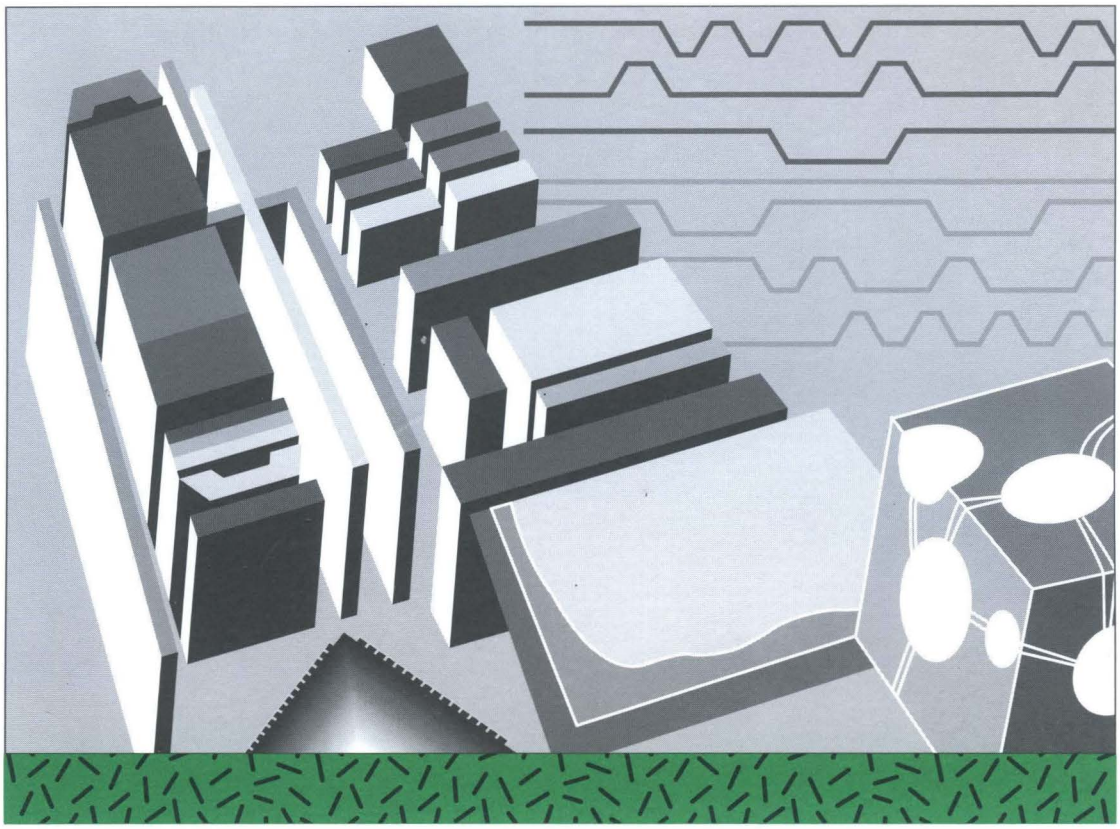
V_R Series™

Programmer's Guide



MIPS RISCompiler and C Compiler

November 1995

NEC

MIPS RISCompiler and C Programmer's Guide

November 1995
Document No. 50777

About This Book

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

Who Should Read This Book?

This book is intended for:

- C programmers
- Programmers using other MIPS high-level languages, supplementing the information in the programmer's guides for these languages.

What Does This Book Cover?

Although the programming environment includes all standard UNIX driver commands and system tools, this book does not describe those tools in detail. For details, you may need to refer to the *User's Reference Manual* and other associated publications.

This book contains implementation details on the supported languages. It does not contain detailed reference information giving the syntax and definition of each language.

For C programmers, this book provides information on compiling and linking programs, storage mapping, language interfaces, and other information specific to the MIPS C implementation.

This book also provides infomation about improving program performance and debugging programs. This information may be useful to programmers using any of *MIPS RISCompilers* (Pascal, or Fortran).

This book has the following chapters:

- **Chapter 1: The Compiler System.** Gives an overview of components of the compiler system.

- **Chapter 2: Linker and Object Tools.** Describes the linker and object tools of the compiler. It also provides reference and guide information in using the various options provided by the compiler drivers.
- **Chapter 3: Storage Mapping.** Describes storage mapping for variables in C.
- **Chapter 4: Language Interfaces.** Provides reference and guide information in writing programs in C that can communicate with Pascal or Fortran programs.
- **Chapter 5: Improving Program Performance.** Describes the profiling and optimization facilities available to increase the efficiency of your programs, and how to use them.
- **Chapter 6: Debugging Your Code.** Shows how to use the source level debugger features.
- **Chapter 7: MIPS–C Implementation.** Describes extensions and modifications supported by the C compiler that differ from other C implementations.
- **Chapter 8: ANSI C Implementation.** Describes features that are new or different from MIPS–C.
- **Appendix A: Byte Ordering.** Describes how the big endian and little endian byte ordering affect the mapping of data in storage.
- **Index.** Contains index entries for this publication.

Summary of Changes By Edition

July 1991 Edition

The following summarizes the changes made to the February 1991 edition of this manual:

- **Chapter 1.** The Link Editor, Archiver, and Object Tools information was removed from this chapter. Information on Language default options was added. Information on Dynamic Shared Objects was added. This chapter was also reorganized.
- **Chapter 2.** This is a new chapter. It describes the Linker and Object Tools. It also explains how to make and use Dynamic Shared Objects.
- **Chapters 3 - 8.** These chapters have been renumbered to reflect the addition of chapter 2.

- **General.** Numerous minor technical and editoral corrections have been made throughout this manual.

February 1991 Edition

The following summarizes the changes made to the December 1989 edition of this manual:

- **Chapter 6.** The C language information formerly in Appendix A is now in Chpater 6.
- **Chapter 7.** This is a new chapter that describes ASNI C features and extensions.
- **Appendix B.** The big and little endian information is now in Appendix A.

December 1989 Edition

The following summarizes the changes made to the December 1988 edition of this manual:

- **Name Change.** The name of this manual was changed from *RISCompiler Languages Programmre's Guide* to *RISCompiler and C Programmer's Guide.*
- All **Pascal** discussion has been moved to the new *MIPS Pascal Programmer's Guide.* **Chapter 3** of this manual has a discussion of the C/Pascal interface.
- **Appendix A.** A description of the **stdarg.h** macros and the **alloca.h** header file have been added.
- **General.** Numerous minor technical and editoral corrections have been made throughout this manual.

December 1988 Edition

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

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

- **New Link Editor Options**: The **-jmopt**, and **-nojmpopt** link editor options are described in Table 1.1 in **Chapter 1**. The *Filling Jump Delay Slots* section in **Chapter 4** describes when to use these options.
- **Pascal**: the text in **Chapter 2** (pg. 2–9) concerning the mapping of Pascal objects has been greatly expanded with additional rules and examples. Additional information has also been provided in Chapter 3 (p. 3–2) on the interface between programs written in Pascal and those written in C.
- **Index**. Approximately 200 entries have been added to the Index, enhancing the ability to retrieve information from this manual more efficiently.
- **General**. Numerous minor technical and editorial corrections have been made throughout the manual.

For More Information

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

- MIPS Assembly Language Programmer's Guide (ASM-01-DOC)
- MIPS RISC/os Programmer's Reference Manual (ROS-01-DOC)
- MIPS RISC/os User's Reference Manual (ROS-02-DOC)
- MIPS Pascal Programmer's Guide (PAS-01-DOC)
- *The C Programming Language* by Kernighan and Ritchie (Prentice Hall, 1978).

Contents

About This Book

1 The Compiler System

2
Linker and Object Tools

3
Storage Mapping

4
Language Interfaces

5
Improving Program Performance

6
Debugging Programs

7
MIPS C Implementation

8
ANSI C Implementation

A
Byte Ordering

The Compiler System

1

This chapter provides an overview of the compiler system, the languages supported and the tools used to create programs.

In addition to the compilers (e.g. C, Pascal) there are text editors for writing and editing programs, a debugger, a profiler, utilities to examine object files, and an archiver. The compiler tools and their functions are summarized in Table 1.1.

Table 1.1: Compiler System and Functions

Task	Tool
Write and Edit programs	vi, emacs
Compile, Link and Load Programs	cc, ld
Debug Programs	dbx
Profile Programs	pixie, prof
Optimize Programs	pixie, prof, cache
Examine Object File(s)	nm, file, size and odump
Build Libraries	ar

Operational Overview

Figure 1.1 shows the relationship between the major components of the compiler system and their primary inputs and outputs.

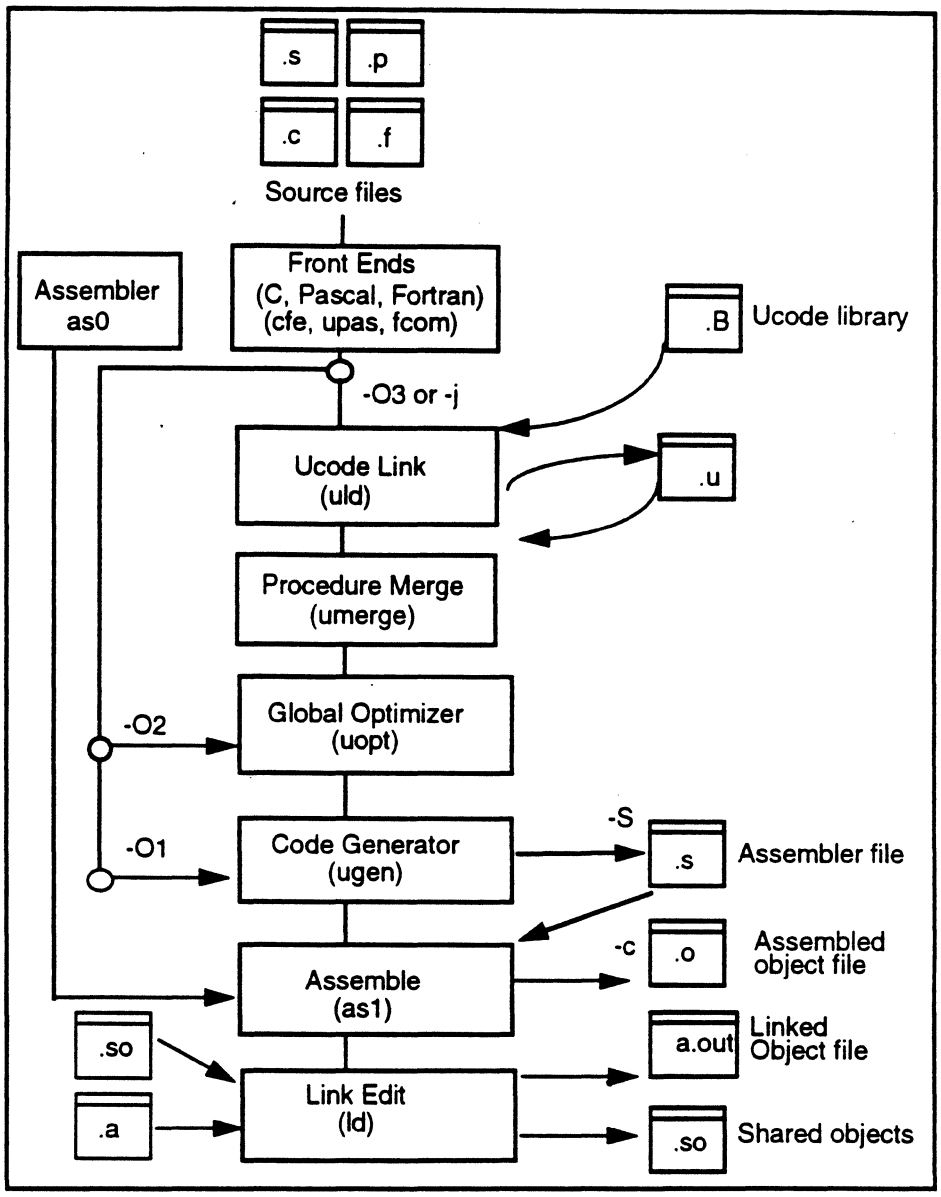


Figure 1.1: The Compiler System Driver

Note: FORTRAN uses additional preprocessors (see Figure 1.2). For more information, see the *efl(1)*, *ratfor(1)*, and *m4*(1) manual pages in the *RISC/os User's Reference Manual*.

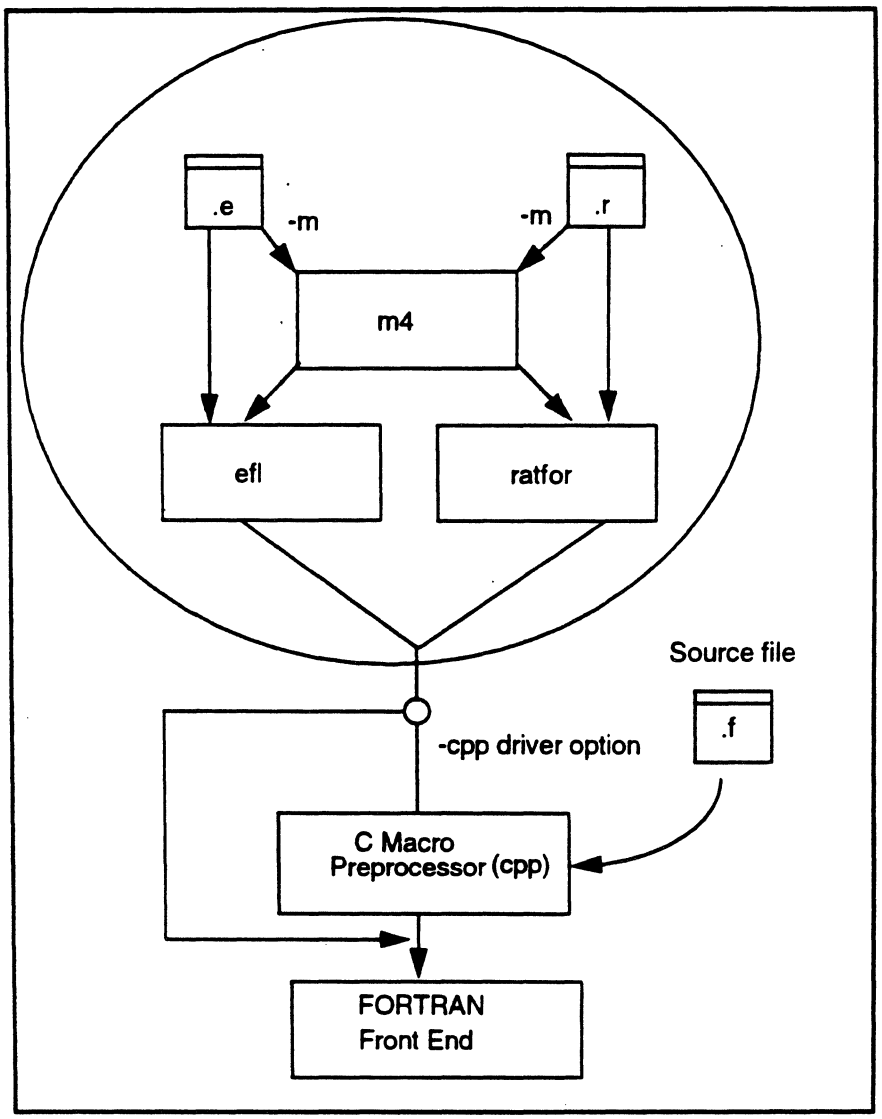


Figure 1.2: The FORTRAN Preprocessors

Driver

Each language has its own driver. These driver programs invoke the components of the compiler system to compile a program: the macro preprocessor (*cpp*), the compilers (C, FORTRAN 77, or Pascal), the assembler, and the link editor.

Languages Supported

The compiler system supports four languages. Please note that the operands for each of the languages, except MIPS Assembly, are the same: [compiler options], [link options] and [source name list]. MIPS Assembly does not use [link options]. Table 1.2 lists the supported languages and their drivers.

Table 1.2: Compiler Drivers

Language	Driver Name
C	cc
FORTRAN 77	f77
MIPS Assembly	as
Pascal	pc

Note: The languages supported by any one system are determined at the time of purchase. The configuration of your particular system may not support all of the languages. Each language requires different libraries at link time. The driver program for a language passes the appropriate libraries to the link editor.

Files

The driver recognizes the contents of an input file by the suffix assigned to the filename, as shown in Table 1.3.

Table 1.3: Driver Recognized File Suffixes

File Suffixes	
Suffix	**Description**
.a	Static (non-shared) object library.
.B	Ucode object library.
.c	C source code.
.e	Elf source.
.f	Fortran 77 source.
.i	Assumes the source code was already processed by the C preprocessor and is in the language expected by the driver. For example, pc -c source.i assumed source.i contains Pascal source statements.
.o	Object file.
.p	Pascal source code.
.r	Ratfor source code.
.s	Assembly source code.
.so	Dynamic shared object library.
.u	Ucode object file.

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

Default Options

The driver predefines the following macros for each language. They are:

C (std0 mode):

-DLANGUAGE_C	-D_LANGUAGE_C
-Dunix	-D__unix
-Dhost_mips	-D__host_mips
-DCFE	-D_CFE
-DSYSTYPE_SVR3	-D_SYSTYPE_SVR3
-DMIPSEB	-D_MIPSEB
-Dmips=1	-D__mips=1

For machines using R6000 architecture, _Dmips=2 is predefined instead of _Dmips=1.

For machines using R4000 architecture, _Dmips=3 is predefined instead of _Dmips=1.

C (std1/std mode (alternative))

-D_LANGUAGE_C
-D__unix
-D__mips=1
-D__host_mips
-D_CFE
-D_SYSTYPE_SVR3
-D_MIPSEB

Assembly

-DLANGUAGE_ASSEMBLY	-D_LANGUAGE_ASSEMBLY
-Dunix	-D__unix
-Dmips=1	-D__mips=1
-Dhost_mips	-D__host_mips
-SYSTYPE_SVR3	-D_SYSTYPE-SVR3
-DMIPSEB	-D_MIPSEB
-D__DSO__	

FORTRAN (only with -cpp)

-DLANGUAGE_FORTRAN	-D_LANGUAGE_FORTRAN
-Dunix	-D__unix
-Dmips=1	-D__mips=1
-Dhost_mips	-D__host_mips
-DSYSTYPE_SVR3	-D_SYSTYPE-SVR3
-DMIPSEB	-D_MIPSEB
-D__DSO__	

Pascal

-DLANGUAGE_PASCAL	-D_LANGUAGE_PASCAL
-Dunix	-D__unix
-Dmips=1	-D__mips=1
-Dhost_mips	-D__host_mips
-DSYSTYPE_SVR3	-D_SYSTYPE-SVR3
-DMIPSEB	-D_MIPSEB
-D__DSO__	

Compiling Multi-Language Programs

When the source language of the main program differs from that of a subprogram, compile each program separately with the appropriate driver and link them in a separate step. It is possible to create objects suitable for link editing by specifying the –c option, which stops the driver immediately after the assembler phase.

For example:

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

produces the results shown in Figure 1.3.

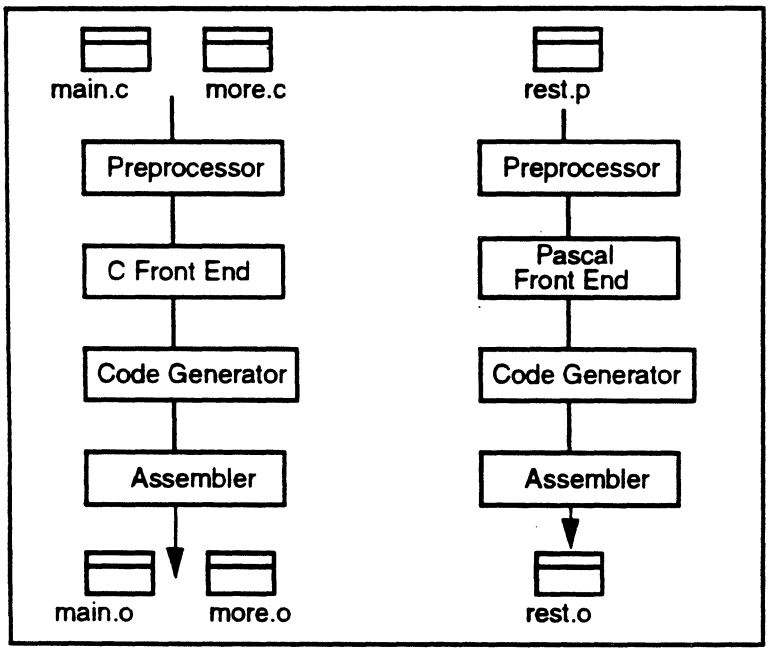


Figure 1.3: Compiler Control Flow with –c Option

Linking Objects

A driver command is used to link edit separate objects into one executable program. When the -c option is not used, the driver compiles and link edits the specified modules. If the modules are all object files, they are link-edited into one executable program. It is possible to link edit the objects created in the last example using the Pascal driver *pc*, as shown below:

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

This command produces the executable object *all*. The example below achieves the same result using the C driver *cc*:

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

The *cc* driver links with *libc* and *libdw* by default. It is your responsibility to link code with any additional libraries. In the above example, -lp specifies the Pascal runtime library.

The Pascal and FORTRAN drivers *pc* and *f77* automatically link with the necessary libraries, including *libc*.

Figure 1.4 shows the flow of control for both the *pc* and *cc* commands shown above.

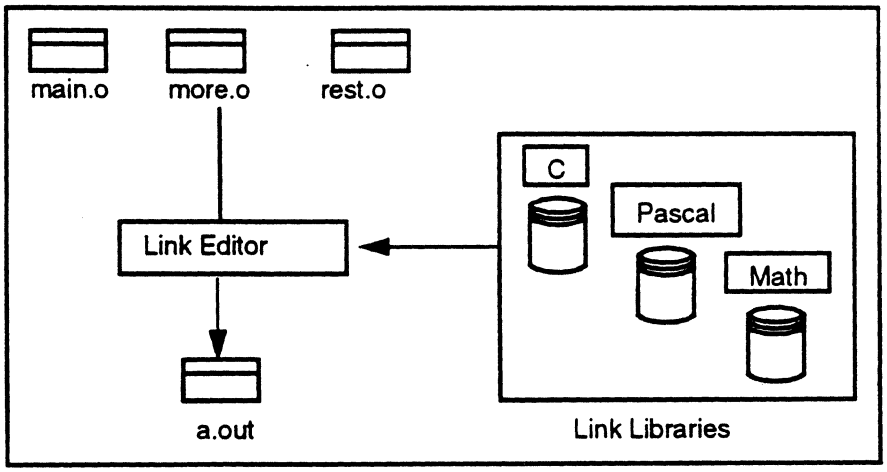


Figure 1.4: Compiler Control Flow of cc and pc

The link editor is described in more detail in Chapter 2. For a detailed list of the default libraries used by each driver, see the *cc(1)*, *f77(1)*, or *pc(1)* manual pages in the *RISC/os User's Reference Manual*.

Compiler Options

There are several different types of compiler options. These include:

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

Some options have defaults which are used when you do not specify an option on the command line. The tables below summarize the different types of options, and indicate which of the options are default options.

Table 1.4 summarizes the general compiler options.

Note: The table lists only the most frequently used options; it does not list all available options. See the *cc(1)*, *f77(1)*, or *pc(1)* manual page in the *RISC/os User's Reference Manual* for a complete list of available options.

Table 1.4: Compiler Options, 1 of 4

General Compiler Options	
Option Name	**Purpose**
-B *string*	Append *string* to all names specified by the -t option.
-C	C and Assembly drivers only. Used with the -P and -E options. Prevents the macro preprocessor from stripping comments. Use this option when you suspect the preprocessor is not emitting the intended code to examine the code with its comments.
-C	Pascal and FORTRAN drivers only. Generates code that causes range checking for arrays during program execution.
-c	Prevents the link editor from linking the program after compilation. This option forces the compiler to produce a *.o* file.
-call_shared	Produce dynamic executable that uses sharable objects during run-time (default).
-check_bounds	For C drivers only. Generates code that causes range checking for arrays during program execution.
-cord	Rearrange the procedures in the link edit object file to reduce cache conflicts in the executable object (*a.out*). At least one -feedback file must be specified. See Chapter 5 for more information.
-cpp	Run the C macro preprocessor on the source code before compiling. The default varies from driver to driver. Refer to the appropriate man page in *RISC/os User's Reference Manual* for the individual driver.
-crt0	Use *crt0.o* as the compiler startup routine in BSD-like environments.
-crt1	Use *crt1.o* and *crtn.o* as compiler startup and finish routines in Sys V-like environments (default).
-D*name* or -D*name=def*	Define a macro name if a *#define* is specified in the program. If *=def* is omitted, the compiler defines the name to be 1.
-E	Run only the C macro preprocessor and send the results to the standard output. Specify -C to retain comments for C and Assembly code. Use -E when you suspect the preprocessor isn't emitting the intended code.
-edit [0-9]	Invoke the eitor of choice when syntax or symantic errors are detected by the compiler's frontend.
.-feedback *file*	When used with the -cord option produces an object with the procedures rearranged to reduce cache conflicts. *file* is the output produced when using the -prof and -feedback options.
-float	Cause the compiler not to promote expressions of type *float* to type *double*.

Table 1.4: Compiler Options, 2 of 4

General Compiler Options	
Option Name	**Purpose**
-framepointer	Assert the requirement of frame pointer for all procedures defined in the source code
-G num	*num* is a decimal number that specifies the maximum size in bytes of an item to be placed in the global pointer area. The default is 8 bytes. Change *num* to control the number of data items placed in these sections. See Chapter 5 for more information.
-h *path*	Use *path* rather than the directory where the *name* is normally found.
-L	When specified in addition to *-L dirname*, the compiler searches the default directory.
-Ldirname	Compiler searches the current directory, *dirname*, and the default directory, */usr/include*, in this order, for the include file.
-j	Similar to -c. Produces a *.u* file containing ucode. Does not produce a *.o* file, unless used with -c.
-k *option*	*option* is one of the link editor options. The driver passes it to the ucode loader, which then performs the link action specified by *option*.
-ko *filename*	*filename* is the name of the output file to be created by the ucode loader.
-M	Cause cpp to print, one per line on standard output, the path names of included files.
-mips1	Generates mips1 instructions (R2000/R3000 architecture) and object file. This is the default for all machines.
-mips2	Generate mips2 instruction (R6000 architecture) and object file. The resultant binary will not be executable on a mips1 machine.
-mips3	Generate code using the instruction set of the R4000 RISC Architecture.
-noinline	Disable the inlining performed under the -O3 option.
-nocpp	Do not run the C macro preprocessor on C and Assembly source files before processing.
-non_shared	Produce an executable that does not use shared objects.
-O limit	Specify the maximum size, in basic blocks, of a routine that will be optimized by the global optimizer.
-o *filename*	Assigns the name *filename* to the program object. When used with the -c option, tells where to leave the *.o* file. The default filename is *a.out*.

Table 1.4: Compiler Options, 3 of 4

General Compiler Options	
Option Name	**Purpose**
-oldc	Use the old MIPS-C preprocessor (cpp) and C front end (ccom). Use this option if the new preprocessor and front end (cfe), the defaults, fail to compile or correctly execute code when compiled with -std0.
-oldcomment	In the preprocessor, delete comments (replace with nothing), rather than replace comments with a space. This allows traditional token concatenation. This is the default in -std0 mode.
-P	Similar to -E option, placing the results in a *.i* file. Specify both -P and -C to retain comments.
-p0	Do not permit any profiling (default).
-p1 or -p	Permit program counter (pc) sampling. This provides operational statistics to use in improving program performance. This option affects only the link editor. It is ignored by the compiler front ends.
-proto [is]	Invoke the prototizer. This assists in the creation of function prototypes and is useful in converting non-ASNI C programs to ANSI C. This takes one or more source files as input and creates a *.H* file for each. The *.H* file contains function prototypes for all functions in the file. No *.H* file is created if the file has compilation errors or if there are conflicting declarations.
-Q	Cause cpp to use ' (single quotes) for the string literal in the __FILE__ expansion (default it to use " (double quotes)).
-S	Similar to -c, producing Assembly code in a *.s* file instead of object code in a *.o* file.
-signed	Cause all *char* declarations to be *signed char* declarations. Default is *unsigned char*.
-std	Cause cpp to define _STDC_ with the value 0, and enforce the ANSI C standard with popular extensions. Issues a warning message when the compiler finds a non-standard feature in the programming language of the source program.
-std0	Indicates that the programming language is MIPS-C (K & R with extensions); the macro _STDC_ is undefined. This is the default. See Chapter 7 for details on MIPS-C features and extensions.
-std1	Indicates the programming language is strict ANSI C and causes the macro _STDC_=1 to be asserted by the preprocessor. Any non-standard features used cause error messages. See Chapter 8 for details on ANSI C.

Table 1.4: Compiler Options, 4 of 4

General Compiler Options	
Option Name	**Purpose**
-systype *name*	Use the specified compilation environment *name*. Supported environments are *bsd4*, *svr3* (default) and *svr4*. This has the effect of changing the directory searched for #include files and runtime libraries. */name* is added to the beginning of the usual search path.
-trapuv	Forces all uninitialized stack, automatic and dynamically allocated variables to be initialized with 0xFFFA5A5A. When used as a floating-point variable, it is treated as a floating-point NaN and causes a floating-point trap. Do not use as a pointer, because a segmentation violation occurs.
-Uname	Overrides a definition of a macro name specified with the -D option, or one that is defined automatically by the driver.
-unsigned	Cause all *char* declarations to be *unsigned char* declarations.
-V	Print the version number of the driver and its phases. Use the version number when reporting a problem.
-v	Lists compiler phases as they are executed. For BSD 4.3 users, this also prints resource usage of each phase.
-varargs	Print warnings for lines that may require the *varargs.h* macros.
-verbose	This option causes output of the long form of error and warning messages. These may give the user some hint as to the reason the compilation failed.
-volatile	Cause all variables to be treated as volatile.
-w or -w1	Suppress warning messages.
-w2	Abort on warning message as if an error occurred.
-w3	Suppress warning messages, but exit with non-zero exit status when warnings occur.
-Zpn	Align structure members on alignment specified by the integer *n*.

Note: There are certain restrictions in mixing compiler options. These include:

- The -oldc flag cannot be used with *std1*.
- The -oldc flag cannot be used with *std*.

Byte Ordering Options

The compiler can produce program objects which are executable on target machines with either a big–endian or little–endian byte ordering scheme. By default, the compiler produces program objects executable on target machines with the same byte ordering scheme as the compilation machine.

Specify one of the options shown Table 1.5 when the byte ordering scheme on the compilation machine differs from that on the target machine.

Table 1.5: Byte Ordering Compiler Options

Byte Ordering Options	
Option Name	**Purpose**
-EB	Produces an object file for a target machine that uses a big-endian scheme. Use this option when compiling on a little-endian machine.
-EL	Produces an object file for a target machine that uses little-endian scheme. Use this when compiling on a big-endian machine.

See Appendix A for more information on big–endian and little–endian byte ordering.

Debugging Options

Table 1.6 shows the compiler options available for debugging source code using *dbx*. Chapter 6 describes the functions and operations.

Table 1.6: Debugging Options

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

Profiling Option

The *pixie* and *prof* programs (see Chapter 5) allow you to profile programs. The –p option to the driver causes the program to be linked with a module that produces a file *mon.out* when the program is executed. *mon.out* contains program–counter sampling information.

Optimizer Options

Table 1.7 summarizes the options available for program optimization. Refer to Optimization section in Chapter 5 for a detailed explaination of optimizing code. See also the *cc(1)*, *f77(1)*, or *pc(1)* manual page, as applicable, in the *RISC/os User's Reference Manual* for details on the –O3 option, and the input and output files related to this option.

Table 1.7: Optimizer Options

Optimizer Options	
Option Name	**Purpose**
-O or O2	Global optimization. Optimizes within the bounds of individual compilation units. This option executes global optimizer (uopt) phase.
-O0	No optimization. Prevents all optimizations, including the minimal ones normally performed by the code generator and the assembler.
-O1	The assember and the code generator perform as many optimizations as possible without affecting performance. This is the default.
-O3	Performs global register allocation across the bounds of individual compilation units. Executes the uld, umerge and uopt phases of the compiler system. This option cannot be used with the -c compiler option. No shared objects will be produced with this option.

Note: When the optimization level is -O2 or less, the link editor defaults to building an executable which uses shared objects. You cannot mix *-O3* optimization with [*-call_shared*].

Compiler Development Options

In addition to the standard options, each driver also has options which primarily aid compiler development work. Table 1.8 shows the compiler options available for development work. For complete information about these options see the *cc(1)*, *pc(1)*, or *f77*(1) man page, as appropriate, in the *RISC/os User's Reference Manual*

Table 1.8: Compiler Development Options

Option Name	Purpose
-H*c*	Halt compiling after the pass specified by the character *c*, producing an intermediate file for the next pass. It selects the compiler pass in the same way as the -t option. If this option is used, the symbol table file produced and used by the passes is the last component of the source file with the suffix changed to *.T* and is not removed.
-K	Build and use intermediate file names with the last component of the source file's name. These intermediate files are never removed even when a pass encounters a fatal error. When ucode linking is performed and the -K option is specified, the base name of the files created is *u.out* by default.
-t	Select the names from the list below. The names selected are those designated by the characters following the -t option according to those listed below. The arguments are processed from left to right so their order is significant. The -B option is always required when using -t.
	Character — **Name** h — include pf — cfe p (with -oldc) — cpp f — ccom (with -oldc), efe, fcom, upas d — ddopt q — uopt0 j — ujoin u — uld s — usplit m — umerge o — uopt c — ugen a — as0 b — as1 l — ld y — ftoc z — cord r — [m]crt [1n].o n — libprof1.a t — btou, utob
-Wc [c...], arg1[,arg2...]	Pass the argument[s] *argi* to the compiler pass/passes: *c [c . . .]*. The c's are one of [pfjusmocablyz]. The c's select the compiler pass in the same way as the -t option.

Including Common Files (Definition Files)

When writing programs, there are often header (or include) files that are shared among a program's modules. These files define constants, the parameters for system calls, procedure prototypes, etc.

Header files have a *.h* suffix. Typically, the manual page for a library routine or system call from the *RISC/os Programmer's Reference Manual* indicates the required include files. Header files can be used in programs written in different languages; header files are handled by the preprocessor.

Note: If you intend to debug your program using *dbx* (see Chapter 6), do not place executable code in an include file. The debugger interprets an include file as one line of source code; none of the source lines in the file appear during the debugging session.

You can include header files in program source files in one of two ways:

- Place the following line in a source file; it must begin in column 1:

  `#include "`*`filename`*`"`

  where *filename* is the name of the include file. The double quotes around the filename indicate that the C macro preprocessor is to search in sequence the current directory and the default directory, */usr/include*.

- Place the following line in a source file; it must begin in column 1:

  `#include <`*`filename`*`>`

  where *filename* is the name of the include file. The greater–than and less–than signs around the filename indicate that the C macro preprocessor is to skip the current directory and search only the default directory */usr/include* for the include file.

The –systype *name* compiler option can be used to change the compilation environment. Currently supported values for *name* are *bsd43*, *svr3* and *svr4*. The -systype option has the effect of changing the default directories that are searched for include files and libraries. If no systype is provided, the compilers driver defaults to *systype svr3*.

C, Pascal, FORTRAN 77, and assembly code can reside in the same include files, and then can be conditionally included in programs as required. To set up a sharable include file, you must create a *.h* file and enter the respective code as shown in Figure 1.5.

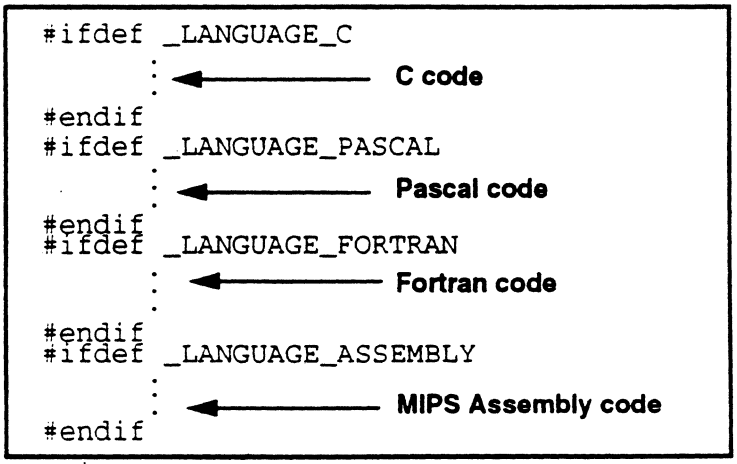


Figure 1.5: Sharable Include File

Dynamic Shared Objects

MIPS RISCompiler supports dynamic shared objects (*dso*). Dynamic shared objects save disk storage. They have few restrictions on memory placement.

Use the link editor *(ld)* to build dynamic shared objects.

Use the runtime linker *(rld)* to link dynamic shared objects.

Refer to Chapter 2 for more information on building and using dynamic shared objects.

Linker and Object Tools

2

This chapter describes the linker and object tools of the compiler system. These tools include:

- Link Editor (*ld*)
- Runtime linker (*rld*)
- Object file tools (*odump, nm, file, size, dump* and *string*)
- Archiver (*ar*)

Link Editor

The link editor (*ld*) and the runtime linker (*rld*) both perform symbol resolution by linking the symbol definition with the calling of that symbol in a different part of a program. Each module of a program is searched for definitions of undefined symbols.

One of the differences between the two linkers is when this symbol resolution occurs. The link editor (*ld*) performs symbol resolution when the executable is created (*static linking*), while the runtime linker (*rld*) performs symbol resolution during program execution (*dynamic linking*).

For more information on *rld*, see the section entitled Runtime Linker.

The link editor (*ld*) performs *static linking* by combining one or more object files (created by the assembler), and, or archives into one program object file. This includes relocation, external symbol resolution, and any processing necessary to create an executable object file.

The link editor is capable of creating either shared (*dynamic*) or non-shared (*static*) object files.

Dynamic vs. Static Object Files

Dynamic shared object files are:

- Shared by several users, and, or programs.
- Relocatable objects which contain Position Independent Code (PIC) and Global Offset Tables (GOT) for indirect references.
- Objects which have runtime data structures that allow the runtime linker (*rld*) to relocate the dynamic executable during execution.

Static or non-shared objects are normal executable object files.

Building Dynamic Shared Objects

Run the link editor by entering *ld* on the command line of the shell or by using one of the driver commands as described in Chapter 1, Linking Objects.

The syntax of the *ld* command is as follows:

```
ld -option[s] object1 [object2...objectn]
```

The following command shows how to build the shared object *libc.so* from an archive *libc.a*:

```
ld -shared -o libc.so -all libc.a -set_version sysv_4.0
```

where:

-shared	Makes a shared object.
libc.so	All shared objects have *.so* suffix.
-all	Link all objects from archives following this option.
-set_version	Specifies an interface version (e.g. sysv_4.0). See Table 2.1 for a complete description of -set_version.

Reference to *so_locations*

When a shared object is created, *ld* looks in *so_locations* for non-conflicting memory addresses for the text and data portions of the object. *so_locations* is a file in */usr/lib* which contains the default addresses assigned to shared objects. It also keeps track of addresses assigned to newly created shared objects.

To avoid possible conflicts with MIPS supplied shared objects, the user should place any newly created shared objects below address 0x60000000. All third party shared libraries should be built with data placed right after text.

Dependencies

When building a shared object, any other shared objects upon which the first depends must be specified. If, for example, *shared object A* uses a global symbol which is defined in *shared object B*, then *A* is dependent upon *B*.

The following command show how to build *libcurses.so* (which has dependencies) from the archive *libcurses.a*:

```
ld -shared -transitive_link -o libcurses.so -all libcurses.a
     -no_archive -lc -set_version sysv_4.0
```

where

-shared	Makes a shared object.
-transitive_link	This causes *ld* to search for all dependent *.so* files automatically.
libcurses.so	Example of the output file name. All shared objects have *.so* suffix.
-all	Link all objects from archives following this option.
-no_archive	Do let any -l option argument use archive (*.a*) files. The default is to use *.a* files only if *.so* files are not found.
-set_version	Specifies an interface version (e.g. sysv_4.0).

Building Static Objects

Run the link editor by entering *ld* on the command line of the shell or by using one of the driver commands as described in Chapter 1's section entitled Linking Objects.

The syntax of the *ld* command is as follows:

```
ld -option[s] object1 [object2...objectn]
```

Note: The assembler driver *as* does not run the link editor. To link edit a program written in assembly language:

- Assemble and link edit using one of the other driver commands (*cc*, for example). The *.s* suffix of the assembly language source file causes the driver to invoke the assembler.

or

- Assemble the file using *as*, then link edit the resulting object file with the *ld* command.

Unless otherwise specified, the link editor names the program object file *a.out*. You can execute the object file or use it as input for another link editor command.

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

Using Dynamic Shared Objects

Why Use Dynamic Objects?

Reasons to use dynamic shared objects include:

- Shared objects can be relocated without having to recompile applications.
- Shared objects reduce the dynamic memory needs of the system.
- Executables using shared objects require less disk space.
- Shared objects can be updated without having to relink the applications which depend upon them.

In short, use dynamic shared objects because they save disk storage.

Requirement

Assembler code must abide by the System V Application Binary Interface (ABI) calling conventions. The loader depends upon it. The link editor traps some of the non-conforming usages by printing error messages.

Calling Conventions

- Calculations of a new value for the *gp (global pointer)* register must occur in the first three instructions of a function which allocates a stack frame.
- The stack pointer must allocate the stack frame prior to any other use of the stack pointer register.
- Adjusting the stack pointer value to deallocate the stack frame must occur only once and it must occur within the last basic block of the function.
- Only one frame pointer may be used in a function which allocates a stack frame.
- Only one exit from a stack adjustment function is allowed. This must be done using the jump register instruction transferring control to return address register $31.
- Branching to a different procedure is not allowed.

Recommendations

To get optimal results when using shared objects:

- All symbols must be defined in some archive or user code. The runtime linker (*rld*) has to resolve all undefined data symbols and the "referenced text symbols" during runtime. This resolution of undefined data symbols slows up the linking or causes a user program to abort.
- Static uninitialized structures and arrays should be demand *malloced* to reduce swap requirements. If they are not *malloced*, RISC/os allocates swap space for these items whether or not they are used. Swap requirements should be reduced.

Using Static Objects

Why Use Static Objects

Although there are advantages to using dynamic objects, it does increase system overhead and record keeping. There are times when it is more appropriate to use static objects. 7zUse *static* objects if a process:

- Calls only a few small libraries, or
- Accesses only limited routines in a library.

Specifying Libraries

There are two kinds of libraries, *shared* and *static*.

A *shared* or dynamically linked library is a single object file which contains the code for every function within the library. It is created by the link editor *(ld)*. This file appears to the system and the user as individual objects within a file system or directory. This shared library has a *.so* suffix.

The compiler looks for shared libraries by default. If one is not found, the compiler looks for archives. The compiler prints a warning message if an archive was found instead of a shared object.

A *static* library or archive, is a collection of object files which each contain the code for functions within the library. It is created by the archiver (*ar*). All of the files in a static library have a *.a* suffix.

Multiple Language Programs

To compile multi-language programs, explicitly load any required runtime libraries. For example, if the main program is in C, and other procedures are in Pascal, explicitly load the Pascal library *libp.a* or *libp.so*

and the math library *libm.so* or *libm.a* with the options *-lp* and *-lm* (abbreviations for the libraries *libp.so* or *libp.a* and *libm.so* or *libm.a*), as shown below, when linking the program.

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

To find the Pascal library, the link editor replaces the *-l* with *lib* and adds a *.so* after *p*. It then searches the */usr/lib/cmplrs/cc/pc* directory for this shared library *libp.so* first. If it cannot find *libp.so*, it searches for the archive library *libp.a*.

For a list of the libraries that a language uses, see the associated driver manual page (*cc*(1), *f77*(1), or *pc*(1)) in the *RISC/os Programmer's Reference Manual*.

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

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

To specify a library created with the archiver, enter the name of the library as follows:

```
% cc main.o more.o rest.o libfft.a (or libfft.so) -lp
```

Note: The link editor searches libraries in the order specified. Therefore, if a library (for example *libfft.so* or *libfft.a)* uses data or procedures from -lp, you must specify *libfft.so (or libfft.a)* first.

Link Editor Options

Table 2.1 summarizes the link editor options. Refer also to the list of general options in Chapter 1 and to the *ld(1)* manual page in the *RISC/os Programmer's Reference Manual* for more information on options and libraries that affect link editor processing.

Table 2.1 Link Editor Options, 1 of 4

Link Editor Options	
Option Name	**Purpose**
-A *file*	Produces an object that may be read into an existing program. The argument, *file*, is the name of the file whose symbol table is used to base the definition of new symbols. Only newly linked information is entered into the text and data portions of *a.out*; the new symbol table reflects every symbol defined before and after the incremental load.
-all *archive name*	Link in all of the objects from archive name.
-B *num*	Sets the starting address of the uninitialized data segment (bss) to the hexadecimal address *num*. This option is valid only when the *–N* link editor option is also used.
-B*string*	Appends *string* to the library name created by the *–lx* or *–klx* option. The library is searched both with and without *string*.
-b	Tells *ld* not to merge symbolic information entries from the same file into one entry for that file. Use this option when a file compiled for debugging has variables with the same names but different attributes. This can occur when compiling two object files that use the same include file, and variables with the same name differ because of conditional compilation statements within the file.
-call_shared	Produce shared executables.
-check_registry *file*	Check the location of this shared object's segments and make sure the segments stay out of the way of others in the *so locations_file*. Multiple instances of this option are supported. This option can only be used in conjunction with *-shared*.
-D *num*	Sets the starting address of the data segment (data) to the hexadecimal address *num*.
-EB	Uses big–endian byte ordering when writing out header and symbol table entries.
-EL	Uses little–endian byte ordering when writing out header and symbol table entries.
-e *epsym*	Sets the default entry point address for the output file to the specified symbol *epsym*.
-exact_version *obj*	Sets the LL_EXACT_MATCH flag in *liblist* flags files. This tells *rld* that *obj* must match the timestamp and checksum from the *liblist* section in addition to the interface version.
-exclude_object	Provides an all but facility. Used with *-all*, this implies that when linking all of the objects from the next archive, we skip the specified object is skipped.
-F or -z	Creates a ZMAGIC file (an object file that loads on demand). This is the default.

Table 2.1 Link Editor Options, 2 of 4

Link Editor Options	
Option Name	**Purpose**
-fini symbol_name	Add a call to function *symbol_name* in the *.fini* section.
-G *num*	Specifies the maximum size (in decimal bytes) of a *.comm* item that should be allocated in the small uninitialized data (*sbss*) section for reference by the global pointers. The default is 8 bytes.
-bestGnum	Prints the optimum value to be specified as the *num* value for *–G*. The link editor uses the following options in determining which objects are to be included or excluded in computing a value to be specified in the *–bestGnum* option. For example, exclude any object for which you do not have the source code for recompilation.
-count	Objects that follow on the command line cannot be recompiled.
-nocount	Objects that follow on the command line can be recompiled.
-countall	Overrides any *–nocount* option appearing after it on the command line.
-hidden *objs*	Specifies that *ld* turns all external symbols from any objects following this flag into local variables.
-hidden_symbols *objs*	Specifies that ld turns the symbol following this flag into a local.
-ignore_version *lib*	Specifies that at runtime, the shared object(s) within the library following this option does not have to match the interface version as specified at linktime. Sets LL_IGNORE_VERSION flag in *liblist*. Version are required to match at runtime by default.
-init symbol_name	Add a call to function *symbol_name* in the *.init* section.
-jmpopt or –nojmpopt	Fill or don't fill the delay slots of jump instructions with the target of the jump and adjust the jump offset to jump past that instruction. Disabled when the *–g1*, *–g2* or *–g* flag is present. When enabled, this option can cause an out–of–memory condition in the link editor.
-L	Indicates that */usr/lib/cmplrs/cc* should NOT be searched. Is useful if *dirname* is the only directory that should be searched for libraries.
-L *dirname*	Indicates that *dirname* should be searched for libraries specified in the *–lx* option before searching directory */usr/lib/cmplrs/cc*. This option must precede the *–lx* option.
–l*x*	Specifies the name of a link library, where *x* is the library name. The link editor searches for lib*x*.a in */usr/lib/cmplrs/cc* and */usr/lib*. If a library relies on procedures or data from another library, specify that library's name first. If a library resides in a directory other than */usr/lib/cmplrs/cc*, use the –L option to specify the appropriate directory for that library. **Note:** If the byte–ordering (endian) scheme of the object module differs from that of the machine on which the link editor executes, the default libraries change. See the *ld(1)* manual page in the *RISC/os Programmer's Reference Manual* for more information.

Table 2.1 Link Editor Options, 3 of 4

Link Editor Options	
Option Name	**Purpose**
-M	Produces a link editor memory map in BSD format.
-m	Produces a link editor memory map in System V format.
-N	Creates an OMAGIC* file. The text segment isn't readable and sharable by other users. The data segment follows immediately after the text segment.
-n	Creates an NMAGIC* file. The text segment is read–only and sharable by all users of the file.
-nN	Creates an NMAGIC* file. The data segment immediately follows the text segment.
-no_preempt *objs*	Turns all relocations for specified objects into local relocations. This effectively disallows preempting externals in these objects for this executables or shared object.
-no_preempt_symbol	Turns all relocations for the symbol following this flag into local relocations. This effectively disallows preemption for this executable or shared object.
-no_unresolveds	This causes *ld* to exit with an error status when it encounters any unresolved symbols. The default allows unresolved symbols in shared executables and objects.
-non_hidden *objs*	Turns off the effects of *-hidden*. All external symbols in objects following this flag are left as externals.
-non_shared	Make the output of this link run as non-shared, and use only the archives. The *-r*, *-N*, and *-n* flags all imply non-shared.
-none	Turns off -all.
-o *filename*	Specifies a name for your object file. If you don't specify a name the link editor uses *a.out* as the default.
-p *file*	Preserves the symbol names listed in *file* when loading ucode object files. The symbol names in *file* are separated by blanks, tabs, or new lines. See *Optimizing Frequently Used Modules* in Chapter 4 for an example.
–r	Performs a partial link–edit, retaining relocation entries. This is required if the object is to be re–link edited with other objects in the future. The option causes the link editor not to define common symbols and to suppress messages on unresolved references.
-rpath	Set the *rpath* (see the generic ABI) to the specified string.
–S	Suppresses non–fatal error reporting.
–s	Strips symbol table information from the program object, reducing its size.

Table 2.1 Link Editor Options, 4 of 4

Link Editor Options	
Option Name	**Purpose**
-set_version\ *version_string*	Used in conjunction with *-shared* flag. The specifies the version included in the liblist section. The *version_string* can contain colon separated version strings. When executables are linked against this shared object at linktime, the linker propagates the first version from the shared object's *version_string* to the objlist of the executable. The runtime linker will only map shared objects whose interface version list contains *liblist's* version string.
-shared	The output of the link is a shared object. This includes creating all of the tables for runtime linking, converting the code to PIC and resolving references to other specified shared objects.
-soname \ *shared_object_name*	Set *DT_SONAME* for a shared object. The name may be a single component name (e.g. *libc.a*) a full (starting with a slash), or relative pathname (containing a slash). Single component names use *rpath, LD_LIBRARY_PATH*, and the default paths to resolve their locations.
-T *num*	Sets the origin for the text segment to the specified hexadecimal number. The default origin is 0x400000. The contents and format of the text segment are described in the *MIPS Assembly Language Programmer's Guide.*
-transitive_link	Use this to resolve any unknown or undefined shared object dependencies.
–u *symname*	Makes *symname* undefined so that library components that define *symname* are loaded.
-update_registry *file*	Register the location of this shared object's segments and make sure they stay out of the way of others in *so_locations*. *so_locations* is updated if it is writable. This option can only be used in conjunction with *-shared.*
–V	Prints the link editor version number. Use this number when reporting a suspected bug in the link editor.
–VS *num*	Puts the specified decimal version stamp *num* in the object file that the link editor produces.
–v	Prints the name of each file as it is processed by the link editor.
–x	Retains external and static symbols in the symbol table to allow some debugging facilities. Doesn't retain local (non–global) symbols.

Note: There are certain restrictions in mixing compiler options. These include:

- *-O3* cannot be used with *-call_shared.*
- *-mips2* cannot be used with *-shared.*

- *-cord* cannot be used with *-shared.*
- *-trapuv* cannot be used with *-shared.*

Runtime Linker (rld)

The runtime linker *(rld)* performs symbol resolution dynamically during runtime *(dynamic linking).* It maps into memory the dynamic shared objects (created by *ld*) which are used by the executable.

rld does the following:

- Checks that the objects used at linktime are the same objects being used at runtime; i.e. objects have not been added, or deleted.
- Checks that each shared object was mapped into its default location.
- Checks that the timestamp, checksum, and interface version of each shared object has not changed since creation or since static linking.
- Constructs an explicit shared object list.
- Resolves each object's conflict list.
- Resolves each object's unresolved variable list.
- Allocates common if needed.

Quickstart

MIPS Application Binary Interface (ABI) includes a number of data structures, conventions, and implied mechanisms which constitute Quickstart. Quickstart requires that all dependencies between shared objects be resolved prior to runtime. It also requires that references between shared objects do not refer to multiple version of the same library.

Quickstart references the *so_locations* addresses.

Timestamp, Checksum and Interface Version

Conditions may have changed in the time between creating and using shared objects. For example, the list of objects used at link time may differ from those used at runtime.

The timestamp, the checksum, and the interface version are each checked separately by *rld* . If each of these match, then the Quickstart condition exists, and the runtime linker (*rld*) will not have to resolve any variables.

rld Options

Options to *rld* can be specified by the *_RLD_ARGS* environment variable to any combination of the options listed in the Table 2.2.

Table 2.2 Runtime Linker Options

Runtime Linker Options	
Option Name	**Purpose**
-clearstack	This option forces *rld* to zero any stack it uses before returning to user code.
-ignore_all_versions	Ignore versions on all objects.
-ignore_version shared_object	Ignore the version stamp checking on the object specified.
-ignore_unresolved	This option does not complain or abort when *rld* cannot resolve data symbols.
-interact	*rld* interactively prompts the user on standard input to fix problems in the link (e.g. *rld* asks the user to provide a full pathname for a missing shared object).
-log file	Prints all messages to a log file instead of standard output.
-pixie	Includes *rld* in the pixie statistics.
-stat	Prints *rld* statistics to standard output.
-trace	Prints all actions done for the user by *rld*.
-v	Prints general actions (less verbose than *-trace*).

Object File Tools

The following tools provide information on object files as indicated:

- **odump**: Displays the contents (including the symbol table and header information) of an object file in COFF format.
- **nm**: Displays only symbol table information.
- **file**: Provides descriptive information on the general properties of the specified file (for example, the programming language used).
- **size**: Prints the size of the *.init, .text, .rdata, .data, .sdata, .lit8, .lit4 .bss,* and *.sbss* sections. The format of these sections is described in Chapter 9 of the *MIPS Assembly Language Programmer's Guide.*
- **dump**: Displays the contents of an elf object file. For complete information on elf, refer to Chaper 11 in the *MIPS Assembly Programmer's Guide.*
- **strings**: Displays the printable strings in a file.

The sections that follow describe these tools in detail.

Dumping Selected Parts of Files (odump)

The *odump* tool displays headers, tables, and other selected parts of an object or archive file.

The syntax for the *odump* command is as follows:

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

where:

options is one or more of the options and suboptions listed in Table 2.3.

filename[1..N] are the names of one or more object files whose contents are to be dumped.

Figure 2.1 shows examples of output produced by *odump;* the command used to produce each is shown in a box. An explanation of the information provided by *odump* can be found in Chapters 9 and 10 of the *MIPS Assembly Language Programmer's Guide.*

Table 2.3 Odump Options

Main odump Options	
Options Name	**Purpose**
-a	Dumps the archive header of each member of the specified archive library file.
-c	Dumps the string table.
-D	Dumps the *.dynamic* section.
-Dc	Dumps the *.conflict* section.
-Dg	Dumps the GOT (global offset table) information.
-Dh	Dumps the hash table information.
-Di	Dumps the register information.
-Dl	Dumps the *liblist* information.
-Dr	Dumps the *.rel .dyn* information.
-Ds	Dumps the dynamic string information.
-Dt	Dumps the dynamic symbol information.
-F	Dumps the file descriptor table.
-f	Dumps each file header.
-G	Dumps the -G n histogram table.
-g	Dumps the global symbols in the symbol table of an archive library file.
-h	Dumps the section headers.
-i	Dumps the symbolic information header.
-L	Interpret and print contents of the *.lib* sections.
-l	Dumps line number information.
-o	Dumps each optional header.
-P	Dumps the procedure descriptor table.
-R	Dumps the relative file index table.
-r	Dumps relocation information.
-s	Dumps the section contents.
-t	Dumps symbol table entries.
-u	Underlines the name of the file for emphasis.

Table 2.3 Odump Options, 2 of 2

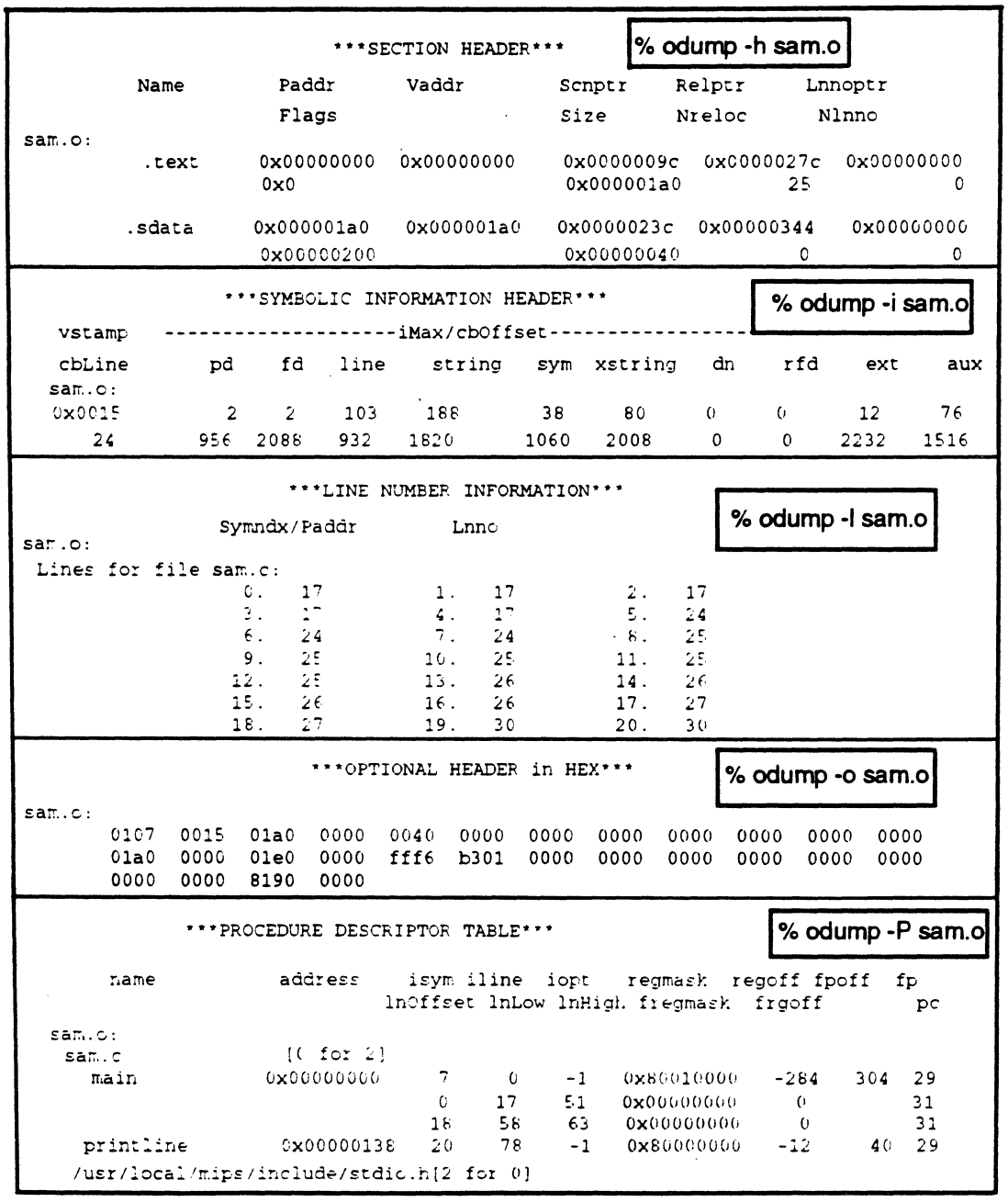
Auxiliary odump Options	
Option Name	**Purpose**
-d *number*	Dumps the section number, or a range of section numbers that starts at the specified number and ends with the last section number or the number you specify with +d.
+d *number*	Dumps the sections in a range that begins with the first section or with the section you specify with -d.
-n *name*	Dumps information only for the named entry. Use this option with -h, -l, -r, -s, and -t options.
-p	Suppress the printing of headers.
-t *index*	Dumps only the indexed symbol table entry. Specify a range of table entries by using this option with +t.
+t *index*	Dumps the symbol table entries in a range that ends with the indexed entry. The range begins with the first symbol table entry or with the section specified with -t.
-v	Dumps information in symbolic representation. Use this option with all dump options except -s.
-z *name, number*	Dumps the line number entry or a range of entries that start at the specified number for the named function.
+z *number*	Dumps the line number that starts at the function name or the number specified by -z, and ends at the number specified at +z.

```
                        ***STRING TABLE INFORMATION***
        [Offset]    Name

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

```
                              ***FILE HEADER***          % odump -f sam.o

          Magic   Nscns  Time/Date    Symptr      Nsyms     Opthdr  Flags
sam.o:
         0000540      2  0x1f22b3750x00000344        96     0x0038  0x0000
```

```
% odump -F sam.o          ***FILE DESCRIPTOR TABLE***

filename                              lnOffset ------iBase/count---------- merge sex
                        address  cbLine    sym   line  pd    aux   rfd  language
sam.o:

        sam.c          0x00000000     0      0      0     0     0     0     ---     el
                                     23     27    103     2    40     0       C
     ps/include/stdio.h0x00000000     0      2      0     2    40     0   merge el
                                      0     11      0     0    36           0  C
```

Figure 2.1 Example of Odump Utility Output, 1 of 4



% odump -h sam.o

```
                    ***SECTION HEADER***
        Name        Paddr       Vaddr       Scnptr      Relptr      Lnnoptr
                    Flags                   Size        Nreloc      Nlnno
sam.o:
        .text       0x00000000  0x00000000  0x0000009c  0x0000027c  0x00000000
                    0x0                     0x000001a0          25           0

        .sdata      0x000001a0  0x000001a0  0x0000023c  0x00000344  0x00000000
                    0x00000200              0x00000040           0           0
```

% odump -i sam.o

```
                ***SYMBOLIC INFORMATION HEADER***
  vstamp    --------------------iMax/cbOffset-----------------
  cbLine       pd     fd   line     string    sym   xstring    dn     rfd     ext     aux
sam.o:
0x0015          2      2    103        188     38        80     0       0      12      76
    24        956   2088    932       1820   1060      2008     0       0    2232    1516
```

% odump -l sam.o

```
                    ***LINE NUMBER INFORMATION***
                Symndx/Paddr        Lnno
sam.o:
 Lines for file sam.c:
                    0.  17          1.  17          2.  17
                    3.  17          4.  17          5.  24
                    6.  24          7.  24          8.  25
                    9.  25         10.  25         11.  25
                   12.  25         13.  26         14.  26
                   15.  26         16.  26         17.  27
                   18.  27         19.  30         20.  30
```

% odump -o sam.o

```
                    ***OPTIONAL HEADER in HEX***
sam.o:
        0107  0015  01a0  0000  0040  0000  0000  0000  0000  0000  0000  0000
        01a0  0000  01e0  0000  fff6  b301  0000  0000  0000  0000  0000  0000
        0000  0000  8190  0000
```

% odump -P sam.o

```
            ***PROCEDURE DESCRIPTOR TABLE***
      name          address     isym iline  iopt    regmask  regoff fpoff   fp
                              lnOffset lnLow lnHigh  fregmask  frgoff        pc
 sam.o:
  sam.c             [0 for 2]
    main          0x00000000      7     0    -1   0x80010000    -284   304  29
                                  0    17    51   0x00000000       0        31
                                 18    58    63   0x00000000       0        31
   printline      0x00000138     20    78    -1   0x80000000     -12    40  29
  /usr/local/mips/include/stdio.h[2 for 0]
```

Figure 2.1 Example of Odump Utility Output, 2 of 4

% odump -r sam.o

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

% odump -R sam.o

```
            ***RELATIVE FILE INDEX TABLE***

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

% odump -s sam.o

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

Figure 2.1 Example of Odump Utility Output, 3 of 4

% odump -t sam.o

```
          ***SYMBOL TABLE INFORMATION***
[Index]     Name          Value         Sclass    Symtype    Ref
sam.o:
[0]         sam.c         0x00000000    0x01      0x0b       0x001b
[1]         line          0x00000108    0x0b      0x07       0x0006
[2]         string        0x00000000    0x0b      0x09       0x000e
[3]         length        0x00000800    0x0b      0x09       0x0004
[4]         linenumber    0x00000820    0x0b      0x09       0x0004
[5]                       0x00000000    0x0b      0x08       0x0001
[6]         LINETYPE      0x00000000    0x0b      0x0a       0x0013
[7]         main          0x00400000    0x01      0x06       0x0017
[8]         argc          0x00000000    0x05      0x03       0x0004
[9]         argv          0x00000004    0x05      0x03       0x0019
[10]                      0x00000014    0x01      0x07       0x0013
[11]        line1         0xfffffef8    0x05      0x04       0x001a
[12]        fd            0xfffffef4    0x05      0x04       0x001c
[13]        i             0xfffffef0    0x05      0x04       0x0004
[14]                      0x000000ac    0x01      0x07       0x0012
[15]        i             0xfffffee8    0x05      0x04       0x0004
[16] curlinenumber        0x000001c8    0x0d      0x02       0x0004
[17]                      0x00000108    0x01      0x08       0x000e
[18]                      0x00000120    0x01      0x08       0x000a
[19]        main          0x00000138    0x01      0x08       0x0007
[20]        printline     0x00000138    0x01      0x06       0x0015
[21]        pline         0x00000000    0x05      0x03       0x0024
[22]                      0x0000000c    0x01      0x07       0x0019
[23]        i             0xfffffffc    0x05      0x04       0x0004
[24]                      0x0000004c    0x01      0x08       0x0016
[25]        printline     0x00000064    0x01      0x08       0x0014
[26]        sam.c         0x00000000    0x01      0x08       0x0000
[27] /usr/local/mips/include/stdio.h0x00000000 0x01     0x0b   0x0026
[28]        _iobuf        0x00000018    0x0b      0x07       0x0025
[29]        _cnt          0x00000000    0x0b      0x09       0x002c
[30]        _ptr          0x00000020    0x0b      0x09       0x0036
[31]        _base         0x00000040    0x0b      0x09       0x0037
[32]        _bufsiz       0x00000060    0x0b      0x09       0x002c
[33]        _flag         0x00000080    0x0b      0x09       0x002b
[34]        _file         0x00000090    0x0b      0x09       0x0030
[35]        _name         0x000000a0    0x0b      0x09       0x0038
[36]                      0x00000000    0x0b      0x08       0x001c
[37] /usr/local/mips/include/stdio.h0x00000000    0x01     0x08   0x001b
[38]        _iob          0x000001e0    0x15      0x01       0x0039
[39]        fopen         0x00000000    0x06      0x06       0x003f
[40]        fdopen        0x00000000    0x00      0x06       0x0035
[41]        freopen       0x00000000    0x00      0x06       0x0036
[42]        ftell         0x00000000    0x00      0x06       0x0031
[43]        fgets         0x00000000    0x06      0x06       0x004a
[44]        printline     0x00000138    0x01      0x06       0x0014
[45]        main          0x00000000    0x01      0x06       0x0007
[46]        fprintf       0x00000000    0x06      0x06       0x001e
[47]        exit          0x00000000    0x06      0x06       0x0020
```

Figure 2.1 Example of Odump Utility Output, 4 of 4

Listing Symbol Table Information (nm)

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

The syntax for the *nm* command is as follows:

```
nm [options] finename1 [filename2 . . filenameN]
```

where:

options is one or more characters (listed in Table 2.2) that specify the type of information to be printed.

filename[1..N] specify the object file(s) or archive file(s) from which symbol table information is to be extracted. If you don't specify a file, *nm* assumes *a.out*.

For more information, please see *nm*(1) in the *RISC/os Programmer's Reference Manual.*

Table 2.4: Symbol Table Dump (nm) Options (-systype svr3)

nm Options (svr3)	
Option Name	**Purpose**
-A	Prints the listing in System V format.
-a	Prints debugging information. Turns BSD output into System V format.
-B	Prints the listing in BSD format.
-b	Prints the value field in octal.
-d	Prints the value field in decimal.
-e	Prints only external and static variables.
-g	Prints only global symbols.
-h	Suppresses printing of headers.
-n	Sorts external symbols by name for System V format. Sorts all symbols by value for BSD format.
-o	Prints the value field in octal for System V output. Prints the filename immediately before each symbol name for BSD format.
-p	Lists symbols in the order they appear in the Symbol table.
-r	Reverses the sort which you specified for external symbols with the -n and -v options.
-T	Truncates characters in exceedingly long symbol names; inserts an asterisk as the last character of the truncated name. This may make the listing easier to read.
-u	Prints only undefined symbols.
-V	Prints the version number of nm.
-v	Sorts external symbols by value.
-x	Prints the value field in hexadecimal.

Table 2.5: Symbol Table Dump (nm) Options (-systype svr4)

nm Options (svr4)	
Option Name	**Purpose**
-e	Prints only external and static variables, obsolete.
-f	Produce full output, obsolete.
-h	Suppresses printing of headers.
-n	Sorts external symbols by name for System V format. Sorts all symbols by value for BSD format.
-l	Append an * to the key letter for *weak* symbols.
-o	Prints the value field in octal for System V output. Prints the filename immediately before each symbol name for BSD format.
-P	Produce terse output.
-r	Prepend object file or archive name to each output line.
-T	Truncate long symbol names, obsolete.
-u	Prints only undefined symbols.
-V	Prints the version number of nm.
-v	Sorts external symbols by value.
-x	Prints the value field in hexadecimal.

Figure 2.2 shows an example of an *nm –B* command and the output it produces. Note that each item has a key describing its storage class.

Example:

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

value field — key — symbol name

Figure 2.2 Symbol Table in BSD Format (option –B)

Table 2.6 describes the meanings of the character keys shown in the example above.

Table 2.6 nm Character Key Meanings

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

Figure 2.3 shows an example of *nm* output in System V format.

Symbols from sam.o:

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

[1] For information on these fields, refer to Chapter 10 of the MIPS Assembly Programmer's Guide

Figure 2.3 Symbol Table in System V Format (option –A)

Determining a File's Type (file)

The *file* tool lists the properties of program source, text, object, and other files. This tool often erroneously recognizes command files as C programs. It does not recognize Pascal or LISP programs. For more information, see the *file*(1) manual page in the *RISC/os User's Reference Manual.*

The syntax of the *file* command is as follows:

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

Example:

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


Determining a File's Section Sizes (size)

The *size* tool prints information about the *text, rdata, data, sdata, bss,* and *sbss* sections of the specified object or archive file(s). The contents and format of section data are described in Chapter 9 of the *Assembly Language Programmer's Guide.*

The syntax for the size command is as follows:

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

where:

options is in alphabetic character (listed in Table 2.6) that specifies the format of the output.

filename[1..N] specify the object or archive file(s) whose properties are to be displayed. If a file name is not specified, *size* uses *a.out.*

For more information, see *size*(1) in the *RISC/os Programmer's Reference Manual.*

Table 2.7: Size Options

size Options	
Option Name	**Purpose**
-A	Prints data section headers in System V format. The default is determined by the UNIX version running on your system.
-B	Prints data section headers in BSD format.
-d	Prints the section sizes in decimal.
-F	Prints the size and permission flags of each loadable segment and the total of the loadable segments.
-f	Prints the size and name of each allocatable section and the total allocatable section size.
-n	Prints non-loadable segment or non-allocatable section size information.
-o	Prints the section sizes in octal.
-V	Prints the version of size currently being used.
-x	Print the section sizes in hexadecimal.

Note: svr3 environment size options are: -A, -B, -d, -o, -V, and -x.

Note: svr4 environment size options are: -F, -f, -n, -o, -V, -x.

Figure 2.4 shows an example of *size* output.

```
% size test
                    Size of test:27776
  Section         Size     Physical Address   Virtual Address
    .text         19840              4194672           4194672
    .init            32              4214512           4214512
    .rdata         1072            268435456         268435456
    .data          4640            268436528         268436528
    .sdata          592            268441168         268441168
    .sbss            64            268441760         268441760
     .bss          1536              268441824           268441824
```

Figure 2.4 Sample size output

Archiver

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

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

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

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

The syntax of the *ar* command is as follows:

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

where:

options is one or more characters (listed in Tables 2.7 and 2.8) that specify the action that the archiver is to take. When specifying more than one option character, group the characters together with no spaces between; don't place a dash (-) character before the option characters.

posObject is the name of an object within an archive library. It specifies the relative placement (either before or after *posObject*) of an object that is to be copied into the library or moved within the library. A *posObject* is required when the m or r options are specified together with the a, b, or i suboptions.

libName is the name of the archive library you are creating, updating, or extracting information from.

object [1..N] are the names of the object(s) or object file(s).

ar Command Examples

To create a new library and add routines to it:

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

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

To add or replace an object (.o) file to an existing library:

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

Option r replaces *mon1.o* in the library *libtest.a*. If *mon1.o* doesn't exist, the new object *mon1.o* is added.

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

To update the library's *symdef* table:

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

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

Note: After creating or changing a library, use the s option to update the *symdef* (symbol definition) table of the archive library. The link editor uses the symdef table to locate objects during the link process.

To add a new file immediately before a specified file in the library:

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

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

Archiver Options

Table 2.7 lists the archiver options. You must specify one of the following options: d, m, p, q, r, or x. In addition, you can specify the c, l, s, t, and v options, and any of the archiver suboptions.

Table 2.8 Archiver Options

Archiver Options	
Option Name	**Purpose**
c	Suppresses the warning message that the archiver issues when it discovers that the specified archive doesn't exist.
d	Deletes the specified objects from the archive.
l	Puts the archiver temporary files in the current working directory. Ordinarily the archiver puts those files in */tmp*. This option is useful when */tmp* is full.
m	Moves the specified files to the end of the archive. If you want to move the object to a specific position in the archive library, specify an a, b or i suboption together with the *posObject* parameter.
p	Prints the specified object(s) in the archive on the standard output device (usually the terminal screen).
q	Adds the specified object files to the end of the archive. An existing object file with the same name is *not* deleted, and the link editor continues to use the old file. This option is similar to the r option (described below) but it is faster. Use it when creating a new library.
r	Adds the specified object files to the archive. This option deletes duplicate objects in the archive. To add the object at a specific position in the archive library, specify an a, b, or i suboption together with the *posObject* parameter. See the examples in the preceding section for an example of using the *posObject* parameter. Use the r option when updating existing libraries. See also the u suboption.
s	Creates a *symdef* file in the archive. Use this option each time you create or change the archive library. If all objects don't have the same endian byte ordering scheme, the archiver issues an error message and doesn't create a *symdef* table. At least one of the following options must be specified with the s option: m, p, q, r, or t.
t	Prints a table of contents on the standard output (usually the screen) for the specified object or archive file.
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.
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. Then the date stamp on the archive file is the last modified.

Table 2.8 lists the *ar* suboptions.

Table 2.9 Archiver Suboptions

Archiver Suboptions		
Suboption Name	**Use With**	**Purpose**
a	m or r	Specifies that the object file follows the posObject file specified in the ar statement.
b	m or r	Specifies that the object file precedes the posObject file specified in the ar statement.
i	m or r	Same as b.
o	x	Used when extracting a file from the archive to the current directory. Forces the last modified date of the extracted file to match that of the archive file.
u	r	Replaces that existing object file when the last modified data is earlier (precedes) that of the new object file.
-z		Suppresses symbol table building.

Storage Mapping

3

C Language

This chapter describes the alignment, size, and value ranges for the C language, and the storage of data in memory. The following topics are discussed:

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

Alignment, Size, and Value Ranges

Table 3.1 shows the C compiler size, alignment, and value ranges for the data types.

Table 3.1 Size, Alignment, and Value Ranges for C Data Types

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

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

Note: Approximate valid ranges for *float* and *double* are:

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

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

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

```
signed char c
```

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

The following sections describe how the data types shown in Table 3.1 affect arrays, structures, and unions.

Storage of C Arrays, Structures, and Unions

Arrays

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

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

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

Structures

Each member of a structure begins at an offset from the structure base. The offset corresponds to the order and size of the members within the structure; the first member is at offset 0.

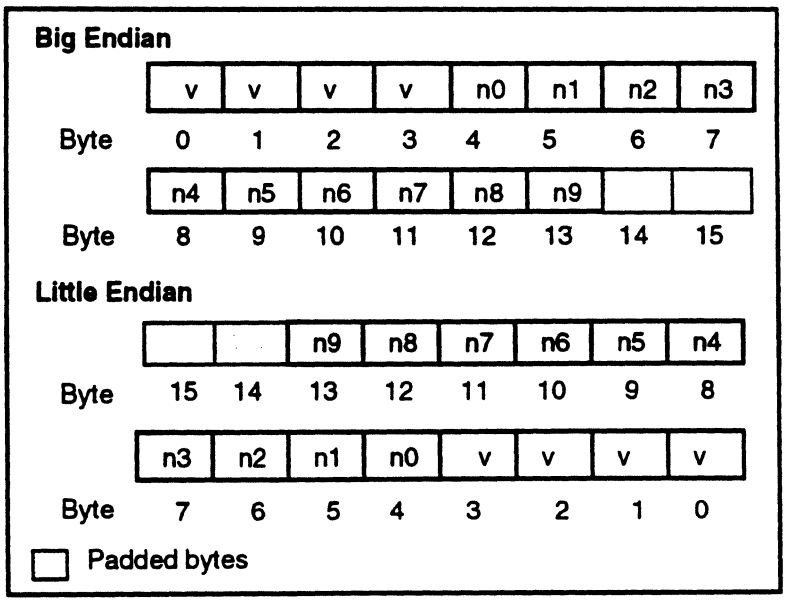
The size of a structure in the object file is the size of its combined members plus padding added, where necessary, by the compiler. The following rules apply to structures:

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

For example, the following structure:

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

is mapped in storage as follows:



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

Note that the length of the structure is 16 bytes, even though the byte count as defined by the *int v* and the *char n* components is only 14. *int* has a stricter boundary requirement (word boundary) than *char* (byte

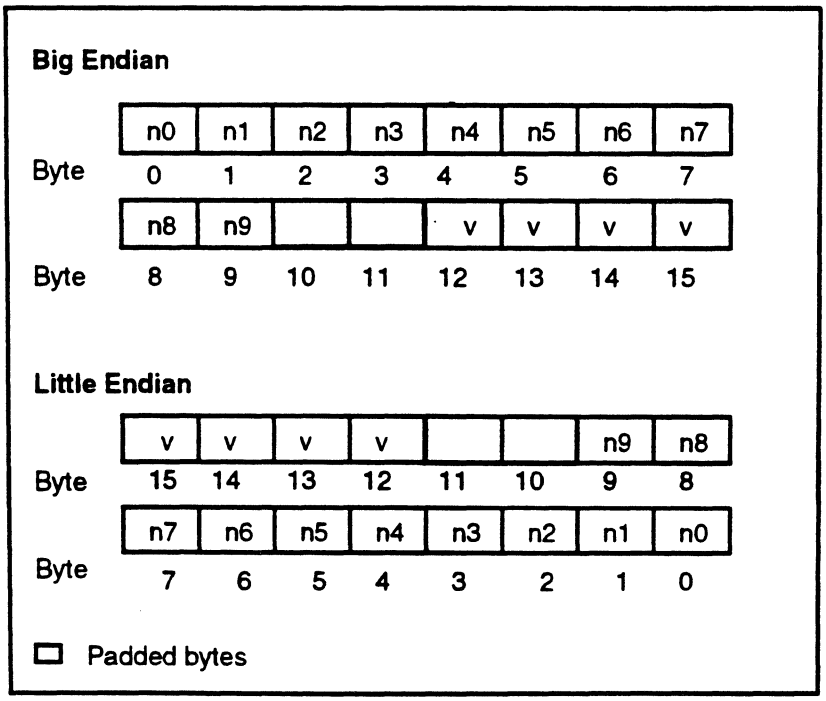
boundary); the structure must end on a word boundary (a byte offset divisible by four). The compiler adds two bytes of padding to meet this requirement.

For example, if the above structure, `struct s`, were the element-type of an array, some of the *int v* components wouldn't be aligned properly without the two-byte pad.

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

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

the compiler maps the structure as follows:

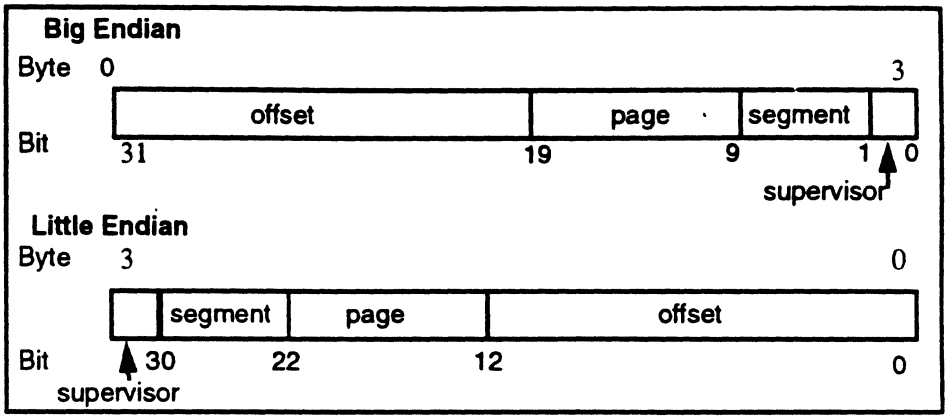


Note that the size of the structure remains 16 bytes, but two bytes of padding follow the *n* component to align *v* on a word boundary.

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

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

is mapped as follows:

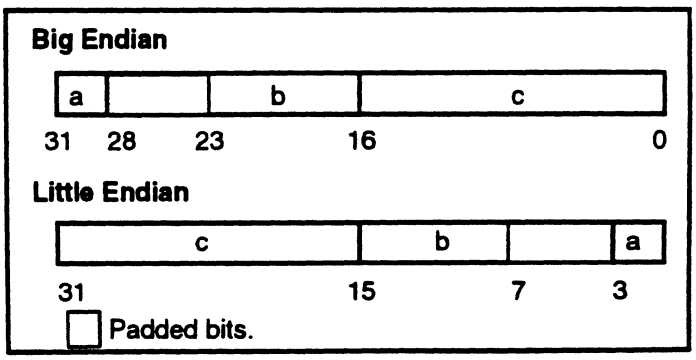


The compiler moves fields that overlap a word boundary to the next word.

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

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

is mapped as follows:



Note that five bits of padding are added after *unsigned a* so that *char b* aligns on a byte boundary, as required.

Unions

A union must align on the same boundary as the member with the most restrictive boundary requirement. For example, a union containing *char*, *int*, and *double* data types must align on a a doubleword boundary, as required by the *double* data type.

Storage Classes

Auto

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

Static

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

Register

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

Extern

The *extern* storage class indicates that the variable refers to storage defined in an external data definition. The compiler does not allocate storage to *extern* variable declarations; *extern*'s are defined and referenced as follows:

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

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

Volatile

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

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

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

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

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

Language Interfaces

4

This chapter describes the calling interfaces between C and Pascal and C and Fortran, including rules and examples for calling and passing arguments between these languages.

You may need to refer to Chapter 3 for information on C data storage.

Pascal/C Interface

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

- Single–precision floating point.
- Procedure and function parameters.
- Pascal by–value arrays.
- File variables.
- Passing string data between C and Pascal.
- Passing variable arguments.

These differences are discussed in the following sections.

Single Precision floating point

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

- If possible, write the Pascal routine so that it receives and returns double–precision values, or
- If the Pascal routine cannot receive a double–precision value, write a Pascal routine to accept double–precision values from C, then have that routine call the single–precision Pascal routine, or
- Use C prototypes to cause floats to be passed directly.

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

Procedure and function parameters

C function variables and parameters consist of a single pointer to machine code, whereas Pascal procedure and function parameters consist of a pointer to machine code and a pointer to the stack frame of the lexical parent of the function. Such values can be declared as structures in C. To create such a structure, put the C function pointer in the first word, and 0 in the second. C functions cannot be nested, and thus have no lexical parent; therefore, the second word is irrelevant.

A C routine with a function parameter cannot be called from Pascal.

Pascal by–value arrays

C never passes arrays by value. In C, an array is actually a type of pointer; passing an array passes its address, which corresponds to Pascal by–reference (VAR) array passing. In practice this is not a serious problem because passing Pascal arrays by value is not very efficient, and most

Pascal array parameters are VAR. When it is necessary to call a Pascal routine with a by–value array parameter from C, pass a C structure containing the corresponding array declaration.

File Variables

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

```
struct pascal_file {
      FILE *stdiofile;
      char *name;
};
```

Strings

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

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

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

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

Figure 4.1 shows a Pascal routine that calls the C routine *atoi* and passes the string s. Note that the routine ensures that the string terminates with a null character.

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

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

program ptest(output);
  var
    s: astring;
    i: astrindex;
  begin
  argv(1, s); { Extension to Pascal }

  writeln(output, s);
  { Guarantee that the string isnull-terminated
    (but may bash the last character if the argument
  is too long).  "lbound" and "hbound" are
    extensions. }
  s[hbound(s)]:= chr(0);
  for i := lbound(s) to hbound(s)do
    if s[i]  = ' ' then
              begin
              s[i] := chr(0);
              break;
              end;
  writeln(output, atoi(s));
  end.
```

Terminates with character

Figure 4.1: Calling a C Routine from Pascal

For more information on *atoi*, see the *atof*(3–BSD) or *strtol*(3c–SysV) man page in the *RISC/os Programmer's Manual*. See Figure 4.5 for another example of passing strings between C and Pascal.

Variable number of arguments

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

Type checking

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

Main() Routine

Only one main routine is allowed per program. The main routine can be written either in Pascal or C. Figure 4.2 shows examples of C and Pascal main routines:

Pascal	C

`program p(input,output);`
`begin`
`  writeln('hi!');`
`end.`
	`main() {`
	`  printf("hi\n!");`
	`}`

Figure 4.2: main() routines

Calling Pascal from C

To call a Pascal function from C, write a C extern declaration to describe the return value type of the Pascal routine; write the call with the return value type and argument types as required by the Pascal routine (see Figure 4.1).

Return Values

Table 4.1 shows the return value type of a C function that accepts Pascal return values.

Table 4.1: Declaration of Return Value Types

If Pascal function returns:	Declare C function as:
integer, integer32 [1]	int
cardinal [2]	unsigned int
integer16	short
char	char
boolean	char
enumeration	unsigned, or corresponding enum enum (C's enum are signed)
real	float
double	double
pointer type	corresponding pointer type
record type -	corresponding structure or union type
array type	structure containing corresponding array type.

1 Applies also to subranges withlowers bound <0.
2Applies also to subranges withlower bounds >=0.

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

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

and then call it with arguments with appropriate types. Table 4.2 shows the values to pass corresponding to the Pascal declarations. C does not permit declaration of the formal parameter types, but instead infers them from the types of the actual arguments passed (see Figure 4.4).

C to Pascal arguments

Table 4.2 shows the C argument types to declare in order to match those expected by the called Pascal routine.

Table 4.2: Pascal to C Argument Types

If Pascal expects:	**C argument should be:**
integer, integer32	integer or char value -2^{31} ..2^{31} - 1
cardinal	integer or char value 0..2^{32} -1
integer16	char value -2^{15}.. $2^{15}-1$
subrange	integer or char value in subrange
char	integer or char (0..255)
boolean	integer or char (0 or 1 only)
enumeration	integer or char (0..N-1)
real	none
double	float or double
procedure	struct {void *p(); int *l}
function	struct {function-type *f(); int *l}
pointer types	pointer type und <0. := lbound(s)
parameter reference	pointer to the appropriate type
record types	structure or union type
by-reference array parameters	corresponding array type
by–reference–file	pointer to the appropriate structure
by-value array parameters	structure containing the corresponding array

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

Example: Calling a Pascal function

Figure 4.3 shows an example of a C routine calling a Pascal function.

Pascal routine

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

C declaration of bah

```
extern double bah();
```

C call

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

Figure 4.3: Calling a Pascal Function from C

Example: Calling a Pascal procedure

Figure 4.4 shows an example of a C routine calling a Pascal procedure.

Pascal routine

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

C declaration

```
extern void zero();
```

C call

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

Figure 4.4: Calling a Pascal Procedure from C

Example: Passing strings to a Pascal procedure

Figure 4.5 shows an example of a C routine that passes strings to a Pascal procedure, which then prints them; the example illustrates two points:

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

C routine

```
/* Send the last command-line argument
to the Pascal routine */

struct pfile {
   FILE *stdiofile;
   char *name;
};


#include <stdio.h>
main(argc, argv)
   int argc; char **argv;
   {
   struct pfile temp;
   temp.stdiofile = stdout;
   temp.name = "stdout";
   if (argc != 0)
     p_routine(&temp, argv[argc -1]);
   }
```

Pascal routine

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

Checks for null character.

Writes to file-stream descriptor passed by C.

Figure 4.5: Passing Strings to a Pascal Procedure from C

Calling C from Pascal

Pascal to C arguments

To call a C routine from Pascal, write a Pascal declaration describing the C routine. Use a procedure declaration or, if the C routine returns a value, a function declaration. Parameter and return value declarations should correspond to the C parameter types, as shown in Table 4.3.

Table 4.3: Pascal Parameter Data Type Expected by C

If C expects:	Pascal parameter should be:
int [1]	integer
unsigned int [2]	cardinal
short [3]	integer or integer16 (–32768..32767)
unsigned short	cardinal (or 0..65535)
char [4]	char
signed char	integer (or -128..127)
float	float
double	double
enum type	corresponding enumeration type
string (char *)	packed character array passed by reference (VAR)
pointer to function	none
FILE *	none
pointer type	corresponding pointer type or corresponding type passed by reference (VAR)
struct type	corresponding record type
union type	corresponding record type
array type	corresponding array type passedby reference (VAR)

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

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

Example: Calling a C procedure

Figure 4.6 shows an example of calling a C procedure from Pascal.

C routine:

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

Pascal declaration:

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

Pascal call:

```
var str: string;
str := "abc\"
bah(i, 1.0, str)
```

Figure 4.6: Calling a C Procedure from Pascal

Example: Calling a C function

Figure 4.7 shows an example of calling a C function from Pascal.

C routine:

```
float humbug (f, x)
    struct f {
       FILE *stdiofile;
       char *name;
    };
    struct scrooge *x;
{
    ...
}
```

Pascal declaration:

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

Pascal call:

```
var sp: scrooge_ptr;
x := humbug(input, sp);
```

Figure 4.7: Calling a C Function from Pascal

Example: Passing arrays

Figure 4.8 shows an example of passing an array to a C function from Pascal.

C routine:

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

Pascal declaration:

```
type
  int_array = array[0..100] of integer;
function sum (
    var a: int_array;
    n: cardinal
   ): integer:
  external;
```

Pascal Call:

```
var samples: int_array;

avg := sum(samples,hbound(samples) + 1) /
       (hbound(samples)+1);
```

Figure 4.8: Passing Arrays Between Pascal and C

FORTRAN/C Interface

This section discusses items to consider when writing a function call between FORTRAN and C.

Procedure and Function Names

In calling a FORTRAN subprogram from C, the C program must append an underscore (_) to the name of the FORTRAN subprogram. For example, if the name of the subprogram is *matrix*, then refer to it as *matrix_*. When FORTRAN is calling a C function, the name of the C function must end with an underscore.

Note that only one main routine is allowed per program. The main routine can be written in either C or FORTRAN. Figure 4.9 shows an example of a C and a FORTRAN main routine.

C	FORTRAN

`main() {`
`printf("hi!\n");`
`}`
	`write(6,10)`
	`10 format('hi!')`
	`end`

Figure 4.9: C and Fortran main() routines

Invocations

Invoke a FORTRAN subprogram as if it were an integer-valued function whose value specifies which alternate return to use. Alternate return arguments (statement labels) are not passed to the subprogram but cause an indexed branch in the calling subprogram. If the subprogram is *not* a function and has no entry points with alternate return arguments, the returned value is undefined. The FORTRAN statement

```
call nret(*1,*2,*3)
```

is treated exactly as if it were the computed goto

```
goto (1,2,3), nret()
```

A C function that calls a FORTRAN subprogram can usually ignore the return value of a FORTRAN subroutine; however, the C function should not ignore the return value of a FORTRAN function. Figure 4.10 shows equivalent function and subprogram declarations in C and FORTRAN programs:

C Function Declaration	FORTRAN Declaration
`double dfort_()`	`double precision function dfort()`
`float rfort()`	`real function rfort()`
`int ifort_()`	`integer function ifort()`
`int ifort_()`	`logical function lfort()`

Figure 4.10: C and FORTRAN Function and Subprogram Declarations

Note the following:

- Avoid calling FORTRAN functions of type *complex* and *character* from C.
- You cannot return complex types between C and FORTRAN.
- A character-valued FORTRAN subprogram is equivalent to a C language routine with two extra initial arguments: a data address and a length.

Thus:

```
character*15 function g(...)
```

is equivalent to:

```
char result[];
long int length;
g_(result, length, ...)
```

and could be invoked in C by:

```
char chars[15];
g_(chars, 15);
```


Arguments

The following rules apply to arguments passed between FORTRAN and C:

- All arguments must be passed by reference. That is, the argument must specify an *address* rather than a value. Thus, to pass constants or expressions, their values must be first stored in variables and the address of the variable passed.

- When passing the address of a variable, the data representations of the variable in the calling and called routines must correspond, as shown in Table 4.4.

Table 4.4: Equivalent FORTRAN and C Data Types

FORTRAN	C
`integer*2 x`	`short int x;`
`integer x`	`long int x; or just int x;`
`logical x`	`long int x;`
`real x`	`float x;`
`double precision x`	`double x;`
`complex x`	`struct { float real, imag; }x;`
`double complex x`	`struct { double dreal, dimag;} x;`
`character*6 x`	`char x[6];`

Note that FORTRAN requires that each *integer, logical,* or *real* variable occupy 32 bits of memory.

- The FORTRAN compiler may add items not explicitly specified in the source code to the argument list. The compiler adds the following items under the conditions specified:
- Destination address for character functions, when called.
- Length of a character string, when an argument is the address of a character string.

When a C program calls a FORTRAN subprogram, the C program must explicitly specify these items in its argument list *in the following order*:

a. Destination address of character functions.

b. Normal arguments (addresses of arguments or functions).

c. Length of character strings. The length must be specified as an absolute value or **integer** variable.

The next two examples illustrate these rules.

Example 1: Figure 4.11 shows how a C routine must specify the length of a character string (which is only implied in a FORTRAN call).

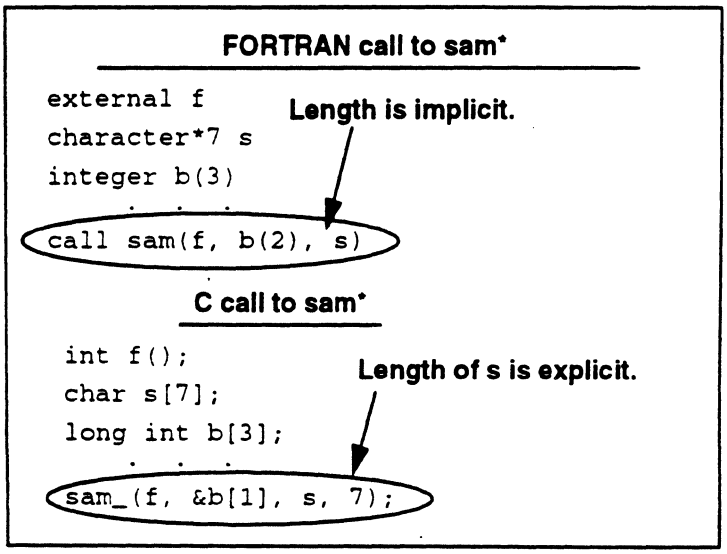
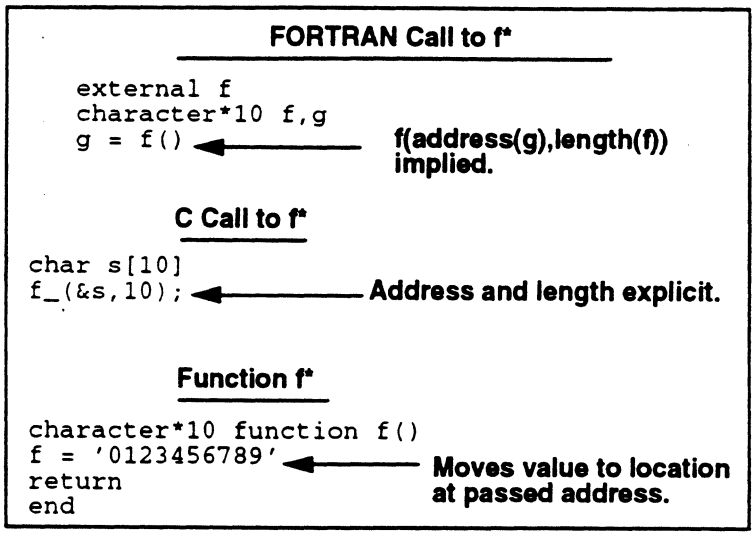


Figure 4.11: Character String length in C and FORTRAN

Example 2: Figure 4.12 shows how a C routine can specify the destination address of a FORTRAN function (which is only implied in a FORTRAN program).



*f is a function written in FORTRAN.

Figure 4.12: Address of a FORTRAN Function

Array Handling

FORTRAN stores arrays in column-major order with the leftmost subscript varying the fastest. C, however, stores arrays in the opposite arrangement, with the rightmost subscripts varying the fastest, which is called row-major order. Figure 4.12 shows the layout of FORTRAN arrays and C arrays:

FORTRAN

```
integer t(2,3)
 t(1,1),  t(2,1),  t(1,2),  t(2,2),  t(1,3),  t(2,3)
```

C

```
int t[2][3];
 t[0][0], t[0][1], t[0][2], t[1][0], t[1][1], t[1][2]
```

Figure 4.13: Array Storage in C and FORTRAN

Note that the default for the lower bound of an array in FORTRAN is 1, whereas it is 0 in C.

When a C routine uses an array passed by a FORTRAN subprogram, the dimensions of the array and the use of the subscripts must be interchanged, as shown in Figure 4.14.

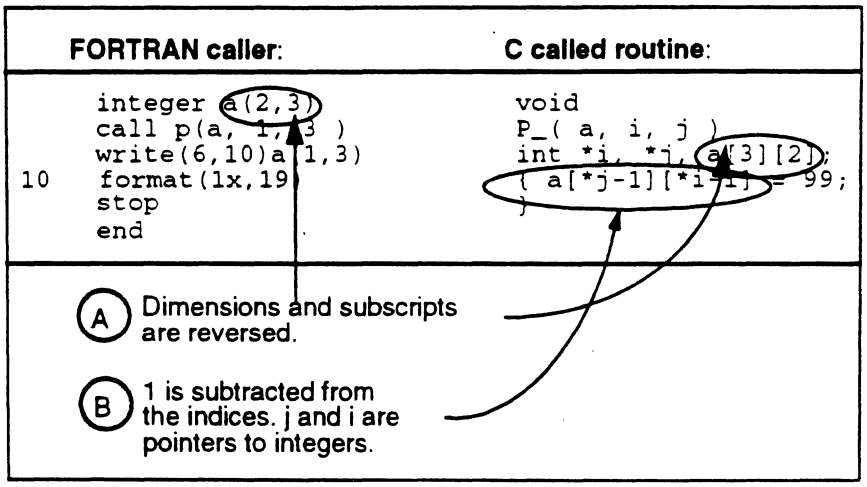


Figure 4.14: Array Subscripts and Dimensions

The FORTRAN caller prints out the value 99. Note the following:

(A) Because arrays are stored in column–major order in FORTRAN and row–major order in C, the dimension and subscript specifications are reversed.

(B) In FORTRAN, the lower–bound default is 1, whereas it is 0 in C; therefore, 1 must be subtracted from the indices in the C routine. Also, because FORTRAN passes parameters by reference, the **j* and **p* are pointers used in the C routine.

Accessing Common Blocks of Data

The following rules apply to accessing common blocks of data:

- FORTRAN common blocks must be declared by *common* statements; C can use any global variable. Note that the common block name in C (sam_) must end with an underscore.

- Data types in the FORTRAN and C programs must match unless you desire equivalencing. If so, you must adhere to the alignment restrictions for the data types described in Chapter 3.
- If multiple routines define the same common block with unequal lengths, the largest of the sizes is used to allocate space.
- Unnamed common blocks are given the name *_BLNK__*.

Figure 4.15 shows examples of C and FORTRAN routines that access common blocks of data.

C	FORTRAN

`struct S {int i; float j;}r_;`
`main() {`
`sam_();`
`printf("%d %f\n", r_.i, r_.j);`
`}`
	`subroutine sam()`
	`common /r/i,r`
	`i = 786`
	`r = 3.2`
	`return`

Figure 4.15: Accessing Common Data in C and FORTRAN

The C routine prints out 786 and 3.2.

Improving Program Performance

5

This chapter describes tools that can help reduce the execution time of programs; the following topics are covered:

- Profiling and how to use it to isolate those portions of code where execution is concentrated and provide reports that indicate where improvements might be made.
- How to use Optimization and examples showing optimization techniques.
- Limiting the Size of Global Data Area and how, through controlling the size of variables and constants that the compiler places in this area, program performance can be improved.

Introduction

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

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

As technological advances cause MIPS to make changes to the current compiler system, anything tailored now might negatively affect future program performance. Moreover, tailored code might not work at all with new versions of the system. Report any possible compiler inefficiencies directly to MIPS.

Profiling

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

Overview

Profiling helps find the areas of code where most of the execution time is spent. In the typical program, execution time is confined to relatively few sections of code; it's profitable to concentrate on improving coding efficiency in only those sections.

Profiling provides the following information:

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

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

- Invocation counting, which gives the number of times each procedure in the program is invoked.

- Basic block counting, which measures the execution of basic blocks (a basic block is a sequence of instructions that is entered only at the beginning and which exits only at the end). This option provides statistics on individual lines.

  You obtain invocation counting and basic block counting information using the *pixie* program. *Pixie* creates a program equivalent to your program containing additional code that counts the execution of each basic block. Executing *pixie* and the equivalent program generates the profile data in raw format.

Using the *prof* program, you can create a formatted display of the raw profile data. The output can indicate where to improve code, substitute better algorithms, or substitute assembly language. The output also indicates if the program has exercised all portions of the code. The *pixstats* program can also be used to analyze this data.

Figure 5.1: shows an example of output produced by a program compiled with the *–p* compiler option; *prof* was used with the *–procedure* option to produce the output.

Procedures: - PC Sampling
Profiler option: -procedure

```
-------------------------------------------------------------------
*  -p[rocedures] using pc-sampling;                               *
*  sorted in descending orderby total time spent in each procedure;  *
*  unexecuted procedures excluded                                 *
-------------------------------------------------------------------

Each sample covers 8.00 byte(s) for 4.2% of 0.2400 seconds

%time     seconds  cum %     cum sec  procedure (file)

 25.0      0.0600   25.0      0.06    main (fixfont.p)
 16.7      0.0400   41.7      0.10    write_string (../textoutput.c)
 12.5      0.0300   54.2      0.13    write_char (../textoutput.c)
 12.5      0.0300   66.7      0.16    write_integer (../textoutput.c)
```

Figure 5.1: Profiler Listing for PC Sampling

The highlighted line in the figure above shows:

a. .03 seconds or 12.5% of execution time was spent in *write_integer*.

b. .16 seconds or 66.7% of total execution time was spent in *main*, *write_string*, *write_char*, and *write_integer* routines combined.

c. The name of the source file for *write_integer* is *../textoutput.c*.

Figures 5.2 through 5.6 show raw data produced by *pixie*. The *prof* option used is given at the top of each figure.

Procedures: - Invocation Counting
Profiler option: -pixie -invocation

```
-------------------------------------------------------------------------------
*  -i[nvocations] using basic-block counts;                                   *
*  the called procedures are sorted in descending order by number of          *
*  calls; a '?' in the columns marked '#calls' or 'line' means that data      *
*  is unavailable because partof the program was compiled without             *
*  profiling.                                                                 *
-------------------------------------------------------------------------------
 called procedure  #calls  %calls   from line calling procedure (file):

   eoln            4017    81.51       37     main (pix.p)
                    453     9.19       35     main (pix.p)
                    428     8.69       19     main (pix.p)
                     30     0.61       17     main (pix.p)
 write_char        4014    81.75       43     main (pix.p)
```

Figure 5.2: Profiler Listing for Procedure Invocations

The circled text in the figure above shows:

a. *eoln* was called 4,017 times from line 37 of *main*. This represented 81.51% of the calls to *eoln*.

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

Procedures: Basic Block Counts
Profiler option: -pixied -procedures

```
-------------------------------------------------------------------
* -p[rocedures] using basic-block counts;                          *
* sorted in descending order by the number of cycles executed in each *
* procedure; unexecuted procedures are excluded                    *
-------------------------------------------------------------------

  148137751 cycles  <-------  Total number of program cycles.

    cycles %cycles  cum %     cycles  bytes procedure (file)
                              /call   /line

  48071708   32.45  32.45         34     32  write_char (../textoutput.c)
  42443503   28.65  61.10   42443503     26  main (fixfont.p)
  26457936   17.86  78.96         30     44  eoln (../textinput.c)
  20662326   13.95  92.91         23     27  read_char (../textinput.c)
```

Figure 5.3: Profiler Listing for Procedures Based on Basic Blocks Counts

The circled text in Figure 5.3 shows:

a. The statistics describe calls to *eoln* compiled from the source file *textoutput.c*.

b. *eoln* used 26,457,936 cycles which represented 17.86% of the total program cycles.

c. The cumulative total of cycles used by *write_char, main* and *eoln* is 78.96%.

d. *eoln* used an average of 30 cycles per call and 44 bytes per line.

Procedures: - Basic Block Counts (with clock time)

Profiler option: -pixie -procedure -clock

```
--------------------------------------------------------------------------
*  -p[rocedures] using basic-block counts;                               *
*  sorted in descending order by the number of cycles executed in each   *
*  procedure; unexecuted procedures are excluded                         *
--------------------------------------------------------------------------

148[illegible] cycles (18.51?? seconds at 8.00 megahertz)

    cycles %cycles  cum %   seconds  cycles   bytes procedure (file)
                                      /call    line

  4[illegible]   32.45   32.45    6.0090      34    32 write_char (../textoutput.c)
  42443[illegible]   28.65   61.10    5.3054            26 main (fixfont.p)
  2[illegible]   17.86   78.96    3.3072      30    44 eoln (../textinput.c)
  2[illegible]   13.95   92.91    2.5828      23    27 read_char (../textinput.c)
   [illegible]    2.91   95.82    0.5385      62     8 write_chars (../textoutput.c)
   3678408    2.48   98.30    0.4598     133    14 write_integer (../textoutput.c)
   1573656    1.06   99.36    0.1967      29    16 write_string (../textoutput.c)
    362700    0.24   99.61    0.0453      26    67 readln (../textinput.c)
    279002    0.19   99.80    0.0349      20    30 writeln (../textoutput.c)
```

Figure 5.4: Profiler Listing for Procedures Based on Basic Blocks Counts (with clock times)

The listing in Figure 5.4 contains the same information as the listing shown in Figure 5.3, and contains the number of seconds spent in each procedure.

The circled text in the figure above shows that the profiler computes the time in seconds based on the machine speed specified in the *-clock* option.

Heavy - Basic Block Counts
Profiler option: -pixie -heavy

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

  procedure (file)                line bytes     cycles       %  cum%

  write_char (../textoutput.c)     120    88   28276478  19.09  19.09
  eoln (../textinput.c)             31   116   22808688  15.40  34.48
  main (fixfont.p)                  42    92   19069136  12.87  47.36
  read_char (../textinput.c)        59    56    9881982   6.67  54.03
  main (fixfont.p)                  43    40    8583512   5.79  59.82
  write_char (../textoutput.c)     105    20    7069725   4.77  64.59
  read_char (../textinput.c)        60    28    5390172   3.64  68.23
  main (fixfont.p)                  37    20    4489680   3.03  71.26
```

Figure 5.5: Profiler Listing for Heavy Line Usage

The circled text in the figure above shows:

a. Line 59, which is located in procedure *read_char* and compiled from source file *textoutput.c* is the fourth most heavily used line.

b. Line 59 has 56 bytes of code and used 9,881,982 cycles, or 6.67% of the total program cycles.

c. Lines 120, 31, 42 and 59 combined executed 54.03% of the total program cycles.

Lines - Basic Block Counts
Profiler option: - pixie - lines

```
--------------------------------------------------------------------------
*  -l[ines] using basic-block counts;                                    *
*  grouped by procedure, sorted by cycles executed per procedure;        *
*  '?' means that because a procedure was compiled without profiling,    *
*  we lack line number information for it                                *
--------------------------------------------------------------------------

procedure (file)                 line bytes     cycles %cycles

write_char (../textoutput.c)      105    20    7069725   4.77
                                  106     8    2827890   1.91
                                  111     8    2827890   1.91
                                  106     4    1413945   0.95
                                  112    16    1413945   0.95
                                  113    72          0   0.00
                                  115    12    4241835   2.86
                                  116    64          0   0.00
                                  117    28          0   0.00
                                  120    88   28276478  19.09
  main (fixfont.p)                 11    60         15   0.00
                                   12    32          8   0.00
                                   13    24          6   0.00
                                   14    24          6   0.00
                                   15     4          1   0.00
                                   16    40       8490   0.01
                                   17    24        166   0.00
```

Figure 5.6: Profiler Listing for Line Information

The circled text in the figure above shows:

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

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

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

How Basic Block Counting Works

To obtain basic block counting data:

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

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

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

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

   Pixie creates a program equivalent to *myprog* containing additional code that counts the execution of each basic block. *Pixie* also generates a file (*myprog.Addrs*) that contains the address of each of the basic blocks. For more information, see the *pixie*(1) manual page in the *RISC/os User's Reference Manual.*

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

   ```
   myprog.pixie
   ```

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

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

   ```
   prof -pixie myprog myprog.Addrs myprog.Counts
   ```

   Note: Specifying *myprog.Addrs* and *myprog.Counts* is optional; *pixie* searches by default for files with names of the form:

   program_name`.Addrs` and *program_name*`.Counts`.

   You can run the program several times, altering the input data, and create multiple profile data files. See Averaging Prof Results in this chapter.

The steps for obtaining basic block count information are shown in Figure 5.7.

You can include or exclude information on specific procedures within a program using the *-only* or *-exclude* options to *prof* (see Table 5.1). You can also run *pixstats* to generate a detailed report on opcode frequencies, interlocks, a mini profile, and more.

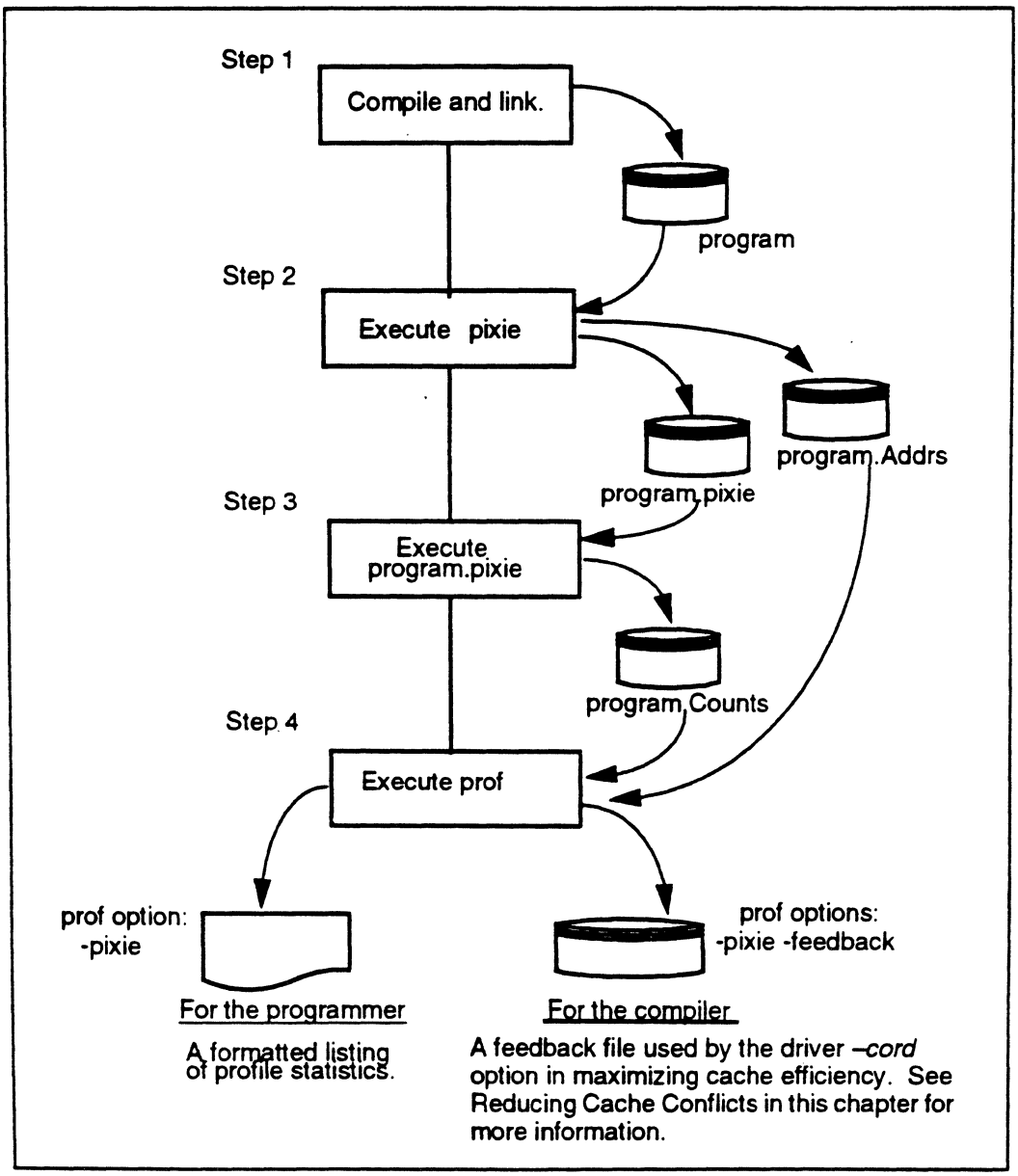


Figure 5.7: Obtaining Basic Block Count Information

Averaging Prof Results

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

1. Compile and link–edit; do *not* use the *—p* option:

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

2. Run the profiling program *pixie*, as follows:

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

This command produces the *myprog.Addrs* file to be used in Step 4, as well as the modified program *myprog.pixie*.

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

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

3. Run *prof* to create the report as follows:

```
prof -pixie myprog myprog.Addrs myprog[123].Counts
```

prof averages the basic block data in the *myprog1.Counts*, *myprog2.Counts*, and *myprog3.Counts* files to produce the profile report.

PC–Sampling

To obtain pc–sampling data on a program:

1. Compile and link–edit using the *–p* option, as follows:

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

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

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

   ```
   myprog
   ```

   You can run the program several times, altering the input data, and create multiple profile data files. See the section *Averaging Prof Results* in this chapter.

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

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

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

You can include or exclude information on specific procedures within your program by using the *–only* or *–exclude* profiler options (see Table 5.1).

Figure 5.8 shows the steps required to obtain pc sampling information.

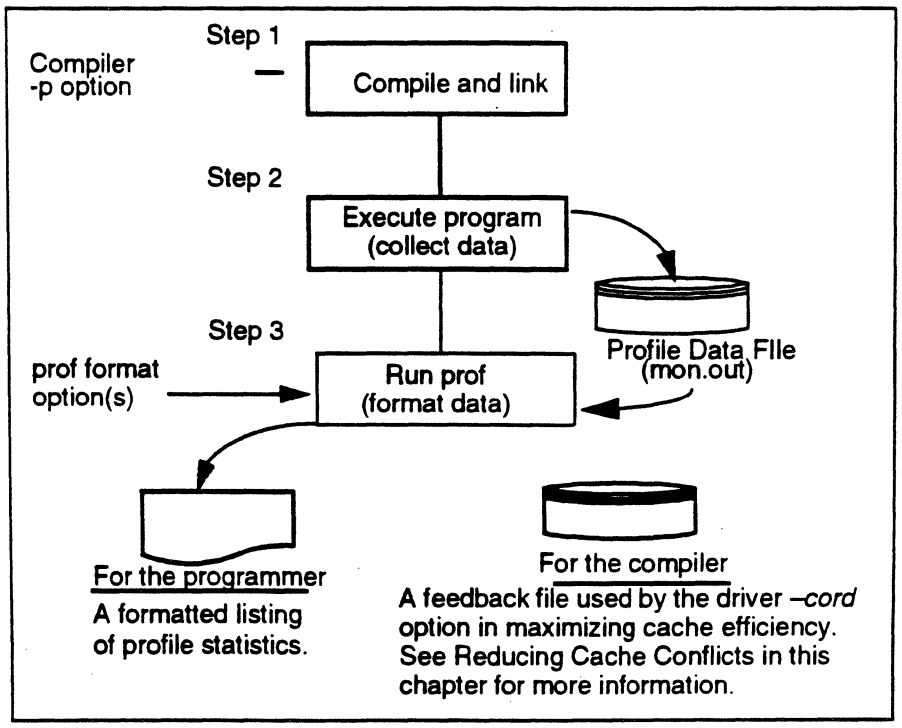


Figure 5.8: Obtaining PC Sampling Data

Creating Multiple Profile Data Files

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

C Shell

```
setenv PROFDIR string
```

Bourne Shell

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

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

Running the Profiler (prof)

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

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

where

options is one of the keyword or keyword abbreviations shown in Table 5.1. You can specify either the entire name or the initial character of the option.

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

profile_filename specifies one or more files containing the profile data gathered when the profiled program executed. If multiple files are specified, *prof* sums the statistics in the resulting profile listings.

pname.Addrs is produced by running *pixie* and *pname.Counts* is produced by running the *pixie*-modified version of the program.

The default for *profile_filename* is determined as follows:

- If you don't specify *profile_filename*, the profiler looks for the *mon.out* file; if this file doesn't exist, it looks for the profile input data file(s) in the directory specified by the PROFDIR environment variable (see the section Creating Multiple Profile Data Files).
- If you don't specify *profile_filename*, but do specify *-pixie*, then *prof* looks for *pname*.`Addrs` and *pname*.`Counts` and provides basic block count information if these files are present.

The *–merge* option can be used when you have multiple profile data files; this option merges the data into one file. See Table 5.1 for information on the *–merge* option.

Table 5.1: Options for the Profile List Program (prof), 1 of 3

Profile List Program (prof) Options	
Name	**Result**
-p[rocedures]	Displays the time spent in each procedure. See Figure 5.3 for an example of the output.
–pixie	*Basic block counting.* Indicates that information is to be generated on basic block counting, and that the *Addrs* and *Counts* file produced by *pixie* are to be used by default. See Figure 5.3 through 5.6 for examples of sample output.
-i[nvocations]	*Basic block counting.* Lists the number of times each procedure is invoked. The *–exclude* and *–only* options described below apply to called routines, but not to callers.
-l[ines]	See Figure 5.2 for sample output. *Basic block counting.* List statistics for each line of source code. See Figure 5.6 for sample output.
–o[nly] *proc_name*	Reports information on only the procedure specified by *procedure_name*, rather than on the entire program. You may specify more than one *–o* option. If you specify uppercase *–O*, prof uses only the named procedure(s), rather than the entire program, as the base upon which it calculates percentages.
-e[xclude] procedure_name	Excludes information on the procedure(s) (and their descendants) specified by *procedure_name*. If you specify uppercase *–E* for Exclude, *prof* also omits that procedure from the base upon which it calculates percentages. If you use one or more *–exclude* options, the profiler omits the specified procedure and its descendants from the listing.
-z[ero]	*Basic block counting.* Prints a list of procedures that are never invoked.

Table 5.1: Options for the Profile List Program (prof), 2 of 3

Profile List Program (prof) Options

Name	Result
-q[uit] n -q[uit] n% -q[uit] ncum%	Allows you to condense output listings by truncating unwanted lines. You can truncate by specifying *n* in one of three ways: **n** — n is an integer. All Lines after n line are truncated. **n%** — n is an integer followed by the percentage sign. All lines after the line containing n% calls in the %calls column are truncated. **ncum%** — n is an integer followed by the characters *cum* (for *cumulative*) and a percentage sign. All lines after the line containing ncum% calls in the cum% column are truncated. Below are three examples of using the **–q** option. Any one of the three specifications shown below would eliminate the items in the box below. -prof -q 4 -prof -q 13% -prof -q 92cum%

```
    calls  %calls    cum%
 48071708   32.45   32.45   6.0090
 42443503   28.65   61.10   5.3054
 26457936   17.86   78.96   3.3072
 20662326   13.95   92.91   2.5828
  4307932    2.91   95.82   0.5385
  3678408    2.48   98.30   0.4598
  1573858    1.06   99.36   0.1967
   362700    0.24   99.61   0.0453
   279002    0.19   99.80   0.0349
   251152    0.17   99.97   0.0314
    30283    0.02   99.99   0.0038
    13391    0.01  100.00   0.0017
     2923    0.00  100.00   0.0004
```

Table 5.1: Options for the Profile List Program (prof), 3 of 3

Profile List Program (prof) Options	
Name	**Result**
-h[eavy]	Basic block counting. Same as the *–lines* option, but sorts the lines by their frequency of use. See Figure 5.5 for a sample output listing.
-c[lock] n	Basic block counting. Lists the number of seconds spent in each routine, based on the CPU clock frequency *n*, expressed in megahertz; *n* defaults to 8.0 of omitted. Never use the default if the next argument *program_name* or *profile_name* begins with a digit. See Figure 5.4 for a sample output listing.
-t[estcoverage]	Basic block counting. Lists line numbers containing code that is never executed.
-m[erge] filename	This option is useful when multiple input files of profile data (normally in *mon.out*) are used. The option causes the profiler to merge the input files into filename, making it possible to specify the name of the merged file (instead of several file names) on subsequent profiler runs.
–f[eedback] *filename*	Produces a file used by the driver *–cord* option to maximize cache efficiency. See *Reducing Cache Conflicts* in this chapter for details.

Optimization

This section describes the compiler optimization tools and their benefits, the implications of optimizing and debugging, and the major optimizing techniques.

Global optimizer

The global optimizer is a single program that improves the performance of RISCompiler object programs by transforming existing code into more efficient coding sequences. Although the same optimizer processes

optimizations for all languages, it does distinguish between the various languages supported by the RISCompiler system to take advantage of the different language semantics involved.

The compiler system performs both machine-independent and machine dependent optimizations. RISComputers and other machines with RISC architectures provide a better target for machine dependent optimizations; the low-level instructions of RISC machines provide more optimization opportunities than the high-level instructions in other machines. Even optimizations that are machine-independent have been found to be effective on machines with RISC architectures. Although most of the optimizations performed by the global optimizer are machine independent, they have been specifically tailored to the RISC/os environment.

Benefits

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

Optimization and Debugging

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

Optimization and Bounds Checking

The compiler option –C, which performs bounds checking in Pascal and Fortran programs, inhibits some optimizations. Therefore, unless bounds checking is crucial, do not specify the –C option when optimizing a Pascal or Fortran program.

Loop Optimization

Optimizations are most useful in code that contain loops. The optimizer moves loop-invariant code sequences outside loops so that they are performed only once instead of multiple times. Apart from loop-invariant

code, loops often contain loop–induction expressions that can be replaced with simple increments. In programs composed of many loops, global optimization can often reduce the running time by half.

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

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

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

Figure 5.9 shows the unoptimized and optimized code produced by the compiler. Note that the optimized version contains fewer total instructions and fewer instructions that reference memory. Wherever possible, the optimizer replaces load and store instructions (which reference memory) with the faster computational instructions that perform operations only in registers.

Unoptimized:

loop is 13 instructions long using 8 memory references.

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

Optimized:

loop is 6 instructions long using 2 memory references.

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

Figure 5.9: Optimized and Unoptimized Code

Register Allocation

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

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

Optimizing Separate Compilation Units

The optimizer processes one procedure at a time. Large procedures offer more opportunities for optimization, since more inter-relationships are exposed in terms of constructs and regions. However, because of their size, large procedures require more time than smaller *-fleedback] filename* ones.

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

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

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

Optimization Options

Figure 5.10 shows the processing phases of the compiler and how the *-On* option determines the execution sequence. Table 5.2 summarizes the functions of each of the -O options.

Table 5.2: Optimizer Compiler Options

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

Note: You should refer to the *cc(1)*, *f77(1)*, or *pc(1)* manual page, as applicable, in the *User's Reference Manual* for details on the *–O3* option and the input and output files related to this option.

The optimizations performed under -O2 or -O3 rely to some extent on the global optimizer's own estimates of the execution frequencies of different parts of the program. In general, the optimizer assumes that loops are executed at least one order of magnitude more frequently than the adjacent code. The more deeply nested the code is, the more frequently it will be executed. At two-way branches that come from if-then-else constructs, the optimizer assumes that each branch has equal likelihood to be taken. Optimizations like register allocation and the inlining of procedure calls can yield better results if such estimates are more accurate. The -feedback compilation option is provided to let the optimizer take advantage of profile data generated by earlier runs of the program being optimized, and not rely on its own guesses as to the relative execution frequencies in different parts of the program.

The *-feedback* option takes the name of a profile data file as an argument. The profile data file is the binary form of the profile listing generated by *prof*. This file is generated if the -f option is given to *prof*. Alternatively, this profile data file can be generated by the feedback command, see *feedback(1)*.

It is best to generate the profile data file when the program is compiled with the -g option. Under the -g option, the profile information is accurate to within individual line numbers. Under -O1, -O2, and -O3 compilations, the compiler can move instructions across line boundaries, so that the execution time associated with individual lines may not be accurate. The degree to which the optimizer can make use of the profile data is also affected by how clearly the code is separated across lines. If a lot of code is packed into each line, or if the source program uses a lot of macros or conditional expressions, the effect of profile feedback may be diminished.

Some programs behave differently when given different data. For these programs, it is important that the run which generates the feedback file represents ordinary conditions and behavior. The user can combine the profile data from different runs so that the final profile data file represents the average program behavior.

If the user follows these guidelines, a program optimized with the -feedback option should always run at least as fast as the version compiled without this option. In most cases, the program should run faster, depending upon how much the run deviates from the compiler's own guess of execution frequencies in the absence of real profile data.

The -feedback option has no effect on the compilation if it is specified with the -O1 and -g options.

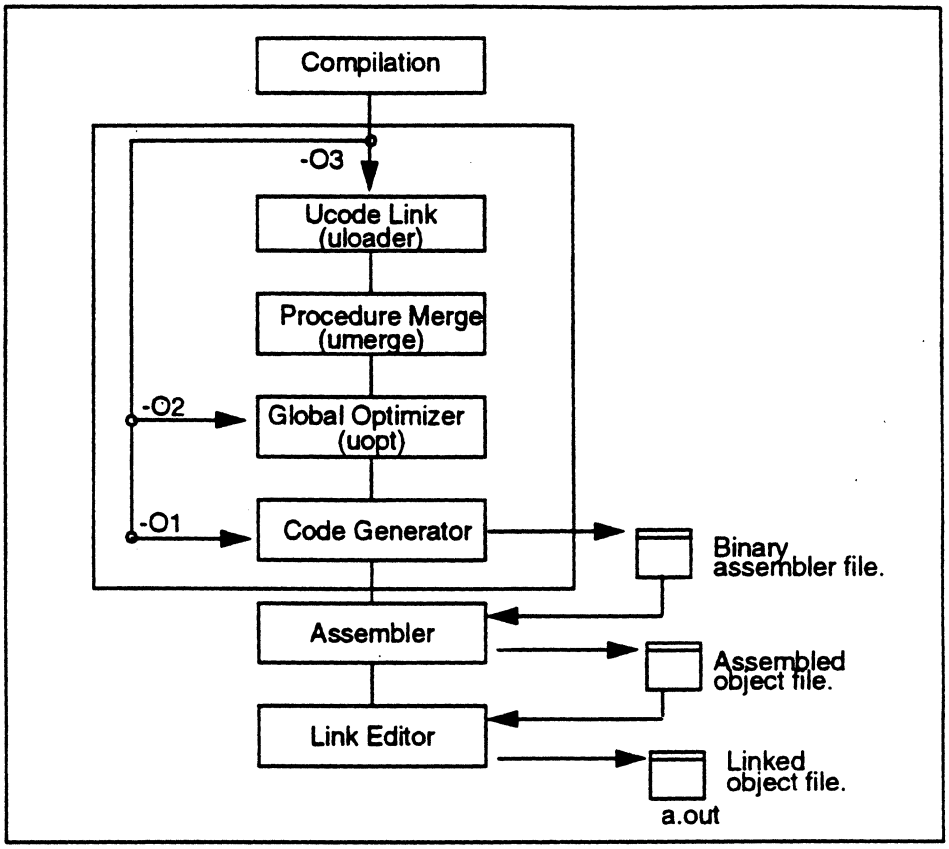


Figure 5.10: Optimization Phases of the Compiler

Full Optimization (–O3)

The following examples assume that the program *foo* consists of three files: *a.c*, *b.c*, and *c.c*.

To perform procedure merging optimizations (–O3) on all three files, enter the following command:

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

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

1. Compile each file separately using the –j option by entering the following commands:

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

The –j option causes the compiler driver to produce a .u file (the standard compiler front–end output, which contains ucode; ucode is an internal language used by the compiler). None of the remaining compiler phases are executed, as illustrated below. Figure 5.11 illustrates the results after execution of the three commands shown above.

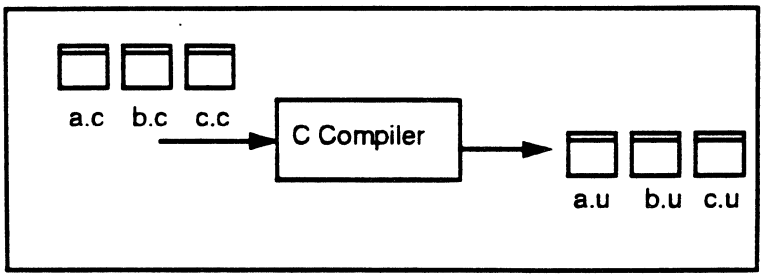


Figure 5.11: O3 Optimization

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

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

Figure 5.12 illustrates the results of executing the above command.

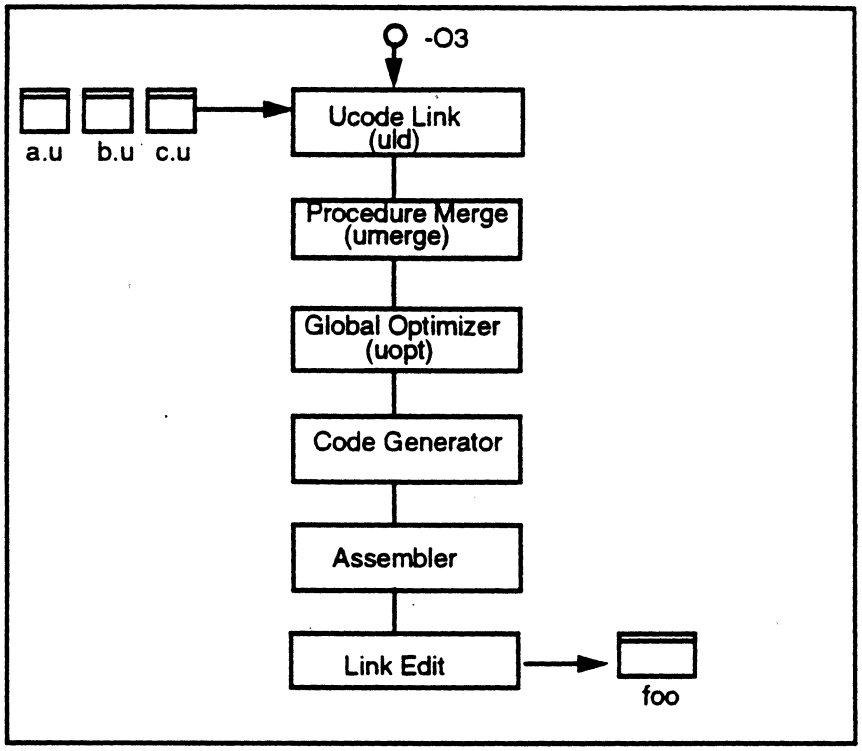


Figure 5.12: Compiler Phases of O3 Optimization

Optimizing Large Programs

To ensure that all program modules are optimized regardless of size, specify the –Olimit option at compilation time.

Because compilation time increases by the square of the program size, the RISCompiler system enforces a top limit on the size of a program that can be optimized. This limit was set for the convenience of users who place a higher priority on the compilation turnaround time than on optimizing an entire program. The –Olimit option removes the top limit and allows those users who don't mind a long compilation to fully optimize their programs.

Optimizing Frequently Used Modules

You may want to optimize modules that are frequently called from other programs. This can reduce the compile and optimization time required for programs calling these modules.

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

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

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

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

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

```
b.c  foo()
      {
      ••
      }
     bar()
      {
      ••
      }
         zot()
          {
          ••
          }
     struct
      {
      ••
      } work;
```

```
c.c    x()
        {
        ••
        }
              help()
               {
               ••
               }
        struct
         {
         ••
         } ddata;
        y()
         {
         ••
         }
```

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

```
foo bar x ddata y
```

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

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

3. Optimize the *b.u* and *c.u* modules using the *extern* file as follows:

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

The -kp option designates that the link editor option p is to be passed to the ucode loader.

Figure 5.13 illustrates Step 3.

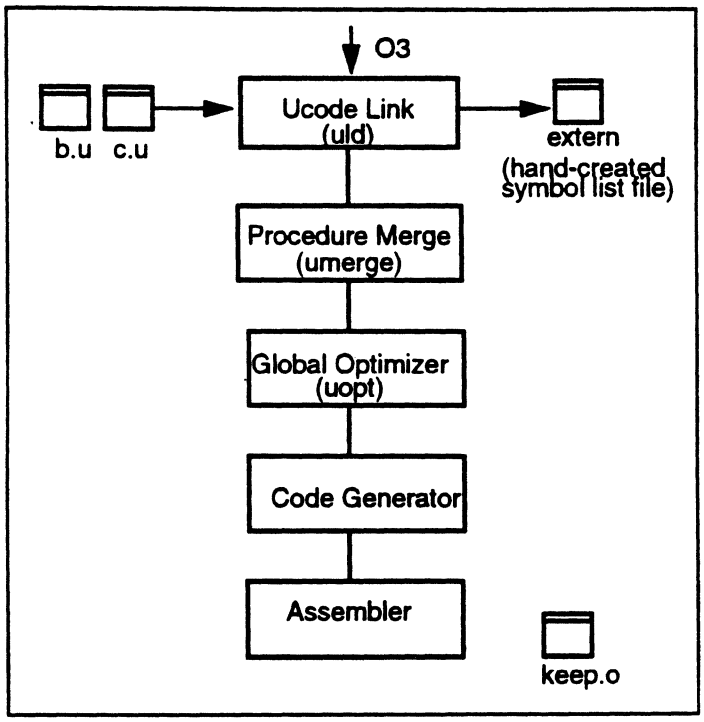


Figure 5.13: Optimizing Phases

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

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

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

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

Go to Step 5 if this message appears.

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

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

Building a Ucode Object Library

Building a ucode object library is similar to building a *coff* object library. First, compile the source files into ucode object files using the compiler driver option -j. To build a ucode library (lib*foo*.b) containing object files for *a.c*, *b.c*, and *c.c*, enter the following commands:

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

Ucode libraries should have names with *.b* as a suffix.

Using Ucode Object Libraries

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

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

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

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

Improving Global Optimization

This section contains coding hints to increase optimizing opportunities for the global optimizer (*uopt*).

C, Pascal, and FORTRAN Programs

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

C and Pascal Programs

Use functions to return values instead of reference parameters.

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

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

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

Use *value* parameters instead of *reference* parameters or *global* variables. *Reference* parameters have the same degrading effects as the use of pointers.

Aliases can often be avoided by introducing local variables to store dereferenced results. (A *dereferenced* result is the value obtained from a specified address.) Dereferenced values are affected by indirect operations and calls, whereas local variables are not; local variables can be kept in registers. Figure 5.14 shows how the proper placement of pointers and the elimination of aliasing lets the compiler produce better code.

Consider Figure 5.14, which uses pointers. Because the statement **p++=0* might modify *len*, the compiler, for optimal performance, cannot place it in a register, but instead must load it from memory on each pass through the loop.

Source Code:

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

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

Generated Assembly Code:

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

Figure 5.14: Pointers and Optimization

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

Using subscripts instead of pointers. The use of subscripting in the procedure *azero* eliminates aliasing; the compiler keeps the value of *len* in a register, saving two instructions, and still uses a pointer to access *a* efficiently, even though a pointer isn't specified in the source code (see Figure 5.15).

Source Code:

```
void
azero()
  {
  int i;
  for (i = 0; i != len; i++) a[i] = 0;
  }
```

Generated Assembly Code:

```
      for (i = 0; i != len; i++) a[i] = 0;
        move    $2, $0          # i = 0
        beq     $4, 0, $37      # len != 0
        la      $5, a
$36:
        sb      $0, 0($5)       # *a = 0
        addu    $2, $2, 1       # i++
        addu    $5, $5, 1       # a++
        bne     $2, $4, $36     # i != len
$37:
```

Figure 5.15: Using Subscripts instead of Pointers

Using local variables. Specifying *len* as a local variable or formal argument (as shown below) ensures that aliasing can't take place and permits the compiler to place *len* in a register (see Figure 5.16).

Source Code:

```
char a[10];
void
lpzero(len)
  int len;
  {
  char *p;
  for (p = a; p != a + len; ) *p++ = 0;
  }
```

Generated Assembly Code:

```
#  8  for (p = a; p != a + len; ) *p++ = 0;
      move          $2, $6          # p = a
      addu          $5, $6, $4
      beq           $5, $6, $33     # a + len != a
$32:
      sb            $0, 0($2)       # *p = 0
      addu          $2, $2, 1       # p++
      bne           $5, $2, $32     # a + len != p
$33:
```

Figure 5.16: Using Local Variables instead of Pointers

In Figure 5.16, the compiler generates slightly more efficient code for the second method.

Pascal Programs Only

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

C Programs Only

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

Bad	Good

`while (n--) {`
`  . . .`
`}`
	`while (n != 0 ) {`
	`  n--;`
	`  . . .`
	`}`

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

Avoid taking and passing addresses (& values). This can create aliases, make the optimizer store variables from registers to their home storage locations, and significantly reduce optimization opportunities.

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

Improving Other Optimization

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

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

C, Pascal, and FORTRAN Programs

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

  For example:

OK	More Efficient
`if ( i == 1) c = "1";` `else c = "0";`	`c = "01"[i];`

- As an optimizing technique, the compiler puts the first four parameters of a parameter list into registers where they remain during execution of the called routine. Therefore, you should always declare as the first four parameters those variables that are most frequently manipulated in the called routine with floating point parameters preceding non–floating point.
- Use word–size variables instead of smaller ones if space is not a consideration. This may use more space, but is more efficient.

C Programs Only

- Use libc functions (e.g. *strcpy, strlen, strcmp, bcopy, bzero, memset, memcpy*) instead of writing similar routines. These functions are hand–coded for efficiency.
- Use the **unsigned** data type for variables wherever possible for the following reasons: (1) because the variable is always greater than or equal to zero (>=0), the compiler can perform optimizations that would not otherwise be possible, and (2) the compiler generates fewer instructions for multiply and divide operations that use the power of two. Consider the following example:

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

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

```
000000    20010002  li     r1,2
000004    0081001a  div    r4,r1
000008    14200002  bne    r1,r0,0x14
00000c    00000000  nop
000010    03fe000d  break  1022
000014    00001812  mflo   r3
```

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

```
000018    0005c042  srl    r24,r5,1   # j / 2
```

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

Pascal Programs Only

Use predefined functions as much as possible. For example,

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

Limiting the Size of Global Data Area

The compiler places constants and variables in the *.lit8*, *.lit4*, *.sdata* and *.sbss* portions of the data and bss segments shown in Figure 5.17. This area is referred to as the *global data area*.

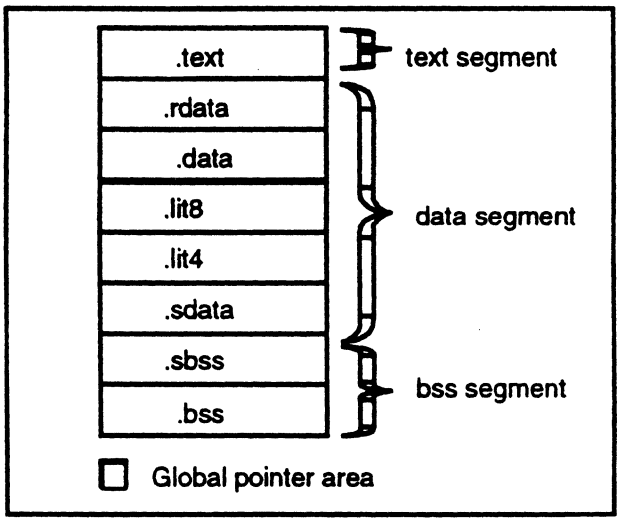


Figure 5.17: Global Data Area

(The *.rdata*, *.data*, *.lit8*, *.lit4*, and *.sdata* sections contain initialized data, and the *.sbss* and *.bss* sections reserve space for uninitialized data that is created by the kernel loader for the program before execution and filled with zeros. For more information on section data, see Chapter 9 of the *Assembly Language Programmer's Guide*.)

Purpose of Global Data

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

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

Controlling the Size of Global Data Area

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

```
-G 12
```

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

Obtaining Optimal Global Data Size

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

Thus, two potential problems exist in specifying a maximum size in the *–G* option:

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

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

Examples (Excluding Libraries)

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

For example:

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

If you specify the option as shown below:

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

the compiler produces a message giving the best value for *-G*; if all program data fits into the global data area, the following message is displayed:

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

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

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

The above example produces the following messages:

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

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

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

Example (Including Libraries)

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

```
cc -o plotter -bestGnum plotter.o -nocount libieee.a \
        -count liblaser.a
```

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

Reducing Cache Conflicts

RISComputer hardware provides two high-speed caches—one for program data and the other for instructions—that temporarily hold data or instructions frequently used by the processor. During execution, instructions or data from specified memory locations are placed in the cache. Because the cache is much smaller than memory, a single cache location is shared by many distinct memory locations. The first cache

location is shared by the 0th, 64KBth, 128KBth, ... memory locations. This mapping of every memory location to exactly one cache location is called a *direct mapped cache.*

A cache conflict occurs when a program references two instructions or data items that compete for the same location in the respective data or instruction cache. Normally this is not a problem. When the references are made repeatedly, as in a loop, such repeated conflicts can degrade performance.

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

You can eliminate major instruction cache misses within your programs by using the *–cord* driver option in combination with the *pixie* and *prof* programs. This option attempts to place the most frequently executed sections of code in memory so that they don't conflict with each other. To optimally reorganize the program *index.f*, execute the following commands:

```
% f77 -c -O index.f
% f77 -o index index.o
% pixie -o index.pixie index
% index.pixie
% prof index -feedback feedfile
% f77 -o index index.o -feedback feedfile -cord
```

Figure 5.18 illustrates the steps for the reorganization of program *index.f.*

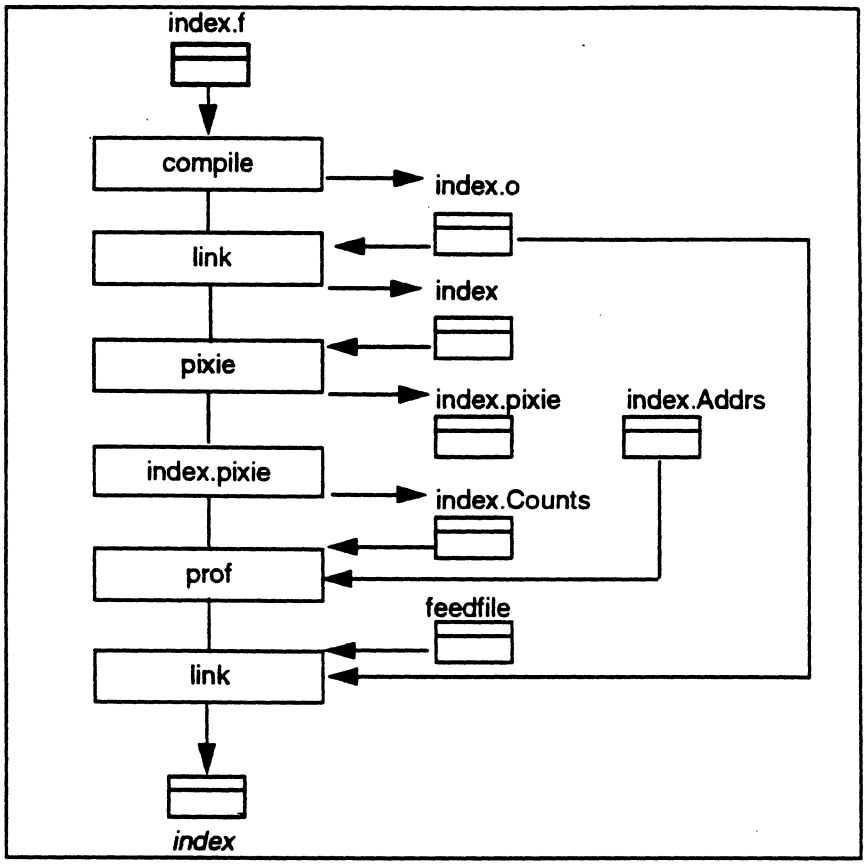


Figure 5.18: Using the –cord Option

For more information, see *prof(1)*, *pixie(1)*, or the *–cord* option in the applicable driver manual page – *cc(1)*, *pc(1)*, or *f77(1)*, in the *RISC/os User's Reference Manual*.

Filling Jump Delay Slots

In jump instructions, there is a jump delay or latency of one instruction, which is called a *jump delay slot*. Whenever possible, the compiler inserts an instruction in the delay slot to avoid stalls in the execution pipeline of instructions. (See delay slot in the *MIPS RISC Architecture* manual for a detailed discussion.) The *-jmpopt* option enables the compiler to fill additional delay slots at the cost of requiring more memory by the link editor. The default is *nojmpopt;* this option ensures that most link edits do not abort because of memory constraints.

For programs requiring high in performance, specify the *-jmpopt* option. Then, the link editor attempts to insert executable instructions into those delay slots that the compiler could not fill.

Debugging Programs

6

This chapter describes the source–level debugger *dbx* and tells how to use it. The debugger can be used with C, FORTRAN 77, Pascal, assembly language, and machine code. This chapter describes how to invoke *dbx* and all debugger commands, giving examples of each. The following topics are covered in this chapter:

Introduction

Introduces new users to the debugger and discusses general debugging issues, including where to start and how to isolate errors. It gives tips for users new to source–level debugging. Users familiar with debuggers may want to skip to the next section.

Running dbx

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

Using dbx Commands

Describes the *dbx* command syntax, expression precedence, data types, and constants, and lists the most common commands.

Working with the dbx Monitor

Describes how to use history, edit the command line, enter multiple commands, and use facilities that help you complete program symbol names.

Controlling dbx

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

Examining Source Programs

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

Controlling the Program

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

Setting Breakpoints

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

Examining Program State

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

Debugging at the Machine Level

Describes the commands used to debug machine code, including those to examine memory addresses and disassemble source code.

Introduction

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

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

Why Use a Source-Level Debugger?

dbx lets you trace problems in a program object at the source code, rather than at the machine code level. With *dbx*, you control a program's execution, monitoring program control flow, variables, and memory locations. You can also use *dbx* to trace the logic and flow of control to become familiar with a program written by someone else.

The advantages to using *dbx* include:

- Easy to use environment.
- High-Level language debugging.
- Remote debugging.
- Stack tracing.
- Single stepping.

- Expression evaluator.
- Assembly debugging.
- Breakpoints.
- Program state examination.
- Line-by-line variable tracing.

What Are Activation Levels?

Activation levels define the currently active scopes (usually procedures) on the stack. An activation stack is a list of calls that starts with the initial program (usually *main()*). The most recently called procedure or block is number 0. The next procedure called is number 1. The last activation level is always the main procedure (the procedure that controls the whole program).

Activation levels can also consist of blocks that define local variables within procedures. You see activation levels in stack traces (see the *where* command) and when moving around the activation stack (see the *up*, *down*, and *func* commands). Figure 6.1 shows the stack trace produced by a *where* command.

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

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

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

Figure 6.1: Stack Trace

Isolating Program Failures

dbx finds only runtime errors; you should fix compiler errors before starting a debugging session.

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

1. Invoke the program under *dbx*.
2. Get a stack trace using the *where* command to locate the point of failure.

Note: If you haven't stripped symbol table information from the program object, you can get a stack trace even if the program was not compiled with the *-g* debug flag.

3. Set breakpoints to isolate the error using *stop* commands.
4. Print the values of variables using the *print* command to see where a variable may have been assigned an incorrect value.

If you still cannot find the error, other *dbx* commands may be useful. Using dbx Commands in this chapter describes each *dbx* command.

Incorrect Output Results

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

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

Avoiding Pitfalls

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

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

```
print (output)
```

- If the debugger does not display any variables or executable code, make sure you compiled the program with the -g option.

Running dbx

Before invoking *dbx*, you need to compile the program for debugging. You may also want to create a *.dbxinit* file that will execute commands when the debugger is started.

Compiling a Program for Debugging

To use the debugger, specify the -g option at compilation time. This option inserts symbol table information in the program object, which *dbx* uses to list source lines.

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

You can do limited debugging on code compiled without the *-g* flag. For example, the following commands work without recompiling for debugging:

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

Although you can do limited debugging, it may be more useful to recompile the program with *-g*. The debugger does not warn you if an object file has been compiled without the *-g* flag.

Building a Command File

You can create a command file, called *.dbxinit*, that contains *dbx* commands, using a system editor. When *dbx* is invoked, the commands are executed (you are prompted for required input). A command file can be used to customize the *dbx* environment or to specify a set of frequently used *dbx* commands.

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

Figure 6.2 shows an example of a *.dbxinit* file:

```
set $page = 5
set $lines = 20
set $prompt = 177DBX>"
alias du dump
```

Figure 6.2: Sample .dbxinit file

Invoking dbx

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

Syntax:

Command	Function
`dbx [options] [objfile][corefile]`	Invoke *dbx* from the shell command line

If *objfile* is not specified, *dbx* uses *a.out* by default. If *corefile* is specified, *dbx* lists the point of program failure. For core files, you can get a stack trace and look at the code; however, you cannot run a program from a core file, for example, set breakpoints or continue.

The available *options* are shown in Table 6.1.

Table 6.1: dbx Options

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

Example:

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

Ending dbx (quit)

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

Syntax:

Command	Function
`quit`	End the debugging session
`q`	

Example:

```
(dbx)  quit
%
```

After entering *quit*, *dbx* prompts you to confirm that you want to exit.

Using dbx Commands

This section describes the conventions used for describing *dbx* command syntax, expressions and precedence, displaying data and constants, and some of the commonly used debugging commands.

Command Syntax

The following conventions are used in the command descriptions:

- Words in lower–case `typewriter` font are literals, and must be entered as they are shown.
- Words in *italics* indicate variable values that you specify.
- Square brackets ([]) surrounding an argument mean that the argument is optional.
- *dbx* variable names appear in *italics*.
- Words in upper–case `typewriter` font indicate variables for which specific rules apply. These words are given in Table 6.2.

dbx lets you enter up to 10240 characters on an input line. Long lines can be continued with a backslash (\). If a line gets too long, *dbx* prints an error message (see *fgets(1)* in the *User's Reference Manual*). The maximum string length is also 10240.

The following example command illustrates the syntax conventions:

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

Enter *stop*, *in*, and *if* as shown. Enter the values for *VAR*, *PROCEDURE* and *EXP* as defined in Table 6.2.

Table 6.2: Keywords Used in Command Syntax Descriptions

Keyword	Value
^ (caret)	Press the control key on your keyboard. Usually, used in conjuction with another key.
ADDRESS	Any expression specifying a machine address.
ARGS	Program arguments (maximum allowed by *dbx* is 1000; however, system limits may also appply.
COMMAND_LIST	One or more commands, each separated by semicolons.
DIR	A directory name.
FILE	File name.
EXP	Any express including program variable names for the command. Expressions can contain dbx variables; for example, ($listwindow+2). If youwant to use the words *in*, *to* or *at* in an expression, you must surround them with parentheses; otherwise, *dbx* assumes that these words are debugger key words.
INT	Integer value.
LINE	A souce code line number.
NAME	*dbx* command name.
PROCEDURE	Procedure name or an activation level on the stack.
REGEX	A regular expression string. See *regcmp*(3) in the *RISC/os Programmer's Reference Manual.*
SIGNAL	A RISC/os system signal. For BSD, see the *sigvec*(2) manual page in the Programmer's Reference Manual. For SysV, see the *signal*(2) manual page.
STRING	Any ASCII string.
VAR	Valid program variable or *dbx* predefined variable. For machine-level debugging, VAR can also be an address.

Qualifying Variable Names

Variables in *dbx* are qualified by file, procedure, block, or structure. When using commands like *print* to print a variable's value, *dbx* indicates the scope of the variable when the scope could be ambiguous (for example, you have a variable by the same name in different procedures). If scope is wrong, you can specify the full scope of the variable by separating scopes with periods. For example:

```
sam.main.i
```

where *sam* is the current file; *main* is the procedure; and *i* is the variable.

dbx Expressions and Precedence

dbx recognizes expression operators from C, Pascal, and FORTRAN 77. Operators follow the C language precedence (see Table 6.3).

Table 6.3: dbx Expression Operators

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

Use the # operator to convert line number into address.

Tables 6.4, 6.5, and 6.6 show language operators; note that // (instead of /) is used for divide.

Table 6.4: C Expression Operators

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

Note: The *sizeof* operator specifies the number of bytes retrieved to get an element, not (number_of_bits+7)/8.

Table 6.5: Pascal Expression Operators

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

Table 6.6: FORTRAN Expression Operators

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

Note: FORTRAN array subscripts use [] instead of ().

dbx Data Types and Constants

dbx commands can use the built-in data types described in Table 6.7.

Table 6.7: Built-in Data Types

Data Types	
Data Types	**Description**
$address	Pointer
$unsigned	Unsigned Integer
$char	Character
$boolean	Boolean
$real	Double Precision Real
$integer	Signed Integer
$float	Single Precision Real
$double	Double Precision Real
$uchar	Unsigned Character
$short	16-bit integer
$signed	Signed Integer
$void	

The built-in data types can be for type coercion – for example, to print a variable as a type that is different from its declaration.

The types of constants that are acceptable as input to *dbx* are shown in Table 6.8. Constants that are output from *dbx* are displayed by default as decimal values.

Table 6.8: Input Constant

Input Constants	
Constant	**Description**
false	0
true	nonzero
nil	0
0x *number*	hexadecimal
0t*number*	decimal
0*number*	octal
number	decimal
number.[*number*][eIE][+I-EXP]	float

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

The *$octin dbx* variable changes the default input expected to octal. The *$hexin* variable changes the default input expected to hexadecimal. See Predefined dbx Variables.

The *$octints dbx* variable changes the default output to octal. The *$hexints* variable changes the default output to hexadecimal. See Predefined dbx Variables.

Basic dbx Commands

dbx offers many commands; however, for most debugging sessions, the commands shown in Table 6.9 are sufficient.

Table 6.9: Commonly Used Debugger Commands

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

Working with the dbx Monitor

dbx provides a command history, command line editing, and symbol name completion. *dbx* also allows multiple commands on an input line. These features can reduce the amount of input required or allow you to repeat previously executed commands.

Using the Command History

The *dbx* command history allows you to re–execute debugger commands. The debugger keeps a list of previously executed commands that can be displayed with the *history* (alias *h*) command.

You can set the number of history lines saved using the *$lines* variable using the *set* command. The default is 20. See Setting dbx Variables.

To repeat a command, use one of the exclamation point (!) commands (see the syntax description for *history*).

Syntax:

Command	Function
history	Print the items in your history list.
!string	Repeat the most recent command that starts the specified string.
!INT	Repeat the command associated with the specified integer.
!-INT	Repeat the command that occurred the specified integer before the most recent command.

Example:

The following example prints the history list and then re-executes one of the commands:

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

Editing the dbx Command Line

dbx provides commands that permit command line editing. These commands allow you to correct mistakes without re-entering an entire command. The editing commands are the same as those used for *csh* command line editing. See *csh(1)* in the *RISC/os User's Reference Manual* for a description of the editing commands. Table 6.10 shows some of the commonly used editing commands.

Table 6.10: dbx Command Line Editing Commands

DBX Command Line Editing	
Command	**Function**
carriage return	Repeat the last command issued to dbx. This feature is turned off by setting the $repreatmode variable to 0. See Setting dbx Variables.
^A	Move the cursor to the beginning of the command line.
^B	Move the cursor back one character.
^D	Delete the character at the cursor.
^E	Move the cursor to the end of the line.
^F	Move the cursor forward one character.
^H, DELETE	Delete the character immediately preceding the cursor.
^N	Move forward one line. (This line comes from the history list.)
^P	Move back on line. (This line comes from the history list.)

Note: In Table 6.10, the notation ^ represents the CTRL key. For example ^A indicates that the CTRL and A keys should be pressed simultaneously.

Entering Multiple Commands

You can enter multiple commands on the command line by using a semicolon (;) as a separator. This can be useful when using the *when* command. See Writing Conditional Code in dbx.

Syntax:

Command	Function
`COMMAND; COMMAND`	Enter mulptiple commands on the command line.

Example:

The following example stops the program and then re–runs it.

```
(dbx) stop at 58; rerun
[1] stop at 58 177sam.c":58
[1] stopped at [printline:58,0x2f8] pline->string
(dbx)
```

Completing Symbol Names

dbx provides symbol name completion; *dbx* completes names from a unique prefix when the partial name is followed by CTRL-Z. If a unique completion is found, *dbx* redisplays the input with it added; otherwise, all possible completions are shown and you can choose one.

Syntax:

Command	Function
`STRING ^Z`	Complete a symbol name or see what symbol names contain the specified string

Example:

The following example displays all names beginning with the letter i.

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

Note: The display may include data types and library symbols.

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

dbx completes the symbol name for you

Controlling dbx

dbx provides commands to set and unset *dbx* variables, create and remove aliases, record and play back input, invoke a shell from *dbx*, and check and delete items from the status.

Setting dbx Variables

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

You cannot define a debugger variable with the same name as a program variable. The *print* command displays the values of variables. The *dbx* predefined variables are listed in Table 6.12.

Syntax:

Command	Function
`set`	Display a list of dbx predefined variables.
`set VAR = EXP`	Assign a new value to a variable or define a new variable.

Example:

The following example lists all debugger variables, changes one, and then redisplays the list.

```
(dbx) set

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

$listwindow      15 <-------------- new value
$datacache        1
$main         "main"
$pagewindow      22
$page             1
$maxstrlen      128
$cursrcline      24
more (no?) no
(dbx)
```

Removing Variables

Use the *unset* command to remove a *dbx* variable. To see a full list of *dbx* variables, use the *set* command.

Syntax:

Command	Function
`unset VAR = EXP`	Unset the value of a dbx variable.

Example:

The following example assigns a value to a new variable and then removes it using the *unset* command.

```
(dbx) set $test = 5
(dbx) set

$listwindow      10
$datacache        1
$main          "main"
$pagewindow      22
$test             5          <-- new variable on list
$maxstrlen      128
$cursrcline      24
more (no?) no
(dbx) unset $test
(dbx) set

$listwindow      15
$datacache        1
$main          "main"
$pagewindow      22
$maxstrlen      128          <-- new variable removed from list
$cursrcline      24
more (no?) no
(dbx)
```

Predefined dbx Variables

The predefined *dbx* variables are shown in Table 6.12. The variables that are preset, but which you can change, are indicated by *I, B,* or *S* notations in the Key column. Variables that only *dbx* can set, but are available for information, are indicated by an *R*.

Table 6.11 summarizes the notations in the Key column of Table 6.12.

Table 6.11: Key Notations for Predefined Variables

Key	Description
I	Integer
B	Boolean
S	ASCII character string
R	Reset exclusively and periodically by the debugger

Table 6.12: Predefined dbx Variables, 1 of 4

Debugger Variables

Key	Variable	Default	Description
S	*$addrfrmt*	"0x%x"	Specifies the format for addresses. This can be set to anything you can format with a C language *printf* statement.
S	*$byteaccess*		Same as *$addrfrmt*.
B	*$casesense*	0	Specifies whether source searching and variables are case sensitive. A nonzero value means case insensitive; a 1 means case sensitive.
IR	*$curevent*	none	Shows the last event number as reported by the status command.
IR	*$curline*	none	Shows the current line in the source code.
IR	*$clusrcline*	none	Shows the last line listed plus 1.
IR	*$curpc*		Shows the current address. Used with the *wi* and *li* aliases.
B	*$datacache*	*1*	Caches information from the data space so that *dbx* only has to check the data space once. If you are debugging the operating system, set this variable to 0; otherwise, set it to a nonzero value.
	$debugflag	0	An internal debug flag used to debug *dbx*.
SR	*$defaultout*	""	Shows the name of the file that *dbx* uses to store information when using the *record output* command.
SR	*$defaultin*	""	Shows the name of the file that *dbx* uses to store information when using the *record input* command.
	$defin *$defout* *$dispix*		Used internally by *dbx*.
B	*$hexchars*	0	Displayed values are shown in hexadecimal when $hexchars is set to a nonzero value; a nonzero value overrides octal.
B	$hexin	0	A nonzero value indicates that input constants are hexadecimal.

Table 6.12 Predefined dbx Variables, 2 of 4

Debugger Variables			
Key	**Variable**	**Default**	**Description**
B	*$hexints*	0	Used to determine the default setting of printing a char*. A 0 will cause output to be the address and string content. A 1 will print only the address in hex value.
B	*$hexstrings*	0	A nonzero value indicates that strings are displayed in hexadecimal; otherwise strings are shown as characters.
IR	*$historyevent*	none	Shows the current history number.
I	*$lines*	20	Specifies the size of *dbx* history list.
I	*$listwindow*	TERM/2	Specifies the number of lines shown by the list command.
S	*$main*	"main"	Specifies the name of the procedure where execution begins. *dbx* starts the program at main() unless otherwise specified.
I	*$maxstrlen*	128	Specifies the number of characters of a string *dbx* prints for pointers to strings. dbx checks mutliples of 4 to see if it exceeds the maximum.
B	*$octints*	0	Changes the default output constants to octal when set to a nonzero value. Hexadecimal overrides octal.
B	*$octin*	0	Changes the default input constants to octal when set to a nonzero value. Hexadecimal overrides octal.
B	*$page*	1	Specifies whether to page long information. A nonzero value turns on paging; a 0 turns it off.
I	*$pagewindow*	22	Specifies the number of lines displayed when viewing information that is longer than one screen. This variable should be set to the number of lines on the terminal. A value of 0 indicates a minimum of 1 line.
S	*$pdbxport*		Port name from */etc/remote[.pdbx]* used to connect to target machine for pdbx.
B	*$printwide*	0	Specifies wide (useful for structures or arrays) or vertical format for printing variables. A nonzero value indicates wide format; 0 indicates vertical.

Table 6.12 Predefined dbx Variables, 3 of 3

Debugger Variables			
Key	**Variable**	**Default**	**Description**
B	$*printwhilestep*	0	For use with the *step[n]* and *stepi[n]* instructions. A nonzero value specifies that all n line and/or instructions should be printed. A 0 value specifies that only the last line and/or instruction should be printed.
B	$*readtextfile*	1	When set to 1, *dbx* tries to read instructions from the object file rather than the process. This variable should always be set to 0 when the process being debugged copies in code during the debugging process.
S	$*prompt*	"dbx"	Sets the prompt for *dbx*.
B	$*regstyle*	1	Specifies the type of register names to be used. A value of 1 specifies hardware names; a 0 specifies software names as defined by the file *regdefs.h*. This variable does not affect coprocessor register names.
B	$*repeatmode*	1	Specifies whether *dbx* should repeat the last command when a carriage return is pressed. A nonzero value indicates that the command is repeated; otherwise it is not repeated.
B	$*rimode*	0	Records input when using the record output command.
S	$*sigtramp*	sigtramp	Tells *dbx* the name of the code called by the system to invoke user signal handlers.
B	$*stop_in_main*	0	Tells *dbx* to stop at *main*() when set to 1. When set to 0, tells *dbx* to debug the dynamic linking process at start up time.
S	$*tagfile*		Contains a filename indicating the file in which the tag command and the tabvalue macro are to search for tags.
B E	$*use_rld_symbols*	0	When set to 1, tells *dbx* to use *rld* symbols in precedence of user symbols; this is useful in debuggind *rld* (runtime linker), which may have collisions with user symbols.

Creating Command Aliases (alias)

The *alias* command defines a new alias or displays a list of all current aliases.

The *alias* command allows you to rename any debugger command. Enclose commands containing spaces within double or single quotation marks. You can also define a macro as part of an alias.

dbx has a group of predefined aliases; you can modify these or add to the list. Aliases can also be included in the *.dbxinit* file to use them in future debugging sections.

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

Syntax:

Command	Function
`alias`	Displays a list of all aliases.
`alias NAME1 [(ARG ...ARGN)] "NAME2"`	Defines a new alias. NAME1 is the new name. NAME2 is the command to rename. ARG1...ARGN are the command arguments.

Example:

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

Removing Command Aliases (unalias)

The *unalias* command removes an alias from a command. You must specify the alias to remove; otherwise, a syntax error is displayed. The alias is removed only for the current debugging session.

Syntax:

Command	Function
`unalias "name"`	Remove an alias from a command, where name is the alias name.

Example:

The following example displays all the aliases and removes the history alias.

```
(dbx) alias
h       history
si      stepi
Si      nexti
ni      nexti
pi      playback input
ro      record output
ri      record input
a       assign
t       where
j       status
bp      stop in
b       stop at
g       goto
s       step
More (n if no)?n
(dbx) unalias h
(dbx) alias
 si      stepi
 Si      nexti
 ni      nexti
 pi      playback input
 ro      record output
 ri      record input
 a       assign
 t       where
 j       status
 bp      stop in
 b       stop at
 g       goto
 s       step
 More (n if no)?n
 (dbx)
```

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

Predefined dbx Aliases

To list current aliases, use the *alias* command. You can override any predefined alias by redefining it with the *alias* command or by removing it from the list with the *unalias* command. Table 6.13 shows the debugger predefined aliases.

Table 6.13: Debugger Aliases

Debugger Aliases		
Alias	**Command**	**Function**
a	assign	Assign a value to a program variable.
b	stop at	Set a breakpoint at a specified line.
bp	stop in	Stop in a specified procedure.
c	continue	Continue program execution after a breakpoint.
d	delete	Delete the specified item from the status list.
e	file	Look at the specified source file.
f	func	Move to the specified activation level on the stack.
g	goto	Go to the specified line and begin executing the program there.
h	history	List all items currently on the history list.
j	status	Display the items on the status list.
l	list	List the next 10 lines of source code.
n or S	next	Step over the specified number of lines without stepping into procedure calls.
ni or SI	nexti	Set over the specified number of assembly code instructions without stepping into procedure calls.
p	print	Print the value of the specified expression or variable.
pd	printf"%d\n"	Print the value of the specified expression or variable in decimal.
pi	playback input	Replay *dbx* commands saved with the record input command.
po	printf"%o\n:	Print the value of the specified expression or variable in octal.
pr	printregs	Print values for all registers.
px	printf"%x\n"	Print the value of the specified expression or variable in hexadecimal.
q	quit	End the debugging session.
r	rerun	Run the program again with the same arguments specified with the run command.
ri	record input	Record every command entered in a file.
ro	record output	Record all debugger output in the specified file.
s	step	Step the next number of specified lines.
si	stepi	Step the specified number of assembly code instructions.
t	where	Get a stack trace.
u	list $curlin-15:10	List the previous 10 lines.
w	list $curlin-10:20	List the 10 lines preceding and following the current line.
wi		List the 5 machine instruction preceding and following the machine instruction.

Recording Input

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

Syntax:

Command	Function
`record input [filename]`	Record all *dbx* commands in a file.

dbx saves the recorded input in *filename*. If *filename* is omitted, *dbx* saves the recorded input in a temporary file, which is deleted at the end of the *dbx* session. The name of the temporary file is in the system variable *$defaultin;* to display the temporary filename, use the *print* command:

```
print $defaultin
```

Use the temporary file to repeat previously executed *dbx* commands only in the current debugging session; specify *filename* to create a command file for use in subsequent *dbx* sessions. The *status* command indicates whether *record input* is set. Use the *delete* command to stop *record input*.

Example:

The following example records input and displays the resulting file.

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

The temporary file from the above *dbx* commands is as follows:

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

Recording Output (record output)

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

Syntax:

Command	Function
`record output [filename]`	Record all *dbx* commands in a file.

dbx saves the recorded output in *filename*. If *filename* is omitted, *dbx* saves the recorded output in a temporary file, which is deleted at the end of the *dbx* session. The name of the temporary file is in the system variable *$defaultout*; to display the temporary filename, use the *print* command:

```
print $defaultout
```

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

The *status* command indicates whether *record output* is set. Use the *delete* command to stop *record output*.

Example:

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

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

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

Playing Back Input

Use these commands to replay the commands recorded with the *record input* command. If a filename is not specified, *dbx* uses the current temporary file that it created for the *record input* command. If the *dbx* variable *$pimode* is set to 1, the commands are printed as they are played back.

Syntax:

Command	Function
`playback input [filename]`	Execute the commands from the specified file
`source [FILE]`	

Example:

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

Playing Back Output

This command displays output saved with the *record output* command. The *playback output* command works the same as the *cat* command. If *filename* is not specified, *dbx* uses the current temporary file created for the *record output* command.

Syntax:

Command	Function
`playback output [filename]`	Print the commands from the specified file.

Example:

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

Invoking a Shell from dbx

To invoke a subshell, enter *sh* at the *dbx* prompt, or enter *sh* and a shell command. To return to *dbx* from a subshell, enter *exit* or press ^D.

Syntax:

Command	Function
`sh`	Invoke a shell from *dbx*.
`sh [SHELL COMMAND]`	Execute the shell command.

Example:

```
(dbx) sh                          <-- invokes a shell
%
% date
Tue Apr 8 17:25:15 PST 1986
% exit
(dbx) sh date                     <-- invoke a shell and execute the date command
Tue Apr 8 17:29:34 PST 1986
(dbx)
```

Checking Shared Objects in Shared Environment

Use listobj to check what objects are linked in shared situations. *dbx* will display the object names and text address ranges.

Syntax:

Command	Function
`listobj`	Check which objects are linked.

Checking the Status (status)

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

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

Syntax:

Command	Function
`status`	Check the status of commands.

Example:

```
(dbx) status
[4] trace i in printline
[3] print pline^ at 177sam.c":58
[2] stop in printline
[1] record output /tmp/dbxt0018898 (0 lines)
(dbx)
```

the status item number

Deleting Status Items

Use the *delete* command to remove items from the status list. This command is used to delete breakpoints.

Syntax:

Command	Function
`delete EXP1,...EXPN`	Delete the specified status item (EXP) from the status list.
`delete all`	Delete all status items.

Example:

```
(dbx) status
[4] trace i in printline
[3] print pline^ at 177sam.c":58
[2] stop in printline
[1] record output /tmp/dbxt0018898 (0 lines)
(dbx) delete 4
(dbx) status
[3] print pline at "sam.c":58
[2] stop in printline
[1] record output /tmp/dbxt0018890 (0lines)
(dbx)
```

the status item number

Examining Source Programs

This section describes how to list and edit source code, change directories, change source files, search for strings in source code, print symbol names, and print variable declarations.

Specifying Source Directories

If *-I* was not specified when invoking the debugger, *dbx* looks for source files in the current directory or in the object file's directory. The *use* command changes the directory and lists the directories currently in use. The command recognizes absolute and relative pathnames (for example, *./*); however, it doesn't recognize the C shell tilde (~).

Syntax:

Command	Function
`use`	List the current directories.
`use DIR1 ... DIRN`	Specify different directories.

Example:

The following example changes the directory searched for files to */usr/local/lib*.

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

Moving to a Specified Procedure

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

The *func* command changes the current line, the current file, and the current procedure. This changes the scope of the variables you can access. The *func* command can be used when a program isn't executing to examine source code.

Syntax:

Command	Function
`func`	Print the current activation levels.
`func PROCEDURE`	Move to the activation level specified by the procedure name.
`func ECP`	Move the to activation level specified by the expression.

Example:

The following example shows a stack trace and moves to the main procedure.

```
(dbx) where
> 0 printline [pline = 0x7fff5b80) [177sam.c177:58,0x2f7]
  1 $block1 [177sam.c":47, 0x2bb]
  2 main(argc=2, argv=0x7fffeba0)["sam.c":47,0x2bb]
(dbx) func 2
main 47          printline(&line1)
(dbx) func main
(dbx)
```

the activation level

the procedure name

the procedure's arguments

the source file name

the current line

the current program counter

Specifying Source Files

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

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

Syntax:

Command	Function
`file`	Print the name of the file currently in use.
`file FILE`	Change the current file to the specified file.

Example:

```
(dbx) file
sam.c          <-------  current file
(dbx) file data.c
(dbx) file
data.c         <-------  new file
(dbx)
```

Listing 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 current file, procedure, and line unless otherwise specified. It moves the current line forward.

Syntax:

Command	Function
`list`	List lines for *$listwindow* lines starting at the current line.
`list EXP`	List the specified line.
`list EXP:INT`	List the specified number of lines (INT), starting at the specified line (EXP).
`list PROCEDURE`	List the specified procedure for *$listwindow* lines.

Example:

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

If you use the predefined alias **w**, (see **Predefined dbx Aliases**), the output is as follows:

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

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

Searching Through Code

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

Syntax:

Command	Function
`/REGEX`	Search forward in the code for the specified regular expression.
`?REGEX`	Search backward in the code for the specified regular expression.

Example:

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

Calling an Editor from dbx (edit)

The *edit* command lets you make changes to source code from within *dbx*. For the changes to become effective, you must exit *dbx*, recompile the program, and, to continue debugging, restart *dbx*.

Syntax:

Command	Function
`edit`	Invoke an editor from *dbx* on the current file.
`edit` [*filename*]	Invoke an editor on the specified file.

The *edit* command loads the editor indicated by the environment variable EDITOR. If EDITOR is not set, the *vi* editor is used. To return to *dbx*, exit the editor.

Printing Qualified Variable Names

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

Syntax:

Command	Function
`which VAR`	Print the default version of the variable.
`whereis VAR`	Print all versions of the specified variable.

Example:

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

Printing Type Declarations

The *whatis* command lists the type declaration for variables and procedures in a program.

Syntax:

Command	Function
`whatis VAR`	Print the type declaration for the specified variable or procedure.

Example:

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

Controlling the Program

This section describes the *dbx* commands to run a program, step through source code, return from a procedure call, start at a specified line, continue after stopping at a breakpoint, and assign values to program variables.

Running the Program

The *run* and *rerun* commands start program execution. Each command accepts program arguments. If arguments are not specified for the *run* or *rerun* command, the last set of arguments is used.

These commands can also be used to redirect program input and output in a manner similar to redirection in the C shell. The optional parameter <FILE1 redirects input to the program from the specified file. >FILE2 redirects output from the program to the specified file. The optional parameter >&FILE2 redirects *stderr* and *stdout* output to the specified file.

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

Syntax:

Command	Function
`run [ARG1,...ARGN][<FILE1][>FILE2]` `run [ARG1,...ARGN][<FILE1][>&FILE2]`	Run the program with the specified arguments.
`rerun [ARG1...ARGN][<FILE1][>FILE2]` `rerun [ARG1...ARGN][<FILE1][>&FILE2]`	Rerun the program with the previously specified arguments or with new arguments.

Example:

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

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

Executing Single Lines of Code

The *step* and *next* commands execute a fixed number of source code lines as specified by EXP. If EXP is not specified for *step* and *next*, *dbx* executes one source code line; otherwise, *dbx* executes the source code lines as follows:

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

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

Syntax:

Command		Function
`step [EXP]`	*	Execute the specified number of lines of source code. EXP refers to the number of lines to be executed in *both* the current procedure and any called procedures.
`next [EXP]`	*	Execute the specified number of lines of source code. EXP refers to the number of lines to be executed in *only* the current procedure, regardless of any called procedures executed.

* Default is 1.

Example:

The following example shows the use of the *step* command.

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

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

Returning from a Procedure Call

The *return* command is used in a called procedure to execute the remaining instructions in the procedure and stop at the first instruction on return from that procedure.

Syntax:

Command	Function
`return`	Execute the current procedure and return to the next sequential line in the calling procedure.
`return PROCEDURE`	Execute the program until *dbx* returns to the specified procedure.

Example:

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

Starting at a Specified Line

The *goto* command shifts program execution to the specified line. This command is useful in a *when* statement – for example, to skip a line known to cause problems.

Syntax:

Command	Function
`goto LINE`	Go to a specified line and continue execution.

Example:

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

Continuing after a Breakpoint

The *cont* command resumes program execution after a breakpoint. If SIGNAL is specified as a parameter (see below), *dbx* sends the specified signal to the program and continues.

Syntax:

Command	Function
`cont`	Continue from the current line.
`cont to LINE`	Continue until the specified line.
`cont in PROCEDURE`	Continue until the specified procedure.
`cont SIGNAL`	Continue from the current lineand send the signal.
`cont SIGNAL to LINE`	Continue until reaching the specified line and send the signal.
`cont SIGNAL in PROCEDURE`	Continue until reaching the specified procedure and send the signal.

Example:

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

Assigning Values to Program Variables

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

Syntax:

Command	Function
`assign EXP1 = EXP2`	Assign a new value to a program variable.

Example:

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

Setting Breakpoints

A breakpoint stops program execution and lets you examine the program's state at that point. This section describes the *dbx* commands to set a breakpoint at a specific line or in a procedure, and stop for signals.

Overview

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

```
[2] stopped at [ main:23,0x1cc] if(argc < 2) {
(dbx)
```

source line

the current program counter (use this number to print the assembly language instructions from this point (see Debugging at the Machine Level).

line number

breakpoint status number

procedure name

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

To select the right procedure, follow these steps:

1. Use the *func* command and specify a procedure name. This command changes the activation level to the specified procedure. See Controlling the Program.
2. List the lines of the procedure using the *list* command. See Controlling the Program.
3. Use a *stop* command to set a breakpoint at the desired line.

Setting Breakpoints at Lines

The *stop at* command sets a breakpoint at a specific line. *dbx* stops only at lines that have executable code. If you specify an unexecutable line, *dbx* sets the breakpoint at the next executable line. If you specify the VAR parameter, the debugger prints the variable and stops only when VAR changes; if you specify *if EXP*, *dbx* stops only when *EXP* is true.

Note: The *delete* command is used to remove breakpoints.

Syntax:

Command	Function
`stop [VAR] at`	Stop at the current line.
`stop [VAR] at LINE`	Stop at a specified line.
`stop [VAR] at LINE if EXP`	Stop at a specified line only if the expression is true.

Note: *if EXP* is checked before *VAR*.

Example:

```
(dbx) stop at 58
[16] stop at 177sam.c":58
(dbx) rerun
[16] stopped at [printline:58,0x2f8] pline->string);
(dbx)
```

the procedure name

the line number

the current program counter

Setting Breakpoints in Procedures

The *stop in* command sets a breakpoint at the beginning or, conditionally, for the duration of a procedure.

Syntax:

Command	Function
`stop in PROCEDURE`	Stop at the beginning of the procedure.
`stop VAR in PROCEDURE`	Stop in the specified procedure when VAR changes.
`stop in PROCEDURE if EXP`	Stop in the specified procedure if EXP is true.
`stop VAR in PROCEDURE if EXP*`	Stop in the specified procedure when VAR changes and EXP is true.

Note: *EXP* is checked before *VAR*.

Specifying both *VAR* and *EXP* causes stops *anywhere* in the procedure, not just at the beginning. Using this feature is time consuming, because the debugger must check the condition before and after each source line is executed.

Example:

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

the procedure name

the line number

the current program counter

Setting Conditional Breakpoints

The *stop if* command causes *dbx* to stop program execution under specified conditions. Because *dbx* must check the condition after the execution of each line, this command slows program execution markedly. Whenever possible, use *stop at* or *stop in* instead of *stop if*.

Syntax:

Command	Function
`stop if EXP`	Stop if EXP is true.
`stop VAR if EXP`*	Stop if VAR changes *and* EXP is true.

* EXP is checked before VAR.

Tracing Variables

The *trace* commands list the value of a variable during program execution as well as determine the scope for the variables being traced.

Syntax:

Command	Function
`trace VAR`	List the specified variable after each source line is executed.
`trace VAR at line`	List the specified variable at the specified line.
`trace VAR in PROCEDURE`	List the specified variable in the specified procedure.
`trace VAR at line if EXP`	List the variable at the specified line when the expression is true.
`trace VAR in PROCEDURE if EXP`	List the variable in the specified procedure when the expression is true.

Note: *EXP* is checked before *VAR*.

Example:

```
(dbx) trace i
[15] trace i in $block1
(dbx) rerun
[printline:58,0x2f8]:i=19
[23] [printline:58,0x2f8] pline->string);
  0 ( 19) #include<stdio.h>
[25] i changed before [177sam.c":41]:
              old value = 19;
              new value = 1;
[25] i changed before [177sam.c":41]:
              old value = 1;
              new value = 14;
[printline:58,0x2f8]: i=14
[23] [printline:58,0x2f8]   pline->string);
  1.   ( 14) struct line {
[25] i changed before [177sam.c":41]:
              old value = 14;
              new value = 22;
More (n if no)n
Escape from listing
(dbx)
```

Writing Conditional Code in dbx

The *when* command allows debugger commands to be executed under specified conditions.

Syntax:

Command	Function
`when VAR [if EXP] {COMMAND_LIST}`	Execute the command list when VAR changes.
`when [VAR] at LINE [if EXP] {COMMAND_LIST}`	Execute the command list when VAR changes, EXP is true, and the debugger encounters LINE.
`when in PROCEDURE {COMMAND_LIST}`	Execute the command list upon entering PROCEDURE.
`when [VAR] in PROCEDURE [if EXP] {COMMAND_LIST}`	Execute the specified commands on each line of PROCEDURE when EXP is true and VAR changes.

Note: *EXP* is checked before *VAR*.

Example:

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

dbx **stops in the procedure printline**

Stopping at Signals

The *catch* command lists the signals that *dbx* catches or specifies a signal for *dbx* to catch. If a child in the program encounters a specified signal, *dbx* stops the process.

Syntax:

Command	Function
`catch`	Print a list of all signals that dbx catches.
`catch SIGNAL`	Add a signal to the catch list.
`ignore`	Print a list of all signals that dbx does not catch.
`ignore SIGNAL`	Remove a signal from the catch list and add it to the ignore list.

Example:

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

Examining Program State

When *dbx* is stopped at a breakpoint, the program state can be examined to determine what may have gone wrong. There are *dbx* commands for printing stack traces, variable values, and register values. *dbx* also provides commands to display information about the activation levels shown in the stack trace and move up and down the activation levels.

Stack Traces

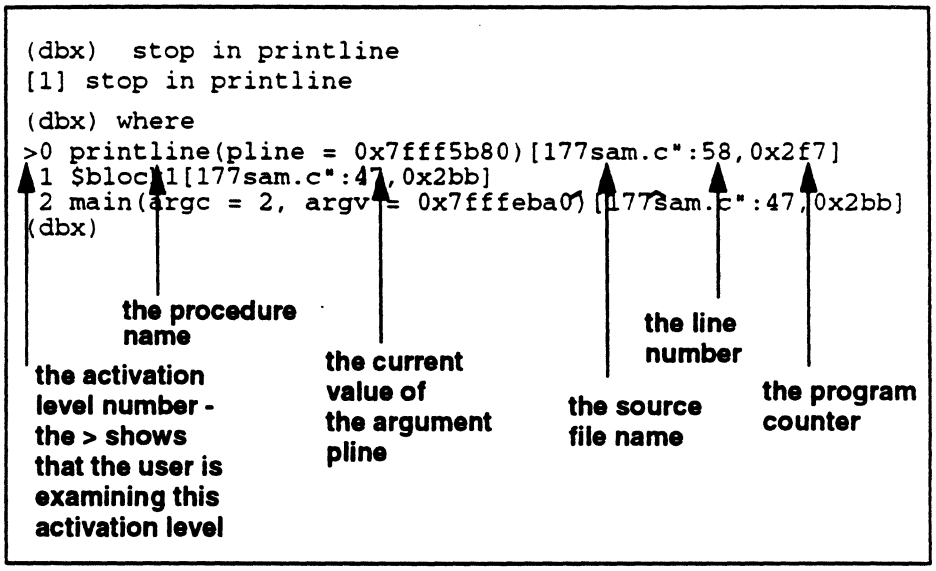
The *where* command display a stack trace. A stack trace shows the current activation levels (procedures) of a program.

Syntax:

Command	Function
`where [EXP]`	Display the stack trace.

Example:

If a breakpoint is set in *printline* in the sample program *sam.c*, (see Sample Program at the end of this chapter), the program runs and stops in the procedure *main*(). If you enter *where*, a stack trace is printed, providing the information shown below.



Note: In the example, *$block1* has the same program counter as *main*. This indicates that *main*() has a block with local variables, which do not appear to all of *main*().

Changing Activation Level

The *up* and *down* commands move up and down the activation levels in the stack. These commands are useful when examining a call from one level to another. You can also move up and down the activation stack with the *func* command. For a definition of activation levels, see What Are Activation Levels?

Syntax:

Command	Function
`up [EXP]`	Move up the specified number of activation levels in the stack. The default is one level.
`down [EXP]`	Move down the specified number of activation levels in the stack. The default is one level.

Example:

```
(dbx) where
>0 printline(pline = 0x7fff5b80)[177sam.c":58,0x2f7]
 1 $block1[177sam.c":47,0x2bb]
 2 main(argc = 2, argv = 0x7fffeba0)[177sam.c":47,0x2bb]
(dbx) down                                          moves
$block1 [177sam.c":47,0x2bb]                        down one level
(dbx) where
 0 printline(pline = 0x7fff5b80)[177sam.c":58,0x2f7]
>1 $block1[177sam.c":47,0x2bb]
 2 main(argc = 2, argv = 0x7fffeba0)[177sam.c":47,0x2bb]
(dbx) up                                            moves up
printline(pline = 0x7fff5b80)[177sam.c":58,0x2f7]   one level
(dbx) where
>0 printline(pline = 0x7fff5b80)[177sam.c":58,0x2f7]
 1 $block1[177sam.c":47,0x2bb]
 2 main(argc = 2, argv = 0x7fffeba0)[177sam.:47,0x2bb]
(dbx)
```

Printing

The *print* commands displays the value of one or more expressions. You can also use *print* to display the program counter and the current value of registers; see the next section, Printing Register Variables, for details.

The *printf* command lists information in a specified format and supports all formats of the *printf*(3S) command except %s. For a full list of formats, see the *printf*(3S) manual page in the *Programmer's Reference Manual*. *printf* can be used to see a variable's value in a different number base. The command alias list has some useful aliases for printing the value of variables in different bases – octal (*po*), decimal (*pd*), and hexadecimal (*px*). The default number base is decimal. See Creating Command Aliases.

Syntax:

Command	Function
`up [EXP]`	Move up the specified number of activation levels in the stack. The default is one level.
`down [EXP]`	Move down the specified number of activation levels in the stack. The default is one level.

Note: If the expression contains a name the same as a *dbx* keyword, it must be enclosed within parentheses. For example, in order to print *output*, a keyword in the *playback* and *record* commands, specify:

```
print (output)
```

Example:

```
(dbx) print i
14 <-------------- decimal
(dbx) pd i
14 <-------------- decimal
(dbx) po i
016 <------------- octal
(dbx) px i
0xe <------------- hexadecimal
(dbx)
```

Printing Register Values

The *printregs* command prints register values, both the real machine register names and the software (from the include file *regdefs.h*) names. A prefix before the register number specifies the type of register; the prefixes used and their meanings are as follows:

Prefix	Register Type
`$r`	Machine register.
`$f`	Floating point.
`$d`	Double precision floating point.
`$pc`	Program counter value.

You can also specify prefixed registers in the print command to display a register value or the program counter. The following commands print the values of machine register 3 and the program counter:

```
print $r3
print $pc
```

Set the *dbx* variable *$hexints* to specify that the display be in hexadecimal.

Syntax:

Command	Function
`printregs`	Print the current values of all registers.

Example

```
(dbx) printregs
r0/zero=0          r1/at=1            r2/v0=19           r3/v1=0
r4/a0=2147441472 r5/a1=34838          r6/a2=4096         r7/a3=80
r8/t0=19           r9/t1=34816        r10/t2=19          r11/t3=0
r12/t4=1           r13/t5=34820       r14/t6=0           r15/t7=1
r16/s0=2147441472r17/s1=0             r18/s2=0           r19/s3=0
r20/s4=0           r21/s5=0           r22/s6=0           r23/s7=0
r24/t8=4086        r25/t9=255         r26/k0=0           r27/k1=0
r28/gp=50529   r29/s0=2147441400r30/fp=2147442536 r31/ra=700
$f0= 0.0           $f1= 0.0           $f2= 0.0           $f3= 0.0
$f4= 0.0           $f5= 0.0           $f6= 0.0           $f7= 0.0
$f8= 0.0           $f9= 0.0           $f10=0.0           $f11=0.0
$f12=0.0           $f13=0.0           $f14=0.0           $f15=0.0
$f16=0.0           $f17=0.0           $f18=0.0           $f19=0.0
$f20=0.0           $f21=0.0           $f22=0.0           $f23=0.0
$f24=0.0           $f25=0.0           $f26=0.0           $f27=0.0
$f28=0.0           $f29=0.0           $f30=0.0           $f31=0.0
$d0= 0.0           $d2= 0.0           $d4= 0.0           $d6= 0.0
$d8= 0.0           $d10=0.0           $d12=0.0           $d14=0.0
$d16=0.0           $d18=0.0           $d20=0.0           $d22=0.0
$d24=0.0           $d26=0.0           $d28=0.0           $d30=0.0
$pc= 760
(dbx)
```

Printing Information about Activation Levels

The *dump* command prints information about activation levels, including values for all variables local to a specified activation level. To see what activation levels are currently active in the program, use the *where* command to get a stack trace.

Syntax:

Command	Function
dump	Print information about the current activation level.
dump .	Print information about all activation levels in the program.
dump PROCEDURE	Print information about the specified procedure (activation level).

Example:

```
(dbx) where
>0 printline (pline=0x7fff5b80)[177sam.c":58,0x2f7]
 1 $block1 [177sam.c":47,0x2bb]
(dbx) dump
printline (pline=0x7fff5b80)[177sam.c":58,0x2f7]
(dbx) dump .
> 0 printline(pline-0x7fff5b80)[177sam.c":58,0x2f7]
    1 $block1 [177sam.c":47,0x2bb]
 curlinenumber = 1
i=19
    2 main (argc=2,argv=0x7fffeba0)[177sam.c":47,0x2bb]
fd = 0x4270
line1=struct {
string=177#include<stdio.h>
"
linenumber=0
}
in "";
(dbx) dump main
main (argc=2, argv=0x7fffeba0)[177saam.c":47,0x2bb]
fd=0x4270
line1=struct {
string="struct line {
length = 14
linenumber = 1
}
(dbx)
```

Debugging Machine Code

This section describes the *dbx* commands provided for debugging assembly code; these commands allows you to set breakpoints, step through instructions, trace variables, display the contents of memory addresses, and disassemble instructions.

Setting Breakpoints in Machine Code

The *stopi* commands set breakpoints in machine code. These commands work in the same way as the *stop at, stop in,* and *stop if* commands as described in the section Setting Breakpoints, except for the *stop at* command, where an address instead of a line number is specified.

Command	Function
`stopi [VAR] at`	Stop at the current address.
`stopi [VAR] at ADDRESS`	Stop at a specified address.
`stopi [VAR] at ADDRESS if EXP`	Stop at a specified address only if EXP is true.
`stopi if EXP`	Stop if EXP is true.
`stopi VAR if EXP`	Stop if VAR changes and EXP is true.
`stopi in PROCEDURE`	Stop at the beginning of the procedure.
`stopi VAR in PROCEDURE`	Stop in the specified procedure when VAR changes.
`stopi in PROCEDURE if EXP`	Stop in the specified procedure ifEXP is true.
`stopi VAR in PROCEDURE if EXP*`	Stop in the specified procedure when VAR changes and EXP is true.

*EXP is checked before VAR.

Example:

```
(dbx) stopi at 0x2f8
[2] stopi at 177sam.c":760
(dbx) rerun
[2] stopped at [printline:58,0x2f8]pline-> string);
(dbx)
```

Continuing after Breakpoints in Machine Code

The *conti* commands continue executing assembly code after a breakpoint.

Syntax:

Command	Function
`conti SIGNAL`	Send the specified signal and continue.
`conti to ADDRESS`	Continue until reaching the specified address.
`conti in PROCEDURE`	Continue until the beginning of the specified procedure.
`conti SIGNAL to ADDRESS`	Continue until reaching the specified address, then send the signal.
`conti SIGNAL in PROCEDURE`	Continue until reaching the beginning of the specified procedure, then send signal.

Example:

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

Executing Single Lines of Machine Code

The *stepi* and *nexti* commands execute a fixed number of machine instructions as specified by EXP. If EXP is not specified, *dbx* executes one machine instruction. If EXP is specified, *dbx* executes the machine instructions as follows:

- *dbx* does not take comment lines into consideration in interpreting EXP. The program executes EXP machine instructions, regardless of the number of comment lines interspersed among them.
- For *stepi*, *dbx* considers EXP to apply to *both* the current procedure *and* to procedure calls (*jal* and *jalr*). The program stops after executing EXP instructions in the current procedure and any called procedures.
- For *nexti*, *dbx* considers EXP to apply to *only* the current procedure. The program stops after executing EXP instructions in the current procedure, regardless of the number of instructions executed in any procedure calls.

Syntax:

Command	Function
`stepi [EXP] *`	Execute the specified number of lines of machine code. EXP refers to the number of lines to be executed in *both* the current procedure *and* any procedure calls.
`nexti [EXP] *`	Execute the specified number of lines of machine code. EXP refers to the number of lines to be executed in *only* the current procedure, regardless of any procedure calls.
* Default is 1.	

Example:

```
(dbx) rerun
[2] stopped at [printline:58,0x2f8]pline->string);
(dbx) stepi
[printline:58+0x4,0x2fc]  pline->string);
lui  r1,0x0
(dbx)
```

Tracing Variables in Machine Code

The *tracei* commands track, one instruction at a time, changes to variables. The *tracei* commands work for machine instruction as the *trace* commands do for lines of source code.

Syntax:

Command	Function
`tracei`	Print the value of the variable as it changes.
`tracei VAR at ADDRESS`	Print the value of the variable when it changes at the specified address.
`tracei VAR in PROCEDURE`	Print the value of the variable when it changes in the specified procedure.
tracei VAR at ADDRESS if EXP	Print the value of the variable at the specified address when the expression is true.
tracei VAR in PROCEDURE if EXP	Print the value of the variable in the specified procedure when the expression is true.

Printing the Contents of Memory

Memory contents can be displayed by specifying the address and the format of the display. *address* is the address of the first item to be displayed, *count* is the number of items to be shown, and *mode* indicates the format in which the items are displayed. The values for *mode* are shown in Table 6.14.

Syntax:

Command	Function
`ADDRESS/<COUNT><MODE>`	Print the contents of the specified address for the specified count.

Table 6.14: Table 6.14 Modes for Printing Memory Addresses

Mode	Print Format
d	Print a short word in decimal.
D	Print a long word in decimal.
o	Print a short word in octal.
O	Print a long word in Octal.
x	Print a short word in hexadecimal.
X	Print a long word in hexadecimal.
b	Print a byte as a character.
s	Print a string of characters that ends in a null byte.
f	Print a single precision real number.
g	Print a double precision real number.
i	Print machine instructions.

Example:

The following example shows the output when printing memory addresses as instructions:

```
(dbx) 0x2f8/10i
[printline:58,0x2f8] lw    r2,32(sp)
[printline:58,0x2fc] lui   r1,0x0
[printline:58,0x300] addiu r4,r1,16860
[printline:58,0x304] lui   r1,0x0
[printline:58,0x308] addiu r5,r1,16780
[printline:58,0x30c] lw          r6,260(r2)
[printline:58,0x310] lw    r7,256(r2)
[printline:58,0x314] jal   fprintf!!
[printline:58,0x318] sw    r2,16(sp)
[printline:59,0x31c] lui   r1,0x0
[printline:59,0x320] jal   fflush<!
[printline:59,0x324] addiu r4,r1,16960
(dbx) 0x2f8/10d
   000002f8: 32 3677 0 0 15361 1690 9252 0 15361
   00000308: 16780 9253
(dbx)
```

Debugger Command Summary

Table 6.15 lists all commands (except for command line editing commands) and gives the syntax for each.

Table 6.15: Command Summary, 1 of 7

Command	Alias	Function	Syntax
/		Search forward in the code for the specified string.	/REGEX
?		Search backward in the code for the specified string.	?REGEX
!		Execute a command from the history list.	!STRING !INT !-INT
alias		List all aliases, or if an argument is specified, define a new alias.	alias [NAME(ARG1,... ARGN)"STRING"]
assign	a	Assign the specified expression to a specified program variable.	assign EXP1 = EXP2
catch		List all signals that *dbx* catches, or if an argument is specified, add the signal to the catch list.	catch [signal]
cont	c	Continue executing a program after a breakpoint.	cont cont in PROCEDURE cont to LINE cont SIGNAL to LINE cont SIGNAL in PROCEDURE

Table 6.15 Command Summary, 2 of 7

Command	Alias	Function	Syntax
conti		Continue executing assembly code after a breakpoint.	conti SIGNAL conti to ADDRESS conti in PROCEDURE conti SIGNAL to ADDRESS conti SIGNAL in PROCEDURE
delete	d	Delete the specified item from the status list.	delete EXP1,...EXPN delete ALL
down		Move down the specified number of activation levels in the stack. The default is one level.	down [EXP]
dump		Print variable information about the procedure. If a dot (.) is specified, information for all global variables is shown.	dump PROCEDURE dump .
edit		Invoke and editor from *dbx*.	edit [FILE]
file	e	Print the name of the current file, or if a filename is specified, change the current file to the specified file.	file [FILE]
func	f	Move to the specified procedure (activation level) or print the current activation level.	func func EXP func PROCEDURE
goto	g	Go to the specified line.	goto LINE

Table 6.15 Command Summary, 3 of 7

Command	Alias	Function	Syntax
help	?	Print a list of *dbx* commands using *more(1).*	help
history	h	Print a list of previously issued commands. The default list length is 20.	history
ignore		List all signals that *dbx* does not catch, or if an argument is specified, add the specified signal to the ignore list.	ignore [SIGNAL]
list	l	List the specified lines. The default is 10 lines.	list list [EXP:INT] list [EXP]
next	n	Step over the specified number of lines. The default is one. This command does not step into procedures.	next [INT]
nexti	ni	Step over the specified number of machine instructions. The default is 1. This command does not step into procedures.	nexti [INT]
playback input	pi	Replay commands saved with the *record input* command.	playback input [FILE]

Table 6.15 Command Summary, 4 of 7

Command	Alias	Function	Syntax
playback output	po	Replay debugger output saved with the *record output* command.	playback output [FILE]
print	p	Print the value of the specified expression.	print EXP1,...,EXPN
printf	pd	Print the value of the specified expression, using C string formatting.	printf 177STRING", EXP1,...EXPN
printregs	pr	Print all register values.	printregs
quit	q	Exit *dbx*.	quit
record input	ri	Record all commands entered to *dbx*.	record input [FILE]
record output	ro	Record all *dbx* output.	record output [FILE]
return		Continue executing until the procedure returns. If you don't specify a procedure, DBX assumes the next procedure.	return [PROCEDURE]
run		Run the program.	run [ARG1 ... ARGN] [<FILE1][>FILE2]
rerun	r	Run the program again using the arguments specified to the *run* command.	rerun [ARG1 ... ARGN] [<FILE1][>FILE2]

Table 6.15 Command Summary, 5 of 7

Command	Alias	Function	Syntax
set		Display the list of debugger variables and values, assign a value to a variable, or define a new variable and assign a value to it.	set set VAR = EXP
sh		Invoke a shell from *dbx*, or execute a shell command.	sh [SHELL COMMAND]
source		Execute *dbx* commands from the specified file. If a filename is not specified, the file created with the *record input* command is used.	source [FILE]
status	j	Print a list of currently set breakpoints, record commands, and traces.	status
step	s	Step the specified number of lines. This command steps into procedures. The default is one line.	step [INT]
stepi	si	Step the specified number of instructions. This command steps into procedures. The default is one instruction.	stepi [INT]
stop	b bp	Set a breakpoint at the specified location.	stop [VAR] at stop [VAR] at LINE stop [VAR] in PROCEDURE stop [VAR] if EXP stop [VAR] at LINE if EXP stop [VAR] in PROCEDURE if EXP

Table 6.15 Command Summary, 6 of 7

Command	Alias	Function	Syntax
stopi		Set a breakpoint in machine code at the specified point.	stopi [VAR] at ADDRESS stopi [VAR] in PROCEDURE stopi [VAR] if EXP stopi [VAR] at ADDRESS if EXP stopi [VAR] in PROCEDURE if EXP
trace	tr	Trace the specified variable.	trace VAR trace VAR at LINE trace VAR in PROCEDURE trace VAR at LINE if EXP trace VAR in PROCEDURE if EXP
tracei		Trace the specified variable in the machine instruction.	tracei VAR tracei VAR at ADDRESS tracei VAR in PROCEDURE tracei VAR at ADDRESS if EXP tracei VAR IN PROCEDURE if EXP
unalias		Remove specified alias.	unalias ALIAS NAME
unset		Unset a debugger variable.	unset VAR
up		Move the specified number of activation levels up the stack. The default is 1.	up [EXP]
use		Print a list of directories which are searched for files. If one or more directory names are specified, change the list of directories to those specified.	use [DIR1 DIR2...DIRN]

Table 6.15 Command Summary, 7 of 7

Command	Alias	Function	Syntax
whatis		Print the type declaration for the specified name.	whatis VAR
when		Execute the specified *dbx* commands under specified conditions.	when [VAR][if EXP] {COMMAND_LIST} when [VAR] at LINE [if EXP]{COMMAND_LIST} when [VAR] in PROCEDURE [if EXP]{COMMAND_LIST}
where	t	Get a stack trace.	where
whereis		Print all qualifications of the specified variable name.	whereis VAR
which		Print the qualification of the variable name currently in use.	which VAR
		Print the contents of the specified address in the format specified by MODE.	ADDRESS/<COUNT><MODE>

Sample Program

The sample C program referred to in command examples, *sam.c*, is shown in Figure 6.3.

```
#include <stdio.h>

struct line {
        char string[256];
        int length;
        int linenumber;
};
```

```
typedef struct line LINETYPE;
void printline();

main(argc, argv)
int argc;
char **argv;
{
   LINETYPE line1;
   FILE *fd;
   extern FILE *fopen();
   extern char *fgets();

   if (argc < 2) {
         fprintf(stderr, "Usage sam filename\n");
         exit(1);
   }
   fd = fopen(argv[1], "r");
   if (fd == NULL) {
         fprintf(stderr, "cannot open %s\n",
                  argv[1]);
         exit(1);
   }

   while(fgets(line1.string, sizeof(line1.string), fd)
         != NULL) {
         int i;
         static curlinenumber = 0;

         i = strlen(line1.string);
         if (i == 1 && line1.string[0] == '\n')
                  continue;
         line1.length = i;
         line1.linenumber = curlinenumber++;
         printline(&line1);
   }
}

void printline(pline)
LINETYPE *pline;
{
   fprintf(stdout, "%3d.   (%3d) %s",
         pline->linenumber,
         pline->length,
         pline->string);
   fflush(stdout);
}
```

Figure 6.3: Sample Program sam.c

MIPS C Implementation

7

Introduction

The MIPS C compiler supports four variations of the C language:

- C as defined in *The C Programming Language* by Kernighan and Ritchie (Prentice Hall, 1978) with some ANSI C extensions (also known as MIPS–C)
- ANSI C as defined in ANSI X3.159–1989 (American National Standards Institute, 1989), this document is referred to by section numbers, e.g. 3.2.2
- ANSI C with extensions
- An older version of MIPS C known as oldc

These variations of C are available with the following *cc* options:

–std0	MIPS C
–std1	strict ANSI C
–std	ANSI C with extensions
–oldc	old version of MIPS C, uses the old *cpp* and *ccom*. instead of the new *cfe*. Oldc will not be supported in future releases of MIPS RISCompilers.

Note: The compiler that comes with RISC/os supports *–std0* mode only. The ANSI C compiler supports all modes and defaults to *–std*.

This chapter covers the following topics:

- Additional options for the C driver.
- Translation limits.
- MIPS C extensions to C as defined in *The C Programming Language* by Kernighan and Ritchie (Prentice Hall, 1978).

- Compatibility issues between previous versions of MIPS–C (referred to as OldC) and ANSI C, ANSI C with extensions.

The ANSI C Language and extensions to ANSI C are described in Chapter 8 of this manual.

Additional Driver Options

In addition to the options discussed in Chapter 1 of this manual, the C driver, *cc*, has options that let you increase the amount of space allowed for various structures used by the compiler. These options are of the form *–Wf, –XNz<number>*, where z is one of the following:

Table 7.1: Additional Driver Options for -oldc only

option	meaning	default
a	temporary string space	1024
b	temporary string space	4096
c	temporary string buffer	40
d	symbol table	3000
e	nesting levels	100
f	parameter stack space	1020
g	switch table space	500
h	tree space	100
i	delayed tree space	20
j	hash table space	20
k	file name space	100
l	string literal space	2048
m	initialization stack space	10
n	line length	515
o	file stack size	1024
p	dimension table size	4200
q	block nesting size	100

If more than one of these options is used, each must be of the form *–Wf, –XNz<number>*. These options are only useful with the *–oldc* flag.

ccom options

The *ccom* (invoked by the driver to compile C sources) options are shown in Table 7.2. The options may appear on the command line in any order and have the form *–Xoption*.

Table 7.2: ccom options for -oldc only, 1 of 2

option	meaning
volatile	makes all variable declarations volatile
varargs	prints warning message if address of parameter is taken in a non-varargs function
v	verbose, prints out names of functions processed
signed	makes `char' same as `signed char'
float	use single prexision math where possible
framepointer	generate a framepointer in each function
W	test at the top for `while' loops
F	test at the top for `for' loops
Sfile	write symbol table to file
c	print warning message on pointer casts
dollar	allow `$' in identifiers
d	print debug info on defid and non-unique member references, multiple *-Xd*'s may be specified, each one yields more verbose output
i	print debug info on initialization processing, multiple *-Xi's* may be specified, each one yields more verbose output
b	print debug info on buildtree
trapuv	traps on uninitialized variables
t	print debug info on tymatch
e	print debug info on expression trees
x	print debug info on `?:' processing
I	intersperse source with object
T	force all names to be <=8 chars
u	generate ASCII ucode and ASCII symbol table
P	obsolete, do not use
g*n*	*n* is a digit, if *n*<>0, then writes debugging information to the symbol table for *dbx* debugging
EB	set big endian mode
EL	set little endian mode
O*n*	0<=*n*<=3, sets optimization level, doesn't affect *ccom*
mips*n*	1<=*n*<=3, sets the mips architecture, doesn't affect *ccom*
std	ANSI plus extension compliance
std*n*	n=0 for traditional compliance, n=1 for strict ANSI compliance. Note that the ANSI implementation is incomplete. n=o is the default.

Table 7.2: ccom options for -oldc only, 2 of 2

option	meaning
Nxnnnn	changes internal table limits, nnnn is the new value. You can use an unkown letter to make ccom list the possibilities, e.g. -XNz999. The known values for x and the default values are listed below; a temporary string space [1024] b temporary string space [4096] c temporary string buffers [40] d symbol table space [3000] e nesting level [100] f parameter stack space [1020] g switch table space [500] h tree space [1000] i delayed tree space [20] j hash table space [20] k file name space [100] l string literal space [2048] m initialization stack space [10] n line length [515] o files stack size [1024] p dimension table size [4200] q block nesting size [100]
l	obsolete, don't use
e	same as -Xe
w	same as -w1
wn	actions on warnings; n is one of: 0 print warnings, default if -w not specified 1 don't print warnings 2 print warnings, exit with nonzero exit status if any warnings occur 3 don't print warnings, exit with nonzero exit status if any (not printed) warnings occur
v	obsolete, don't use
framepointer	same as -Xframepointer
f	print the tree in the second pass
trapuv	same as -Xtrapuv

In addition, *ccom* accepts up to two filenames in the argument list. The first one, if present, is the input file. The second one, if there, is the output file. They default to *stdin* and *stdout* respectively.

Translation Limits

Table 7.3 shows the maximum limits imposed on certain items by the C compiler.

Table 7.3: C Compiler Limitations.

C Specification	Maximum	Maximum (-oldc)
Nesting levels Compound statements Iterations Selections Conditional compilations	200	<30
Maximum number of type modifiers (arrays, pointers, function, volatile)	*	9
Case labels	500	<500
Function call parameters	*	150
Significant characters External identifier Internal identifier	32	<32
* means no limit		

MIPS–C

This section covers the following topics:

- Specifying vararg or stdarg macros, *a requirement for all functions that take a variable number of argument.*
- Deviations from and extensions to C as defined in *The C Programming Language* by Kernighan and Ritchie (Prentice–Hall).
- Compatibility with previous versions of MIPS–C.
- New header files.

Varargs.h Macros

Currently, the MIPS C compiler supports *varargs.h*. The compiler also supports the ANSI *stdarg.h* method of variable argument accessing. Use *stdarg.h* wherever possible as *varargs.h* will be obsolete in the future.

If a function takes a variable number of arguments (for example, the C library functions *printf* and *scanf*), you must use the macros defined in the *varargs.h* header file.

The *va_dcl* macro declares the formal parameters *va_alist*, which is either the format descriptor for the remaining parameters or a parameter itself.

The *va_start* must be called within the body of the function whose argument list is to be traversed. The function can then transverse the list or pass its *va_list* pointer to other functions to transverse the list. The *type* of the *va_start* argument is *va_list*; it is defined by the *typedef* statement in varargs.h.

The *va_arg* macro accesses the value of an argument rather than obtaining its address. The macro handles those type names that can be transformed into the appropriate pointer type by appending an asterisk (*), which handles most simple cases. The argument type in a variable argument list must never be an integer type smaller than *int*, and must never be *float*. The current implementation of varargs does not work for *struct* types. Furthermore, the first parameter must not be a *double*.

For more information on the *varargs.h* macros, see *varargs(3)* in the *RISC/os Programmer's Reference Manual*. Figure 7.1 shows an example of the use of varargs macros; the expected output from the example is as follows:

```
load I 0 4
load I 4 4
add I
store I 0 4
```

```
#include <varargs.h>
#include <studio.h>
enum operations {load,store, add, sub};
main () {
    void emit ();
    emit (load, 'I', 0, 4);
    emit (load, 'I', 4, 4);
    emit (add, 'I');
    emit (store, 'I', 0, 4);
}
void
emit (op, va_alist)
/* emit takes a variable number of arguments and prints
     them according to the operation format */
enum operations op;
va_dcl {
va_list arg_ptr;
register int length, offset;
register char type;
va_start (arg_ptr);
switch (op) {
    case add:  /* print operation and length */
           type=va_arg (arg_ptr, int);
           printf ("add %c\n", type);
           break;
    case sub:  /* print operation and length  */
           type=va_arg (arg_ptr, int);
           printf ("sub %c\n", type);
           break;
    case load:  /* print operation, offset and length */
           type=va_arg (arg_ptr, int);
           offset=va_arg (arg_ptr, int);
           length=va_arg (arg_ptr, int);
           printf ("load %c %d %d\n", type, offset, length);
           break;
    case store:
           type=va_arg (arg_ptr, int);
           offset=va_arg (arg_ptr, int);
           length=va_arg (arg_ptr, int);
           printf ("store %c %d %d\n", type, offset, length);
}
va_end (arg_ptr);
```

Figure 7.1 Passing a Variable Number of Arguments to a C Function

Stdarg.h Macros

This is the ANSI C variable argument header file which replaces *varargs.h*. It must be included in each module which defines functions expecting a variable number of arguments. There is also a prototype syntax used to declare such functions, which must be used in modules that call stdarg functions. *Stdarg* correct *varargs* limitations such as the inability to pass *struct* parameters and not allowing the first argument to be a *double*.

As an example, the *stdarg* version of the *varargs* example would be coded as shown in Figure 7.2:

```
/* example variable argument function */
#include <stdarg.h>
#include <stdio.h>
enum operations {load, store,add, sub};
main()
    void emit  enum operation, ...);
          /* prototype with ... notiation*/
    emit (load, 'I', 0, 4);
    emit (load, 'I', 4, 4);
    emit (add, 'I');
    emit (store 'I', 0, 4);
}
void
emit (enum operations op, ...)
          /* note prototype function definition form */
           * emit takes a variable number of arguments
           * and prints them according to the operation format */

va_list arg_ptr;
register int length, offset;
register char type;
va_start  arg_ptr, op ;  * the argument prior to the variable part
                           of the function must be named here */
switch op
    case add:  /*  print operations and length */
          type=va_arg (arg_ptr, int);
          printf ("add %c\n", type);
          break;
    case sub: /* print operations and lenght  */
          type=va_arg (arg_ptr, int);
          printf ("sub %c\n", type);
          break;
    case load:  /* print operation, offset and length */
          type=va_arg (arg_ptr, int);
          offset=va_arg (arg_ptr, int);
          length-va_arg (arg_ptr, int);
          printf ("load %c %d\n", type, offset, lenght);
    case store:  /* print operation, offset and length */
          type=va_arg (arg_ptr, int);
          offset=va_arg (arg_ptr, int);
          length=va_arg (arg_ptr, int);
          printf ("store %c,%d %d/n", type, offset, length);
    }
va_end  arg_ptr);
```

Figure 7.2: Passing a Variable Number of Arguments to a C Function (stardg version)

Deviations

MIPS–C does not support the *entry* keyword, which has no defined use. Additionally, MIPS–C does not support the *asm* keyword, as implemented by some C compilers to allow for the inclusion of assembly language instructions.

Extensions

Extensions to K & R C include the following:

- A *cast* is allowed on the left side of an assignment operator.
- The *enumeration* type, a set of values represented by identifiers called enumeration constants; enumeration constants are specified when the type is defined. For information on the alignment, size, and value ranges of the *enumeration* type, see Chapter 3.
- The *void* type, which allows you to specify that no value be returned from a function.
- *void* *, which is a generic pointer. Any pointer may be assigned or compared to a pointer to void.
- The *volatile* type modifier, which is used when programming I/O devices and the *signed* type. In addition, the *const* keyword has been reserved for future use. For more information on the *volatile* modifier, see Chapter 3.
- *prototypes*, which are function prototypes as defined by the ANSI standard for C. Function prototypes can assist in locating assumptions about type compatibilities that may not be true when code is ported.
- C++ style comments are permitted.

Header Files

alloc.h

This header file should be included if the built–in version of the C library routine *alloca*(3) is desired. The built–in version is more efficient than the portable libc version because space is allocated on the stack and freed on exit.

The header file redefines the name *alloca*:

```
extern char *alloca(int size);
#pragma intrinsic(alloca)
```

Compatibility

This section describes the differences between the old MIPS C compiler (referred to as OldC, and available with the *-oldc* option) and the new compiler, which has three modes:

- MIPS-C (*-std0*)
- ANSI C (*-std1*)
- ANSI C with extensions (*-std*)

Differences Between OldC and All Modes

A warning is issued if constants exceed the limits (the value of ULONG_MAX). A similar warning occurs if octal and hexadecimal character escapes exceed the value of UCHAR_MAX. OldC does not issue a warning in these cases.

The value of the integer when a multi-character constant is converted may not be the same if the character type is signed and there are negative values in the constant.

The ANSI standard requires that a backslash followed by a carriage return be stripped early in the translation phases. In OldC, the pair was stripped fairly late (around translation phase 5, section 2.1.1.2). The behavior of *cpp* will be different; programs containing such constructs may not work properly when fed into the new compiler.

A *typedef* name used as a type specifier cannot be modified with a type modifier (i.e. *signed, unsigned*). A syntax error message is printed if this construct is found in a program. OldC permits modifying a user-defined type.

In the ANSI standard, preprocessor directives can occur in any column of a line as long as there is no preprocessing token in front of the '#' sign. OldC recognizes directives only if the '#' sign is on the first column of a line. The assembly language style of comment can be compiled with the *-oldc* option. To make this feature compatible, the new preprocessor conforms to the old style of directive if -DLANGUAGE_ASSEMBLY is used on the command line.

Declaring or defining a type within a function prototype causes the parameter to be incompatible with any other type. OldC permits this. For example, in the following declaration, if *struct S* has no previous declaration, any further type matching of the parameter list will result in an error; at the end of the prototype the scope closes, causing S to be forgotten.

```
int foo( struct S*p; );
```

OldC allows casting of the left hand side of the assignment expression, if the object pointed to by the left hand side and right hand side expressions have the same size. This is no longer permitted.

The *cpp* of OldC allows an *#if* directive in the middle of macro call. This is not permitted in any other mode.

OldC is very liberal regarding placement of braces in initializers. For example:

```
struct S { char i[10]; int i} y = {{"aeiou", 1}};
```

is acceptable in OldC, even though all standards require that the array be initialized to the nested initializer. The new compiler will complain about the initializer containing too many initial values since the array element is single–valued whereas the initializer is multi–valued.

Typedef names cannot be redeclared except within an inner block.

OldC and MIPS C (–std0)

The ANSI standard requires that each comment be replaced by one space character during preprocessing. In OldC, a comment is deleted entirely. The new behavior does not permit a comment to be used as a concatenation operator as in OldC.

The ANSI specification defines a string as a contiguous sequence of characters terminated by, and including, the first null character. As the result, a partial string is not a valid processing token, and it is not viable in the replacement list of a macro definition. The OldC preprocessor accepts a partial string. For example, in OldC, the following code fragment defines a partial string:

```
#define abc "123
```

and could be used as follows:

```
printf(abc 456");
```

In OldC, macros cannot be defined recursively. However, *–std0* mode supports recursively defined macro expansion.

OldC and ANSI C (–std1)

Local variables are allowed to hide externally declared variables at the same lexical level in OldC. This is treated as a redeclaration in ANSI C, and is an error:

```
f() {
  extern int i;
  int i;
}
```

In ANSI C, hexadecimal escape sequences in character and string constants are allowed. In OldC, this is not permitted. For example, '\x' is interpreted as 'x' in OldC.

The escape sequence '\a' is new to ANSI C. In OldC, this is translated to 'a' in and a warning message issued.

In ANSI C, a trailing comma in an enumerator list, as in:

```
enum good_stuff { cake, pie, cookie, };
```

generates a warning message. OldC permitted this without warning. In strictly standard mode (–std1), this is an error.

In ANSI C, an empty declaration (";") at the top level generates an error message. The empty declaration is tolerated in *–std0* mode.

In ANSI C, top level variable declarations (not function definitions) where there is no declaration specifier generate an error. OldC assumes that the variable is *extern int*.

A missing ending semicolon in the structure declaration list results in a warning message being issued. OldC permitted constructs such as:

```
struct {int a,b} a;
```

without warning.

In OldC, to declare two mutually referencing structures within a block, declarations similar to the following are required:

```
struct x { struct y *p; /* ... */ };
struct y { struct x *q; /* ... */ };
```

In ANSI C, if *struct y* is already defined in a containing block, the first field of *struct x* refers to the older declaration. Thus special meaning is given to the form:

```
struct y;
```

struct y now hides the outer declaration of *struct y*, and creates a new instance of the structure in the current block.

MIPS–C (–std0) and ANSI C (–std1)

In MIPS–C, array elements can have zero size; this is not allowed in ANSI C. For instance:

```
extern struct file file[];/* struct file is incomplete */
```

is accepted in *–std0* mode, but not in *–std1* mode.

In MIPS–C, local variables are allowed to hide externally declared variables at the same lexical level. In ANSI C, this is treated as a redeclaration.

In MIPS–C, array elements can have zero size. For example:

```
extern struct file file[];
   /* struct file is incomplete */
```

is accepted in MIPS–C, but is not permitted in ANSI C.

In MIPS–C, integral constants can have type *int* or *long*. In ANSI C, integral constants can have type *int, unsigned int, long,* or *unsigned long*. In MIPS–C, the type is *unsigned int* or *unsigned long* if the 'u' or 'U' suffix is used.

In MIPS–C, the preprocessor recognizes macro names inside strings in a macro expansion. This is not supported in ANSI C. In ANSI C, the # operator should be used (see the Macros section in Chapter 8 of this manual).

In ANSI C, a comment is replaced with one white–space character. In MIPS–C, a comment is removed.

In ANSI C, the preprocessor supports trigraphs. These are not supported in MIPS–C.

In MIPS–C, the preprocessor allows macro definitions to be redefined. This is not allowed in ANSI C.

Any macro name that is included from ANSI standard header file cannot be undefined, except in MIPS–C.

In ANSI C, the preprocessor issues a warning message if there is a preprocessing token following the *#endif* directive. In MIPS–C, no warning appears.

In ANSI C, the preprocessor issues a warning message if non–unique parameter name is detected for a macro definition.

In the following example:

```
struct y;
struct x { struct y *p; /* ... */ };
struct y { struct x *q; /* ... */ };
```

the reference to *y* in *struct x*, refers to the local declaration of *y*. In ANSI C, special meaning is given to the form:

```
struct y;
```

struct y now hides any declaration of *struct y* in an enclosing block, and creates a new instance in the current block.

ANSI C (–std1) and ANSI C with extensions (–std)

The C++ style comment is supported in ANSI C with extensions (*–std* mode).

Special Options for Compatibility

Comments are removed in OldC; this feature can be used as a concatenation operator in macro definitions. The *–oldcomment* option to the new compiler causes comments to be removed instead of replaced with a single space.

ANSI C Implementation

8

Introduction

The MIPS C compiler supports four variations of the C language:

- C as defined in *The C Programming Language* by Kernighan and Ritchie (Prentice Hall, 1978) with some ANSI C extensions (also known as MIPS C)
- ANSI C as defined in ANSI X3.159–1989 (American National Standards Institute, 1989)
- ANSI C with extensions
- An older version of MIPS C known as oldc

MIPS CThese variations of C are available with the following *cc* options:

–std0	MIPS C
–std1	strict ANSI C.
–std	ANSI C with extensions
–oldc	old version of MIPS C, uses the old *cpp* and *ccom.* instead of the new *cfe.* Oldc will not be supported in future releases of MIPS RISCompilers.

If none of the above options are used on the *cc* command line, the default is *–std0* unless an ANSI C license is acquired, in which case the default is *–std.*

Chapter 7 contains a discussion of compatibility issues for the variations of C provided by the MIPS compiler.

This chapter discusses new features of ANSI C. A complete description of the Language may be found in ANSI X3.159–1989. In addition to describing the C language, the ANSI standard for C describes the functionality of the preprocessor and the library routines. This chapter discusses the following topics:

- Translation Limits
- Preprocessor
- Language
- Library Routines
- Implementation Defined Behavior
- Quiet Changes
- Extensions to ANSI C

ANSI C is identical to MIPS C in many respects. Each of the following sections describes features of ANSI C that are not found in MIPS C.

Note: With *–systype bsd43* and *–systype sysv*, a *conforming freestanding implementation* of ANSI C is available and accepts any strictly conforming program in which the use of library routines is confined to those defined in the standard headers *float.h, limits.h, stdarg.h,* and *stddef.h.*

A *conforming hosted implementation* of ANSI C is not yet available. This will be provided in a future release and will include the new and modified header files and libraries.

Translation Limits

The MIPS C compiler uses dynamic data structures and therefore, program components are limited only by the amount of available memory. The following list indicates minimums which are guaranteed (i.e. a program that meets but does not exceed each minimum is guaranteed to compile). However, if a program significantly exceeds one or more minimums, it is possible to run out of memory and receive an error message on a component that has not yet reached its minimum.

- Compound statements (a set of statements grouped with braces), iteration control statements, and selection control statements may be nested at least 15 levels.
- Conditional include directives may be nested 8 levels.
- Arithmetic, structure, union, or incomplete type declarations may have at least 12 pointer, array, and function declarators modifying them.

- A declaration may have at least 31 nested levels of parenthesized declarators.
- An expression may have at least 32 nested levels of parenthesized expressions.
- An internal identifier or macro name may have 32 significant characters.
- An external identifier may have 32 significant initial characters.
- A single translation unit may have at least 511 external identifiers.
- A block may have at least 127 identifiers declared with block scope.
- A single translation unit may have at least 1024 macro identifiers defined simultaneously.
- A function definition may have at least 31 parameters and a function call 31 arguments.
- A macro definition may have at least 31 parameters and a macro invocation 31 arguments.
- A logical source line may have at least 509 characters.
- A string literal or wide string literal may have at least 509 characters (after string concatenation).
- An object may consist of at least 32767 bytes.
- A *switch* statement may have 257 *case* labels (excluding any nested *switch* statements).
- A single *struct* or *union* may have at least 127 members.
- A single enumeration may have at least 127 enumeration constants.
- A single structure declaration may have at least 15 levels of nested structure or union definitions.

Preprocessor

Directives

Any token may be continued on the following line with a back-slash (\) followed by a new-line. Previously, only character strings could be continued in this fashion.

The # and the directive name (i.e. *line, ifdef*) are separate tokens.

A null directive, consisting of a # followed by a new-line, is permitted and has no effect.

White–space, consisting of any number of spaces and tabs, may appear in directives between preprocessing tokens anywhere in the line. Directives may be nested at least eight levels.

New Directives

#Elif

The *#elif* (else if) directive allows nested *#if*s to be simplified:

```
#if x < 0
...
#elif x == 0
...
#else
...
#endif
```

#Error

The *error* directive is as follows:

```
#error token-sequence
```

This directive causes a warning diagnostic message to be generated that includes the specified token sequence.

#Pragma

The *pragma* directive has the form:

```
#pragma token-sequence
```

The *intrinsic, function, weak,* and *pack* pragmas are supported. Any unrecognized pragmas are ignored by the compiler and a warning diagnostic message is generated.

Intrinsic Pragma

Some library functions can be compiled in-line using the *intrinsic* pragma. This directive affects the specified function from the pragma until the end of the file or the next function *function* pragma that references the same function.

```
#pragma intrinsic (function1 [,function2] ...)
```

The following functions can be compiled in–line using the *intrinsic* pragma:

```
alloca(), sqrt(), strcpy()
```

Function Pragma

The function name must be defined at the time the *#pragma* is processed. If a function name is not recognized as an intrinsic, no action is taken. Intrinsic processing can be turned off using -D_NO_INTRINSICS on the command line. In *-std1* and *-std* modes, intrinsics are enabled by default. In *-std0* mode, intrinsics are disabled by default. To enable intrinsics, add -D_INTRINSICS to the command line.

The *function pragma* escapes the in-line code generation. A function call is forced for the specified functions for all subsequent calls unless an *intrinsic pragma* is encountered further on.

```
#pragma function (function1 [,function2] ...)
#pragma function ()
```

The second form of the function pragma disables intrinsic functionality of all currently intrinsic functions.

The *function* and *intrinsic pragmas* can only be used at the file scope level.

Weak Pragma

The *weak* pragma defines a new weak external symbol and associates this new symbol with an external symbol.

```
#pragma weak(secondary_name, primary_name)
#pragma weak secondary_name = primary_name
```

These two forms of the weak pragma are equivalent and cause the *primary_name* to be a weak symbol and associate it with the *secondary_name*. If a weak symbol and a strong symbol of the same name exist, the strong symbol is resolved and a warning is issued for the unresolved weak symbol.

A third form of the *weak* pragma may be used to indicate that a global symbol should not cause an error if it is not resolved by the linker:

```
#pragma weak identifier
```

Pack Pragma

The *pack* pragma is used to change the alignment restrictions on structure members.

```
#pragma pack(n)
#pragma pack()
```

In the first form, *n* specifies the new alignment restriction in bytes. If *n* is omitted, as in the second form, the default alignment restriction is used (8 bytes, the alignment requirements for a double).

Directives with Additional Functionality

Defined

The *defined* unary operator is used with an *#if* and is equivalent to an *#ifdef*. The new form is provided to allow multiple tests in one directive. For example:

```
#if defined (debug) && defined (error)
```

#Include

ANSI C defines *#include* as follows:

```
#include identifier
```

After all macro replacement is completed, the identifier must be either *"filename"* or *<filename>*.

#Line

The ANSI C *line* directive has the form

```
#line line-number filename
```

The line–number may be a macro that has a decimal value or a constant. The filename may be a macro, a string literal, or a filename.

Macros

Operators

There are two new operators for macro parameters. A # placed before a parameter causes the # and the parameter to be replaced with a string consisting of the parameter name. For example, if the following macro

```
#define print(x) printf(#x " = %d", x)
```

is called as

```
print(result);
```

It is expanded to

```
printf("result" " = %d", result)
```

Adjacent string literals are concatenated, so the result of the macro call becomes

```
printf("result = %d", result);
```

New macros

ANSI C defines a new *offsetof* macro:

```
offsetof(type, member)
```

The macro expands to an integral constant expression of type *size_t* and indicates the offset in bytes from the beginning of the structure to the indicated member.

ANSI C defines *errno* as a macro that expands to a modifiable lvalue of type *int*.

ANSI C defines the macros EXIT_SUCCESS and EXIT_FAILURE in *stdlib.h*. These macros expand to integral expressions that may be used as the argument to *exit*() (see *exit*(2)) to indicate successful or unsuccessful termination to the host environment.

FOPEN_MAX is the minimum number of files that it is guaranteed can be open simultaneously.

Predefined Macros

All predefined macros begin with an underscore that is followed by a capital letter or another underscore.

The following predefined macros provide information about the file being compiled and cannot be redefined or undefined:

```
__DATE__  date the file was compiled
__TIME__  time the file was compiled
__FILE__  name of the file being compiled
__LINE__  line number in the file being compiled
__STDC__  has the value 1 if -std1 is used on the cc
          command line, 0 if -std is used, and is
          undefined if -std0 is used.
```


Expressions

Constant expressions in preprocessor directives may not contain *casts* or *enums*.

Language

Trigraph sequences

A trigraph is a sequence of three characters that is used to represent a single character. Trigraphs are intended to be used on machines where the character set does not contain all of the characters required by C.

A trigraph sequence is two question marks followed by another character. The trigraphs and the characters they represent are as follows:

??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

You should not need to use trigraph sequences. However, if any of these sequences appear in string literals in a source file, they will be interpreted as a trigraph which may cause unexpected results.

main()

Argv, the argument list passed to *main()*, ends with a NULL pointer. Therefore the number of arguments reported by *argc* is one more than the number of parameters passed to the program. *Argc* and *argv* are modifiable by the user.

Declarations

Keywords

ANSI C has defined the following new keywords: *const, volatile, signed, enum,* and *void.*

Identifier Name Space

The following categories of identifiers have separate name spaces:

- Label names.
- Tags of structs, unions, and enums.
- Each struct or union has a separate name space for its members.
- All other identifiers.

The identifiers that are found in function prototypes have their own name space. The scope of these variables is from the name to the end of the prototype definition.

Constants

Unsigned Constants

Unsigned constants have a *u* or *U* as a suffix:

```
4321U or 4321u
```

Unsigned long constants are suffixed with both *u* or *U* and *l* or *L*:

```
987654321UL
```

Floating–point Constants

Floating–point constants are specified with an *f* or *F* suffix:

```
0.2F   or   1e7f
```

Floating–point constants can also be specified with a decimal point (4.321) or an exponent (6e–4) as in MIPS–C.

Wide Constants

A wide character constant has type `wchar_t` and an *L* as a prefix:

```
L'z'
```

The value of a wide character constant containing one multibyte character is the corresponding wide character code defined by the library function *mbtowc*.

A wide string literal is prefixed with an *L*:

```
L"abc"
```

String Constants

In ANSI C, string concatenation occurs when two string literals are adjacent. For example:

```
printf("a character string that is continued"
       "on the next line");
```

String literals containing trigraph sequences (see the Trigraph Sequences section) may have unexpected results. For example, the string "what??!" becomes "what|" during preprocessing.

There are two new escape sequences for use in string literals:

```
'\a'  alert
'\v'  vertical tab
```

In addition, a '\x' sequence introduces a hexadecimal escape sequence that represents a character. One or two hexadecimal digits may follow the 'x'.

```
'\xb'   or   '\x1e'
```

All lower case alphabetic escape sequences are reserved for future use.

Type modifiers

ANSI C defines the following new type modifiers:

const indicates that the variable or argument will not be changed. *const* variables are placed in the read only section of the object file.

volatile is used to suppress undesirable optimizations (e.g. reads that may appear to be redundant).

signed may modify *short, int, long int,* or *char.* If a type is not modified by either *signed* or *unsigned,* it defaults to `signed`, except for `char` which is *unsigned* by default (unless the *–signed* flag is used at compile time).

Types

Bit fields may be type *int, unsigned int,* or *signed int* only.

ANSI C introduces a new floating-point type *long double* intended to give greater precision than *double.* In MIPS implementation, *long double* and *double* are the same.

ANSI C defines the following new types:

void is any empty set of values. This type is commonly used for return values of functions that do not return a value and as a generic pointer (*void**). Any pointer type may be assigned to a pointer to *void. void* cannot be used to declare types.

An *enum* is a set of named integer constants. For example:

```
enum primary {red, yellow, blue};
```

Typedefs

The following typedefs are available in ANSI C:

jmp_buf is declared in *jmpbuf.h.* It is an array type suitable for holding information needed to restore a calling environment and may be used as the type of the argument to *setjmp*(3).

size_tis defined in *stddef.h* and is an unsigned integral type that is the result of the *sizeof* operator.

ptrdiff_t is defined in *stddef.h* and is a signed integral type that is the result of subtracting two pointers.

sig_atomic_t is defined in *signal.h* and is an integral type that can be accessed as an atomic entity (even in the presence of asynchronous interrupts).

wchar_t is defined in *stddef.h* and is an integral type capable of holding values representing all codes of the largest extended character set among the supported locales.

Empty Declarations

Structures and unions may have empty declarations. This allows the user to define mutually referential structures and unions. For example:

```
struct y;
struct x {struct y * yptr;};
struct y {struct x * xptr;};
```

The first *struct y* in the above example has an empty declaration. This ensures that *struct x* refers to the local definition of *struct y* and not a global definition that may exist.

Tagless declarations

A *struct* or *union* that has no tag name following its declaration may be referred to only by the declaration in which it is found.

```
struct {
      int i;
} a,b;
```

A tagless enumeration can be used to define constants (which can also be defined with the *#define* preprocessor directive):

```
enum {cow, sheep, goat, chicken};
```


Structs, Unions, Arrays

Arrays

Array dimensions must be constant integral expressions and greater than zero.

In ANSI C, automatic arrays may be initialized provided the initializer list consists of constant expressions.

Structures and Unions

Automatic *structs* and *unions* may be initialized either with a constant expression or a non-constant expression of the same type. When an automatic *union* is initialized, the value stored is cast to the type of its first member.

Structures and unions cannot be cast; a pointer to a structure or union can be cast to a pointer of another type.

A structure or union can be passed as an argument to a function by value (the *struct* or *union*) as well as by address (a pointer to a *struct* or *union*) and can also be returned from a function by value or address.

Expressions

Any parentheses in expressions must be honored at execution time.

Operators

Assignment operators, such as **+=** or ***=**, are a single token; no space is allowed between the operator and the **=**. Assignment operators of the form **=***op* are not permitted. You should use the *op*= form.

ANSI C provides a *unary plus* operator. In the following example:

```
i = +10;
```

10 is assigned to *i*.

A cast expression is not an lvalue and cannot have a value assigned to it.

Arithmetic

When a *float* is converted to an integral type, the fractional part is discarded.

The controlling expression of a *switch* statement must be an integral type.

Integral Promotions

A character, short integer, or integer bit-field, whether signed or unsigned, or an enumeration may be used in expressions wherever an integer may be used. If all the values of the original type can be represented by an *int*, the value is converted to *int*; otherwise the value is converted to *unsigned int*. This is a *value preserving* method of integral promotion.

Many C implementations have used an *unsigned preserving* method of integral promotion. This approach promotes an unsigned character or unsigned short integer to *unsigned int*.

In most cases, the two schemes give the same effective result. Both give the same result in even more cases in implementations with twos complement arithmetic and quiet wraparound on signed overflow (that is, most current implementations). In these implementations, differences between the two schemes appear when the following conditions are both true:

- An expression involving an *unsigned char* or *unsigned short* produces an *int* length result in which the sign bit is set.

- The result of the preceding expression is used in a context in which its sign is significant.

  In such circumstances, *value preserving* integral promotion causes the negative signed integer to become a very large unsigned integer, which may not be the desired result. This can be avoided with the use of appropriate casts.

Note: *-std0* uses the unsigned preserving method.

Conversion Rules

The conversion rules for ANSI C are as follows:

First, if either operand is *long double*, the other operand is converted to *long double*.

Otherwise, if either operand is *float*, the other operand is converted to *float*.

Otherwise, the integral promotions are performed on both operands. Then the following rules are applied:

- If either operand is *unsigned long int*, the other operand is converted to *unsigned long int*.
- Otherwise, if one operand is *long int* and the other is *unsigned int*, the *unsigned int* is converted to *long int*.
- Otherwise, if either operand is *long int*, the other operand is converted to *long int*.
- Otherwise, if either operand is *unsigned int*, the other operand is converted to *unsigned int*.
- Otherwise, both operands are *int*.

Sequence Points

The following are known as sequence points:

- A function call, after the arguments have been evaluated.
- The end of the first operand of the following operators: logical AND (&&), logical OR (||), conditional (?), and comma (,).
- The end of a full expression: an initializer, the controlling expression of an *if*, *switch*, *while*, or *do* statement, each of the three expressions of a *for* statement, or the expression in a *return* statement.

At a sequence point, all side effects of previous evaluations are complete and no side effects of subsequent evaluations have taken place.

If processing is interrupted by a signal, only the value of objects as of the previous sequence point may be relied on. Objects modified since the last sequence point and before the next, need not have received their correct values.

Note: Order of evaluation in expressions is unspecified except for sequence points.

Pointers

A function pointer cannot be cast to a data pointer or a pointer to void and a data pointer or pointer to void cannot be cast to a function pointer.

A pointer cannot be converted to another pointer type without an explicit cast.

Functions

ANSI C has a new style of function definition that is similar to function prototype style. The following function:

```
sum(i,j)
int i;
int j;
{
...
}
```

can now be defined as:

```
init sum(int i, int j)
{
...
return i;
}
```

A function with no arguments would be defined as follows:

```
print(void)
{
...
}
```

Function Prototypes

The following is an example of a function prototype:

```
int sum(int x, int y);
```

This declaration indicates that the function sum expects two *int* arguments and returns an *int*. The definition of the function and each call to the function must agree with the prototype; otherwise, an error message is generated by the compiler.

A prototype for a function with a variable number of arguments would be declared as follows:

```
int print(char *format, ...);
```

The ellipsis (...) indicates that the number and type of the arguments may vary and can only appear at the end of the argument list. At least one parameter must precede the ellipsis in the declaration.

Function Pointers

A function pointer may be used to call the function in either of the following ways:

```
(*func_ptr)();
```

OR

```
func_ptr();
```


Implementation Defined Behavior

The ANSI Standard for C allows implementations to vary in specific instances. This section describes the implementation defined behavior of the MIPS ANSI C compiler.

Translation

Diagnostic messages are identified as follows:

```
compiler-phase: error-type: filename, line: error-message[(section- number)]
```

and are followed by the line in question and an indication of the location of the problem. For example:

```
cfe: Error: misc.c, line 7: syntax error
      lon int *c;
 ------^
```

The error message may be followed by the section number of the ANSI C standard that has been violated.

Environment

The arguments to *main*() are:

argv[0]	the name of the executable file
argv[1]...argv[argc – 1]	command line parameters
argv[argc]	a null pointer

An interactive device is a video display terminal.

Identifiers

Only the first 31 characters of an internal identifier are significant.

An external identifier has 6 significant characters.

Case is significant for external identifiers.

Characters

The source and execution character sets are identical and are as defined in the ANSI standard for C.

The C locale is the default locale. Currently, no other locales are supported; therefore, there are no shift states for encoding multibyte characters.

There are eight (8) bits in a character in the execution character set.

Source characters are mapped one–to–one into the execution character set.

There are no invalid characters or escape sequences in the basic execution character set.

The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character is as follows for character constants with 2 to 4 characters:

in big–endian mode:

2 characters, "ab":

```
'a'*256 + (unsigned)'b'
```

3 characters, "abc":

```
'a' * 65536 + (unsigned)'b' * 256 + (unsigned)'c'
```

4 characters, "abcd":

```
'a' * 16777216 + (unsigned)'b' * 65536 +
(unsigned)'c' * 256 + (unsigned)'d'
```

and in little–endian mode:

2 characters, "ab":

```
'b' * 256+ (unsigned)'a'
```

3 characters, "abc":
'c' * 65536 + (unsigned)'b' * 256 + (unsigned)'a'

4 characters, "abcd":
'd' * 16777216 + (unsigned)'c' * 65536 +
(unsigned) 'b' * 256 + (unsigned)'a'

The C locale is used to convert multibyte characters into corresponding wide character codes. The value of the wide character is equal to the value of the first byte in the multibyte sequence (whose value is taken as an unsigned value).

A "plain" *char* has the same range of values as an *unsigned char*.

Integers

The ranges of values for the integral types are:

`char`	0 to 255
`signed char`	-128 to 127
`short int`	-32768 to 32767
`int`	-2147483648 to 2147483647
`long int`	-2147483648 to 2147483647
`unsigned char`	0 to 255
`unsigned short int`	0 to 65535
`unsigned int`	0 to 4294967295
`unsigned long int`	0 to 4294967295

Converting an integer to a shorter signed integer causes a representation change by discarding the high order bits. Converting an unsigned integer to a signed integer of equal length does not cause a representation change. However, the converted value may be negative.

Bitwise operations on signed integers produce signed results, represented in two's complement. How the value is interpreted depends on whether the sign bit is on or off after the operation. The operation is performed on the data as if the values were unsigned.

When the operator is % (remainder of integer division), if the dividend is negative and the divisor is positive, the result is negative. If the dividend is positive and the divisor is negative, the result is negative. If both are negative the result is negative.

A right shift of a negative signed integral type causes the sign bit to be replicated.

Floating Point

The ranges of values for the floating point types are:

float	1.17549435e–38 F to 3.40282347e+38F
double	2.2250738585072014e–308 to 1.7976931348623157e+308
long double	2.2250738585072014e–308 to 1.7976931348623157e+308

When an integral number is converted to a floating–point number that cannot be exactly represented, the number is truncated to be nearest value that can be represented.

When a floating–point number is converted to a narrower floating–point type, the value is truncated or rounded to the nearest value that can be represented by the narrower type.

Arrays and Pointers

size_t is defined in *stddef.h* to be *unsigned int*.

Casting a pointer to an integer or vice versa does not cause any representation change.

ptrdiff_t is defined in *stddef.h* to be *int*.

Registers

The *register* storage class specifier cannot be used with structure or array declarations. A *register* variable may be changed to a *non–register* type or a *non–register* type changed to *register* by the optimizer.

Structures, Unions, Enumerations, and Bit–fields

Consider a union as a block of memory the size of the union. The result if a member of a union has a value stored in it and is subsequently accessed using a member of a different type is defined as the value of the accessed type at that block of memory. If the size of the type stored is smaller than the accessed type, the result is undefined. If the type stored is a structure with holes, and the accessed value overlaps any of the holes, the value is undefined. If a floating point value is stored, the bit pattern for the IEEE format for single or double precision numbers is stored. A NULL pointer is stored as a bit–pattern of all zeroes.

Each member of a structure is aligned on the boundary required by its type. Padding is added between members as necessary. See Chapter 2 of this manual for more details on alignment of data types.

A plain *int* bit–field is a *signed int* bit–field.

Bits within an integer bitfield are allocated most significant bit first in big–endian mode and least significant bit first in little–endian mode

A bit–field cannot straddle a storage unit boundary.

The values of an enumeration declaration are type *int*.

Qualifiers

Each time a value is needed from a *volatile* object, a "read" access is made to it. Each time the value needs to be written, a "write" access is made. This ensures that *volatile* objects are accessed in the same way as in the abstract semantics. However, the one exception is when a *volatile* bitfield is written to, the hardware constraints may force a "read" to occur prior to the "write", in order to read the values of the parts of the storage unit that are not changed in the write. Avoid using *volatile* bitfields unless you really know what you are doing.

Declarators

An arithmetic, structure, or union type may have at least 12 declarators modifying it. The maximum number of declarators allowed is limited only by the amount of available memory.

Statements

The maximum number of *case* values in a *switch* statement is limited only the the amount of available memory.

Preprocessing Directives

The value of a single–character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. A single–character character constant is an unsigned character and therefore cannot be negative.

When an include file is specified as *"filename"*, the current directory is searched first, and if not found, then */usr/include* is searched. If an include file is specified as *<filename>*, */usr/include* is the only directory searched. The *–systype bsd43* or *–systype sysv* options to `cc` modify the directory searched. The *–I* option can also be used to modify the directory searched. See Chapter 1 of this manual or *cc*(1) in the *User's Reference Manual*.

MIPS ANSI C supports the *intrinsic, function, weak,* and *pack #pragmas*.

When the date or time of translation is not available, the definitions of the __DATE__ and __TIME__ macros are January 1, 1970 and 00:00:00, respectively.

Library Functions

The macro NULL expands to the value zero (0).

assert writes a message to the standard error output in the following form:

`Assertion failed:` *expression*`, file` *filename*`, line` *xxxx*

The sets of characters tested for by the *isalnum, isalpha, iscntrl, islower, isprint,* and *isupper* functions are as follows:

isalnum	0–9, a–z, A–Z
isalpha	a–z, A–Z
iscntrl	the delete character (0177) andcharacters less than ASCII code 040.
islower	a–z
isprint	any printable character (ASCII code 040 to 0176)
isupper	A–Z

The value returned by the mathematics functions on domain errors is either EDOM (33) or ERANGE (34).

The mathematics functions set the macro *errno* to the value of the ERANGE (34)on underflow range errors.

When the *fmod* function has a second argument of zero, zero is returned.

The set of signals, and the default action for each, that are accepted by the *signal* function are as follows:

Signal	Action	Event
SIGHUP	Exit	hangup
SIGINIT	Exit	interrupt
SIGQUIT	*	quit
SIGILL	*	illegal instruction
SIGTRAP	*	trace trap
SIGABRT	*	abort
SIGEMT	*	EMT instruction
SIGFPE	*	floating point exception
SIGKILL	Exit	kill (cannot be caught or ignored)
SIGBUS	*	bus error
SIGSEGV	*	segmentation violation
SIGSYS	*	bad argument to system call
SIGPIPE	Exit	write on a pipe with no one to read it
SIGALRM	Exit	alarm clock
SIGTERM	Exit	software termination signal
SIGUSR1	Exit	User defined signal 1
SIGUSER2	Exit	User defined signal 2
SIGCLD	Ignore	child status has changed
SIGSTOP	Stop	stop (cannot be caught or ignored)
SIGSTP	Stop	stop signal generated from keyboard
SIGPOLL	Exit	selectable event pending
SIGIO	Ignore	I/O is possible on a descriptor (see *fentl*(2))
SIGURG	Ignore	urgent condition present on socket
SIGWINCH	Ignore	window size change
SIGVTALRM	Exit	virtual time alarm (see *getitimer*(2))
SIGPROG	Exit	profiling timer alarm (see *getitimer*(2))
SIGCONT	Stop	continue after stop
SIGTTIN	Stop	background read attempt from control terminal
SIGTTOU	Stop	background write attempted to control terminal
SIGXCPU	*	cpu time limit exceeded
SIGXFSZ	*	file size limit exceeded
SIGLOST	exit	resource lost (e.g. record-lock)

A * indicates that the action is to terminate the process and produce a core image.

The default handling is not reset if the SIGILL signal is received by a handler specified to the signal function.

The last line of a text stream does not require a terminating new–line character.

Space characters that are written out to a text stream immediately before a new–line character appear when the text is read.

In RISC/os, a binary stream is the same as a text stream.

When a file is opened in append mode, the file position indicator is initially positioned at the end of the file.

A write on a text stream does not cause the associated file to be truncated beyond that point.

A zero–length file actually exists.

Valid file names consist of 1 to 14 characters. The null character and the slash (/) may not appear in a filename.

It is permissible to open the same file multiple times.

When the *remove* function is given the name of an open file as its argument, –1 is returned and the file is not removed.

If a file with the new name exists prior to a call to the *rename* function, this file will be removed.

The *%p* conversion of the *fprintf* function print the address indicated by the pointer in hexadecimal.

The input for the *%p* conversion of the *fscanf* function is expected to be a pointer previously printed by *fprintf*.

A '–' character that is neither the first nor the last character in the scan list for *%[* conversion in the *fscanf* function indicates a range of values (e.g. 0–9). The value preceding the '–' must be lexically less than or equal to the value after the '–'.

When the *fgetpos* or *ftell* functions fail, the macro *errno* is set to EBADF.

The *perror* function generates a message consisting of the text string, if any, that was passed to *perror*, followed by a colon and a space if the text string is non–empty, followed by the system message for the error number indicated by the macro *errno*.

If the *calloc*, *malloc*, or *realloc* functions are called with a size request of zero, the function returns zero.

The *abort* function closes all open files before terminating the program.

The status returned by the exit function if the value of the argument is other than zero, EXIT_SUCCESS, or EXIT_FAILURE, is the argument that was passed to the function.

putenv(3) is used to modify the environment list used by *getenv*. *putenv* is called as follows:

```
putenv(char *string)
```

string is of the form "*name=value*". The environment variable *name* is set to *value* by changing an existing variable or creating a new one.

The *system* function expects a text string that is a shell command which it passes to *sh*(1). The function waits until the shell completes and returns the exit status of the shell.

The error message string returned by the *strerror* function is the system message corresponding to the error number.

The local time zone is PST and Daylight Saving Time is PDT.

The era used by the *clock* function is 00:00:00 GMT, January 1, 1970.

alignof returns the alignment assigned to *type* by the compiler. This extension is independent of any mode (*–std[01]*) and is supported when the user includes *alignof.h*.

Quiet Changes

This section describes the *quiet changes* that occurred in the ANSI C implementation. These are changes in the functionality of the compiler that are not noticeable at compile time, but produce different results during execution.

For example, the following line of code

```
i=-*p;
```

has new meaning in ANSI C. Previously, this would decrement the value of *i* by the value stored in *p*. In ANSI C, the negated value stored in *p* is assigned to *i*.

- Programs with character sequences such as ??! (a trigraph) in string constants, character constants, or header names produce different results.
- A program that depends on internal identifiers matching only a limited number of significant characters may behave differently.
- A program that relies on file scope rules may be valid under block scope rules but behave differently.
- Unsuffixed integer constants may have different types. In K & R, unsuffixed decimal constants greater than INT_MAX, and unsuffixed octal or hexadecimal constants greater than UINT_MAX are of type *long*.

- A constant of the form '\078' is valid, but has different meaning. It denotes a character constant whose value is the combination of the value of the two characters '\07' and '8'. In some implementations, the old meaning is the character whose code is 078 (equal to 64 decimal).
- A constant of the form '\a' or '\x' has different meaning.
- A string of the form "\078" is valid, but has different meaning. The new meaning is the same as for a constant '\078'.
- A string of the form "\a" or "\x" has different meaning.
- Identical string literals may be represented by a single copy of the string in memory, but this is not required; a program that depends upon either scheme may behave differently.
- Expressions of the form x=-3 have different meaning.
- A program that depends on unsigned preserving arithmetic conversions now behaves differently, probably without complaint.
- Expressions with *float* operands may now be computed at lower precision.
- A program that uses *#if* expressions to determine information about the execution environment may behave differently.
- The empty declaration *struct x;* now has meaning.
- A program which relies on a bottom-up parse of aggregate initializers with partially elided braces does not yield the expected initialized object.
- Expressions of type *long* and constants in *switch* statements are no longer truncated to *int*.
- Functions that depend on parameters of type *char* or *short* being widened to *int*, or *float* to *double*, may behave differently.
- A macro that relies on formal parameter substitution within a string literal now produces different results.
- A program that relies on size zero allocation requests returning a non-null pointer now behaves differently.

Extensions to ANSI C

The features discussed in this section are available with the *-std* option to the *cc* command (see *cc*(1)).

Comments

The C++ style of comment

```
printf("sun %d\n", i);// print results
```

is permitted. The comment is introduced by the '//' and extends to the end of the line. The comment characters '//' have no special meaning within a // comment and are treated just like other characters.

alloca

```
#include <alloca.h>
char     *alloca(int);
```

alloca allocates the requested number of bytes of space in the stack frame of the caller. This temporary space is automatically freed on return. If *alloca.h* is included, *alloca* will be a built-in function. The built-in function is more efficient than the portable libc.a version, but can only be applied to integral types (char, signed and unsigned integer, and enumeration). This extension is independent of any mode (*-std[01]*) and is supported when the user includes *alloca.h*.

alignof

```
#include      <alignof.h>
unsigned int  alignof(type);
```

alignof returns the alignment assigned to *type* by the compiler. This extension is independent of any mode (*-std[01]*) and is supported when the user includes *alignof.h*.

cast lhs

A cast is allowed on the left hand side of an assignment operator.

Byte Ordering

A

What Is Byte Ordering?

A machine's byte ordering scheme (or whether a machine is big–endian or little–endian) affects memory organization and defines the relationship between address and byte position of data in memory. MIPS machines can be big–endian or little–endian.

Big–Endian Byte Ordering

Big–endian machines number the bytes of a word from 0 to 3. Byte 0 holds the sign and most significant bits. For halfwords, big–endian machines number the bytes from 0 to 1. Again, byte 0 holds the sign and most significant bits. Machines that use big–endian schemes include the IBM s/370 and Motorola MC68000.

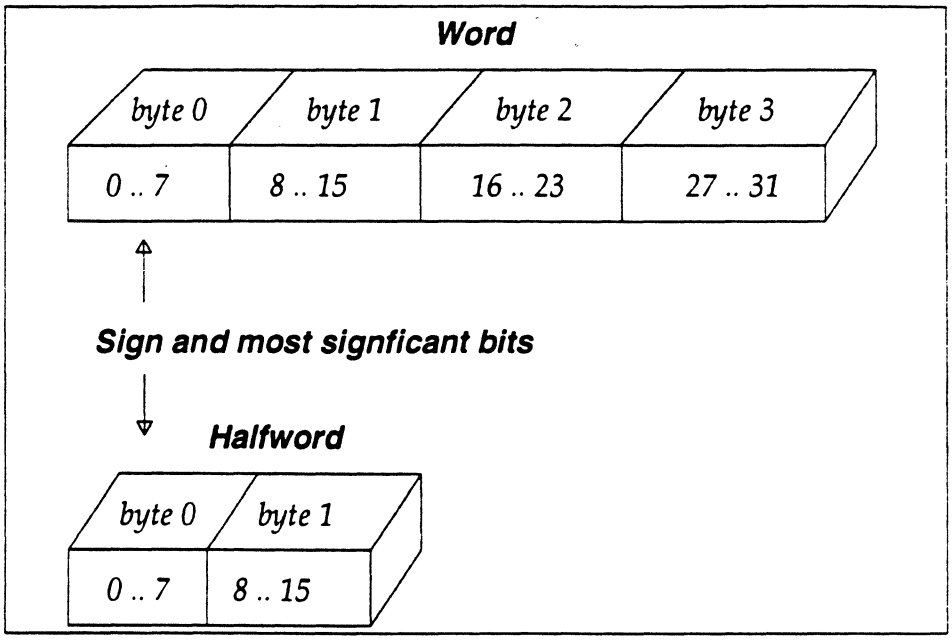


Figure A.1 Big–endian byte ordering

Little–Endian Byte Ordering

Little–endian machines number the bytes of a word from 3 to 0. Byte 3 holds the sign and most significant bits. For halfwords, little–endian machines number the bytes from 1 to 0. Byte 1 holds the sign and most significant bits. Machines that use little–endian schemes include: DEC VAX & 11/780, Intel 80286, and National Semiconductor 32000.

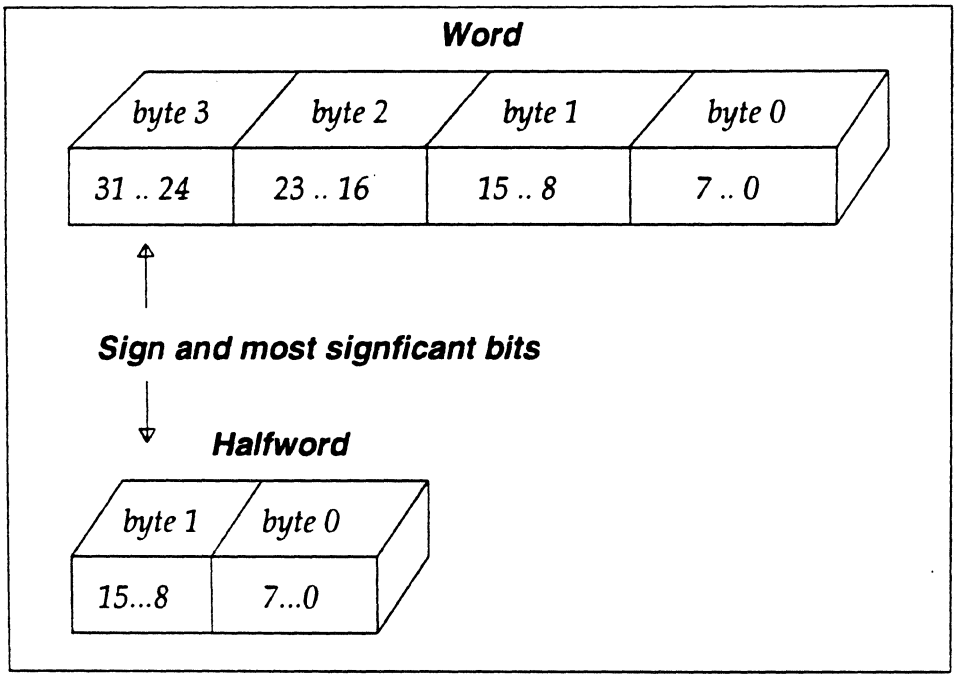


Figure A.2 Little–endian byte ordering

Index

T

U

V

W

MIPS RISCompiler Programmer's Guide

NEC
NEC Electronics Inc.
CORPORATE HEADQUARTERS
475 Ellis Street
P.O. Box 7241
Mountain View, CA 94039
TEL 415-960-6000

For literature, call toll-free 7 a.m. to 6 p.m. Pacific time: **1-800-366-9782**
or FAX your request to: **1-800-729-9288**

50777