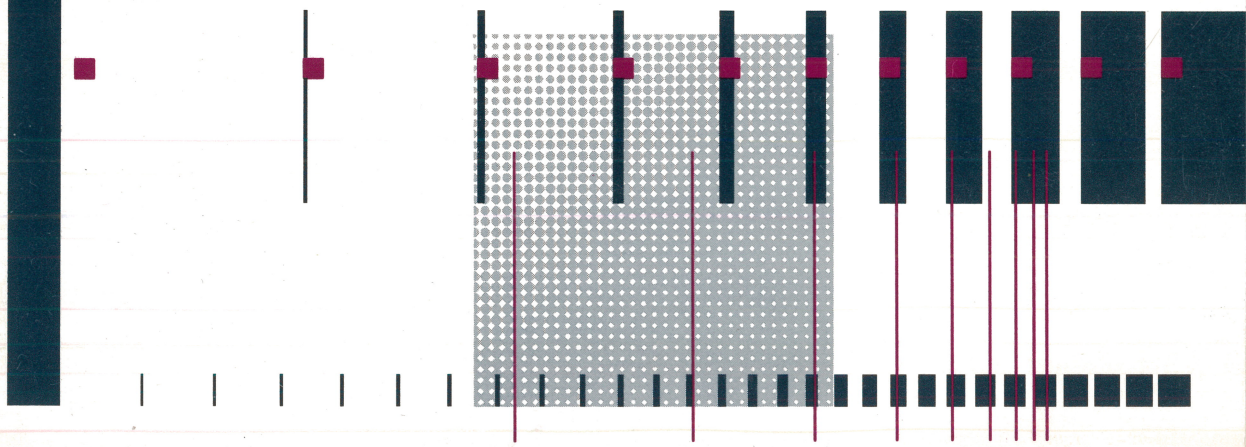
MIPS® RISCompiler™ Porting Guide

Order Number 3140DOC



mips

The power of RISC is in the system.

MIPS® RISCompiler™ Porting Guide

Order Number 3140DOC

June 1989

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

84–00065–001(A)/02–00333–001

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

Customer Service Telephone Numbers:

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

Electronic Mail: decwrl!mips!hotline

84–00065–001(A)/02–00333–001

This book provides information that you need to know in order to port programs from other operating systems to the RISC/os operating system.

Prerequisites

This book assumes you are porting a program from another machine to a MIPS RISComputer. You should understand both RISC/os and the operating system that you are porting from. You should also be fluent in the programming language you're using and be comfortable using UNIX system tools to write your programs.

What Does This Book Cover?

This book has these chapters:

- **Chapter 1—Overview.** Describes a process to follow when you are porting a program to the MIPS computing environment.
- **Chapter 2—Trouble Shooting.** A composite of information taken from all of the chapters in this manual. It contains tables that summarize many problems, and their solutions, that you may encounter when porting a program.
- **Chapter 3—RISC/os Considerations.** Describes operating system dependencies that you need to be aware of when you are porting a C program from another operating system to a MIPS system.
- **Chapter 4—Hardware Related Considerations.** Discusses specific MIPS RISComputer implementations that you must consider when porting a program.
- **Chapter 5—Undefined Language Elements.** Explains how the RISCompiler System defines certain constructions in C, Pascal, PL/I, and other high–level languages that may differ in the program that you are porting.
- **Chapters 6—10**—Contains information specific to C, FORTRAN, Pascal, COBOL, RISCwindows, and PL/I programming languages.
- **Chapter 10—Programming Tools.** Discusses considerations for debugging, compiling, and link editing your programs.

For More Information

As you begin to port programs to our system, you may also need to refer to these books:

Book	Order Number
Assembly Language Programmer's Guide	*3201DOC*
Language Programmer's Guide	*3100DOC*
MIPS–COBOL Programmer's Guide and Language Reference	*3105DOC.*
MIPS–FORTRAN Programmer's Guide and Language Reference	*3103DOC*
MIPS–PL/I Programmer's Guide and Language Reference	*3107DOC*
MIPS R2000 RISC Architecture	*3113DOC*
RISC/os Programmer's Reference Manual	*3203DOC*
RISC/os User's Reference Manual	*3204DOC*
RISCwindows Reference Volumes I, II, and III	*3130DOC*

While porting a program, you will need to refer to the *MIPS Release Notes* that accompanied your RISC/os software. You should also have the publications listed below available for reference:

ANSI Pascal

ANSI Fortran

ANSI PL/I

Cobol Standard

IEEE 754–1985 (floating point)

Contents

1 Overview

The purpose of this chapter is to describe a process to follow when you are porting programs to the MIPS RISComputer environment.

1.1 Assembly Language Programs

This manual is a compilation of information that you need to port a C, Pascal, FORTRAN, or PL/I program to a MIPS RISComputer. This manual does not describe how to port assembly language programs. If you are thinking about porting an assembly program and your only reason for coding in assembly language is speed, seriously consider re–coding in a high–level language. Because it produces highly efficient machine language code for all supported high–level languages, the MIPS RISCompiler reduces the need for assembly language programming.

1.2 Making a Program Portable

You can make the task of porting programs easier by following the guidelines listed below when you create your program.

- Avoid breaking the rules of the source language and avoid using its non–standard features
- Avoid using source language that is machine dependent
- Avoid relying on anything that is operating system dependent
- When you have to do any of the things listed above, encapsulate them in modules marked "system dependent", use conditional compilation *#ifdef* statements around them, and explain the situation with plenty of comments

1.3 Information You Need to Know

Before you undertake a porting project, look over the following check list. Where applicable, cross–references are given for detailed information. Make sure that you are aware of each topic listed below, as you'll save time when porting your program.

☑ All programming languages:

How MIPS Makefiles work (**Chapter 1**)

What is the –G value (**Chapter 10**)

Floating Point differences (**Chapter 4**)

Differences in the address space organization (**Chapter 4**)

How the *#ifdef* conditional works (**Chapter 6**)

☑ C programming language:

How to use the lint program (**Section 6.2**)

How to use variable arguments (**Section 6.8**)

How to determine which *#defines* are appropriate (**Section 6.1**)

Characters unsigned by default (**Section 6.4**)

☑ Pascal programming language:

About the precision of type "real" (**Section 4.3**)

How MIPS RISComputer's single compilation process differs from yours

How memory allocation works (**Section 7.2**)

No two Pascals are the same, because the language has never been adequately standardized and the basic Pascal package is limited without adding extensions.

☑ FORTRAN programming language:

How static variables differ from automatic variables (**Chapter 8**)

If the size of your machine's double precision differs from MIPS RISComputer

☑ PL/I programming language:

The differences between the full ANSI PL/I and ANSI subset G PL/I

How MIPS RISComputer handles the nil value for PL/I (**Section 5.1**)

☑ COBOL programming language (for information on COBOL, see *MIPS–COBOL Programmer's Guide and Language Reference*):

Usage Optimization

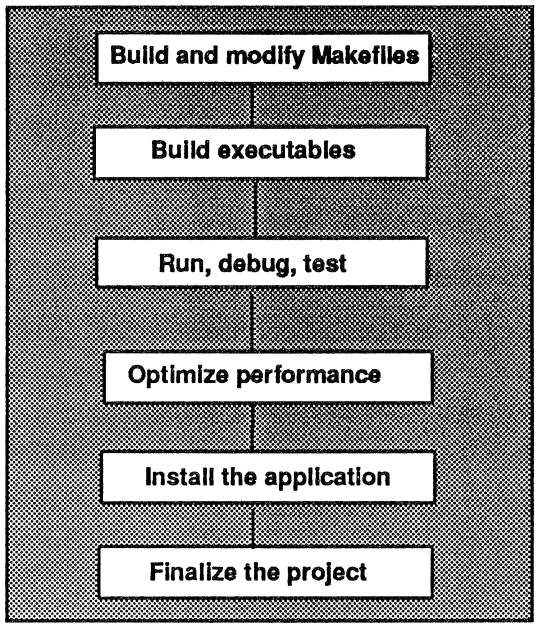
Packed Decimal Representation

Calling Separately Linked Routines

Sequential Files

1.4 Porting a Program

The following figure shows the major steps involved in porting an application to a RISComputer.



The following sections describe each of the above steps in detail.

1.4 .1 Build/Modify Makefiles

Large applications are accompanied by UNIX *Makefiles*, which contain the commands that build an executable program and which are processed by the the RISC/os **make** facility. This facility provides a method for maintaining up–to–date versions of programs that consist of a number of files that may be generated in a variety of ways. The Makefile is the description file through which the **make(1)** command keeps track of the commands that create files and the relationship between files. Whenever a change is made in any of the files that make up a program, the **make** command creates the finished program by recompiling only those portions directly or indirectly affected by the change.

For more information on Makefiles, refer to the **make(1)** manual page and **Chapter 13** in the *Languages Programmer's Guide*.

Before you can build your application, you need to modify the following parts of your Makefile:

- Include the path names for your source program and the include files that it uses.
- Include the path names for the link libraries. See **intro(3)** for a list of MIPS supported libraries.
- Include a path name for the directory where the application is to be accessed if the Makefile has an *install* target.
- Add compilation and link editor flags. See **cc(1)**, **f77(1)**, **pc(1)**, **ld(1)**, **cobol(1)**, and **pl/l(1)**.

If a Makefile does not exist for the application, and you need to create one, then refer to the MIPS Makefile standard in **Appendix B** of this book.

1.4 .2 Build Executables

To obtain a debugging version of your program that is not stripped or optimized:

1. Build the executable programs for the application by running the Makefile to compile and link the programs. Obtaining the correct driver option settings may involve modifying the Makefile several times.
2. Compile using the **–g** debugger option for full symbolic debugging.

If for some reason you wish to run your application before stripping the symbol table information and optimization, you should skip the above two steps, and wait until the step outlined in 1.4.4 below.

1.4 .3 Run the Application

When you run your application, more errors may occur. The trouble shooting guide in **Chapter 2** should help solve some of the run–time errors that you receive. If your program still has errors, then use debugging tools such as **dbx(1)** and **nm(1)**. Also, refer to chapters 3–5, and chapters 6–10, as applicable, for additional information on possible causes of errors.

Commercial applications usually include a test suite. This is the time to run the tests. If no tests are provided, then write and automate your own tests with shell scripts **sh(1)** or **csh(1)**.

1.4 .4 Optimize Performance

Once your application is debugged, tested, and working, then you should recompile the final version with the proper optimization level and link edit flags. For information on optimization, see **Chapter 10** in this manual, and **Chapter 4** in the *Language Programmer's Guide*.

All tests should be rerun at this point to further reduce the chance that machine, language, or operating system dependencies have not slipped through. If the application fails unexpectedly at this point, then you probably still have machine or language dependencies that the optimizer did not detect. If this is the case, then isolate the area causing the problem and repair it.

Refer to **Chapter 4** of the *Language Programmer's Guide* for a description of optimization techniques for your application. You may wish to profile your code (see **prof(1)** and **pixie(1)**) to see where additional performance gains can be made. Another useful tuning tool is **cord(1)**, which helps you improve cache performance.

1.4 .5 Install the Application

Once your application is tested and optimized, install it in the proper directory. You can install the program either by hand, or with the Makefile *install* target.

1.4 .6 Finalize the Application

The last step is probably the most tedious, but also the most important. If the user interface has changed at all, it is important to document the changes, not only in the code itself, but in the documentation. This is also the time to finalize the source code control system to manage the code. See **sccs(1)** and **rcs(1)**.

2
Trouble Shooting

This chapter is a composite of information taken from all of the chapters in this manual. It contains tables that summarize many problems, and their solutions, that you may encounter when porting a program.

Each table contains five columns, each of which contains a characteristic of the error; each column heading and a description of the information it contains is given below:

Column 1	*	When or how did the error manifest itself
Column 2	**	The source language(s) of the program most likely to create the error
Column 3		Symptom (general description of the error)
Column 4		Possible problem source
Column 5		Action (Recommended action to correct the error. Often, this is a cross–reference to another section in this manual.)

*	**	Symptom	Possible Problem Source	Action
L	A	Link edit fails	*gp* area overflow	Specify link editor **–G** *num* option . See **section 11.4.1.**
O	F	Output misformatted	VMS format extensions assumed	Specify –vms and recompile.
E	C	Memory Problems	Incorrect memory allocator	See **Section 6.3.**
E	P	Memory Problems	Incorrect memory allocator	See **Section 7.2.**
L	A	Invalid answers	Alignment problems	Specify –align8 or 16 and recompile. See **Section 4.3.**
B	A	Source files missing Library files missing	Library paths not correctly defined in makefile	See **Section 11.4.**
E	C	Segmentation violation	Dereferencing nil pointer	See **Section 4.2.**
E	A	Bus error	Alignment, data	Use **dbx** to locate, then correct.
E	A	Bus error/library routine	Alignment I/O	Copy data to aligned structure and recompile; allign all data structures.
E	A	Range of errors	Uninitialized values	Initialize–Fix code and recompile.
E	A	Range of errors	Wrong assumption nil value	See **Section 4.1.**
E	A	Out–of–range error	Floating point declaration	See **Section 4.3.**
E	A	Poor performance	Not optimized	See **Section 11.2.**
E	A	Program traverses directory tree and does not return to original place.	Use of *chdir()*	Avoid using relative paths .
E	A	Incorrect results	Order of evaluation	See **Section 5.2.**
E	A	Incorrect results	Uninitialized values	See **Section 4.6.**
E	F	Incorrect results	Assuming subroutine variables hold value between calls	Specify **–static** and recompile.
E	F	High system time	Uninitialized value	Specify **–static** and recompile.

*Error manifested itself during: B(build) C(compile) L(Link Edit) E(execution) or O (incorrect output)
**Can appear in which languages: A (all) F (Fortran) P (Pascal) Co (Cobol) C (C language) PL (PL/1)

*	**	Symptom	Possible Problem Source	Action
E	F	Bus error	Alignment problems	See **Section 8.2.**
C	F	Link edit fails	*gp* area overflow	See **Section 11.4** (General) and **Section 8.3** (Fortran).
E	A	Incorrect results	Replacing double precision for extended	Analyze why your program is using extended precision.
E	C	Degenerate unsigned comparison	Character variables pointers wrong	Use a mask or propagate bits. See **Section 6.4.**
L	A	*$gp* Area Overflow	Conflicting Devinitions of variables Large number of small variables	Relink using suggested **-g** *num.* See **Section 11.2.1.**
L	A	Undefined EXTERNAL, but external in library	Use of EXTERNAL occurs after scan of particular library	See **Section 11.2.2.**
L	A	*$gp* pointer not initialized	use of non–standard start–up code	See **Section 11.2.1.**
L	A	Unexpected undefined externals	Use of non–standard or machine specific library funcitons	Replace with RISC/os Equivalent functions.
L	F	Incompatible common block length warning	Program uses subject of common subroutines Different variable types declared in various subprograms	Assure full definition occurs in main routine. See **Section 8.3.**
L	F	Illegal initialization error	Common data initialized in two or more locations	See **Section 8.4.**
C	F	Numerous invalid/unrecognized statements	Program source in card format or uses over 72 columns	Use –col72 or –col120 switches.
C	P F C	Undefined conditional flag	Error in use of –D flag	See **section 6.1.**
L	C	Unexpected or undefined externals	Assumption compiler prepends _	See **Section 6.7.**
C	C	Illegal variable usage	use of TYPEDEF Name as an argument to function	See **Section 6.9.**

*Error manifested itself during: B(build) C(compile) L(Link Edit) E(execution) or O (incorrect output)
**Can appear in which languages: A (all) F (Fortran) P (Pascal) Co (Cobol) C (C language) PL (PL/1)

*	**	Symptom	Possible Problem Source	Action
E	A	Uninitialized data warnings	Not initializing variables	See **Section 4.6.**
E	PL	Wrong results	Wrong order in evaluation of expressions	Introduce temporary variables. See **Section 5.2.**
E	C	Wrong results	Program assumes that data types are of equal length	Use variable argument macros. See **Section 6.8.**
E	C	Compute–time error	Casting on the left hand side of an assignment	Don't do it. See **Section 6.11.**
E	F	Runtime error	Subscripts exceed the specified range.	See **Section 8.1.**
E	F	*$gp* area overflow	Uneven block sizes	See **Section 8..3.**
E	F	Uninitialized local variables	**–static** is repeated	See **Section 8.5.**
C	Co	Problem with truncation	Numeric data item (PIC 9) optimized into "USAGE COMP" by the compiler.	Use **–comptrunc.**
C	F	Illegal integer Illegal space Multiple defined symbols	Failure to initialize each named common block within one sub-program	Use the **fsplit** utility. See **Section 8.4.**
L	A	Cannot put a large variable into the *gp* area.	Inconsistent size declaration	See **Section 11.4.1.**
L	A	*gp* register is not initialized	You use your own start–up code	Use runtime start–up code. See **Section 11.4.**
L	A	Link editor fails to obtain correct libraries.	Did not use the language's default library and did not specify the user's library.	See **Section 11.4.2.**
L	A	Include files missing or errors in include files.	BSD sources	Use **–I/usr/include/bsd.**

*Error manifested itself during: B(build) C(compile) L(Link Edit) E(execution) or O (incorrect output)
**Can appear in which languages: A (all) F (Fortran) P (Pascal) Co (Cobol) C (C language) PL (PL/1)

*	**	Symptom	Possible Problem Source	Action
C	C	Illegal Identifier Name	Identifier with $ in name	See **Section 6.12.**
E	C	Segmentation or bus error	Improper assumptions with *malloc*	See **Section 6.13.**
E	A	Very long runtime	Denormalized FP values Algorithm terminates based on inappropriate FP values	Check for uninitialized values or reference of DP by SP variable or vice versa. See **Section 4.3.**
E O	F	Incorrect results, especially very small DP floating point answers	Equivalence making invalid assumption on storage size of variables	See **Section 8.2.**
L	A	Unable to satisfy externals when sub–procedure is in second language	Assumption about compiler external names; for example, _prefix, _postfix	See appropriate language user's guide.

*Error manifested itself during: B(build) C(compile) L(Link Edit) E(execution) or O (incorrect output)
**Can appear in which languages: A (all) F (Fortran) P (Pascal) Co (Cobol) C (C language) PL (PL/1)

3
RISC/os 4.0 Considerations

This chapter briefly describes RISC/os 4.0 and describes operating system dependencies that you need to be aware of when you are porting an application program from:

- BSD 4.3 to RISC/os BSD mode
- BSD 4.3 to RISC/os System V mode
- SystemV to RISC/os BSD mode
- System V to RISC/os SystemV mode

3.1 RISC/os 4 .0

RISC/os 4.0 is an AT&T System V 3.0–based kernel with BSD enhancements, including all BSD 4.3 system calls, BSD 4.3 library functions, most BSD 4.3 commands, TCP/IP networking, the NFS remote file system, and the Berkeley Fast File System.

RISC/os is packaged so that a user can concurrently access either BSD or System V commands. However, programmers are restricted to programming in either BSD or System V environments; mixed mode programming is not fully supported. UMIPS 3.0 permitted System V programs to use some BSD system calls. These system calls, as noted in Section 3.3.1, plus a few more are still available in RISC/os 4.0 for System V programmers.

3.2 Porting from BSD–Derived Systems

3.2.1 Compiling BSD Programs Under RISC/os 4.0

By default, RISC/os 4.0 is set up to compile programs under System V . In order to successfully compile programs for BSD functionality, you must do one of two things:

1. Use the compile time switch **systype bsd43** which prepends */bsd43* to your normal path.

```
% cc -systype bsd43 -g -o sample sample.c
```

2. Place */bsd43/bin* before */bin* in the PATH variable in your *.cshrc*, *.profile*, or *.login* file. When you compile, your system goes to the /bsd43 command directory and uses the BSD **cc** command which contains the switch **systype bsd43**.

If however, you want to compile a BSD program for in System V functionality and you have placed */bsd43* in your path prior to */bin*, you must use the **–systype sysv** switch. As in:

```
% cc -systype sysv -g -o sample sample.c
```

The default compile time switch for */bin/cc* is **–systype sysv** and the default compile time switch for */bsd43/bin/cc* is **–systype bsd43**.

3.2.2 Porting from 4.3 BSD to RISC/os (BSD based)

Several areas must be considered when converting a program from a regular BSD system to a RISC/os BSD system.

Include files Though textually different, the 4.3 BSD compilation environment include files are functionally equivalent to the 4.3 BSD include files.

The only differences between the two are in areas where the system cannot be made compatible. For example, the file */etc/utmp* does not contain the field *ut_host* and the include file that describes the *utmp* file (*/bsd43/usr/include/utmp.h*) contains a special marker that causes any code using the *ut_host* file to not compile. Such code must be changed.

Libraries All standard 4.3 BSD libraries are provided. However in some cases, the libraries use the corresponding System V code. For example, the libc routines that get password and group file entries have been copied from System V.

In the case of curses, the System V.3 curses (based on terminfo) is used. Except where the programs try to use the value of the buffer returned by the *tgetent()* function, this version of curses provides the entire 4.3 BSD interface.

Termcap Only terminfo is supported. In general, programs that use termcap and/or curses work as expected.

The features of termcap that are missing are:

- The ability to modify the termcap on the fly. This is often done to set the terminal size. This can be done by setting the window size with **winsize(1)** or by setting the environment variables LINES and COLUMNS. The former is the preferred method.
- The ability to add new capabilities to the database. This cannot be emulated without changing the code.

IOCTL commands Virtually all 4.3 BSD IOCTLs are supported on RISC/os when using the 4.3 BSD Compilation Environment. The only time you need to make any changes to your code is if your program does extensive tty manipulation. If this is the case, you should convert the tty handling to System V. For more information, see **termio(7)**.

Command functionality. If a program "exec"s a BSD command, then you should verify that the command exists as a BSD command; that is, it can be found in */bsd43/bin* and the "exec" command path should be changed accordingly. Otherwise, you should make sure that System V functionality is sufficient.

For a complete description of RISC/os system functionality, please see the *4.0 Release Notes* and check with your system administrator to verify that the BSD subpackage has been installed on your system.

3.2.3 Porting from 4.3 BSD to RISC/ os (SysV based)

Many programs written in C can be ported from BSD systems without changing any code by specifying compiler and link editor driver options as follows:

- During the compilation step, use the **–I/usr/include/bsd** and **–signed** options. The **–I/usr/include/bsd** option causes include files to be searched for in */usr/include/bsd* before */usr/include*, so that BSD values will take precedence, and the **–signed** option causes all char–typed data to be signed.

- During the link editor step, use the **–lsun**, **–lbsd**, and **–lrpcsvc** driver options to link in routines that are not part of the standard C library for System V, but which are needed by BSD and NFS programs.

Note: Use only the **–lsun** and **–lrpcsvc** driver option for programs requiring RPC and/or XDR.

3.2.4.1 Differences Between RISC/os SysV and BSD

This section describes some of the differences between RISC/os 4.0 and 4.3 BSD UNIX.

math.h Programs that use *libm* should be modified to include the library math.h. You cannot assume that the type of these functions under RISC/os 4.0 is the same as on other systems.

longjmp() If your program calls the function *longjmp()* from the signal handlers, it may need special work before being optimized with **–02**. This is because global variables may be placed in the registers and the values may not be restored properly. You may either explicitly declare the appropriate variable as volatile or use the **–volatile** compile option. Keep in mind that using this option significantly reduces the amount of optimization that is done. For a complete description of the **–volatile** option, see the *Language Programmer's Guide*.

Variable Arguments The typical mechanism for passing variable argument lists on BSD systems is to assume that a parameter is a pointer to an array of pointers; this does not work on MIPS machines. Instead you must use the vararg.h or stdarg.h macros. For a description of these macros, see **Appendix A** of the *Language Programmer's Guide*.

System Administration Files System administration files, such as */etc/passwd*, */etc/inetd.conf*, and the *utmp* file may differ from some applications. Some other files, such as */etc/ttys*, may be missing.

Dereferencing a Pointer Address 0 is an invalid address. On many BSD systems, this address is addressable; C programs may depend on being able to dereference pointers with this address. Dereferencing a pointer with a value of 0 is incorrect according to all C standards.

Pseudo–ttys Pseudo–ttys use a "clone device" instead of having pairs of pty/tty.

COFF Format The MIPS object file format (COFF) is a modified UNIX System V COFF and differs markedly from the BSD object file format. Therefore, BSD programs that process object file programs as input must be modified so as to access information correctly from the MIPS COFF. See **Chapter 10** of the *Assembly Language Programmer's Guide* for a description of the format and contents of the MIPS COFF.

tty Interface The tty drive interface does not have a complete emulation. Programs that rely heavily on the tty ioctls are difficult to port.

Load Average The "avenrun" (load average) kernel symbol contains items of type FIX, as defined in *sys/fixpoint.h*, not doubles.

malloc() On Apollo systems, there is a system call to pre–allocate memory for *malloc()*. This call is not supported and is not needed on MIPS machines. For additional information on additional malloc() library calls, see **Chapter 11** in this manual.

3.3 Porting from System V–Derived Systems

A program that runs on System V will port more easily to RISC/os if you include *math.h* for math library functions. Do not assume that the type of these functions is the same as on other systems.

libbsd.a

The library */usr/lib/libbsd.a* is a System V library provided by MIPS which contains some 4.3 bsd system calls and library routines. Because of file sizes when the library was introduced, the routines in this library have been renamed to approximate their 4.3 bsd routine names. For example, the *getdomnm* in *libbsd.a* is *getdomainname* in the 4.3 bsd *libc.a* library.

Note: All *yp* routines, such as yp_bind, yp_first, yp_match, yp_next, yp_order, ypclnt, yppasswd, and so on, have been removed from *libbsd.a* because yellow pages are not supported for RISC/os 4.0. You must provide your own stubs for these routines. Two additional compatibility libraries are also provided, */usr/lib/librpcsvc.a* and */usr/lib/libsun.a*, for network applications. If you find that a source file no longer compiles with *libbsd.a*, include *librpcsvc.a* and/or *libsun.a* in the link step.

Table 3.1 lists the *libbsd.a* routines, the BSD *libc.a* names, and gives an explanation of the routine.

Table 3.1 libbsd.a Routines

libbsd.a file name	4.3 BSD libc.a file name	Description
accept 2–BSD		Accepts a connection on a socket
bcopy, bcmp, bzero, ffs 3–BSD		Bit and byte string operations
bind 2–BSD		Bind a name to a socket
connect 2–BSD		Initiate a connection on a socket
dbm_open, dbm_close, dbm_fetch, dbm_store, dbm_delete, dbm_firstkey, dbm_nextkey, dbm_error, dbm_clearerr 3		Data base subroutines
getdomnm (2–BSD)	getdomainname, setdomainname	Get/set name of current domain
getdtabsz (2–BSD)	getdtablesize	Get descriptor table size
gethostid, sethostid (2–BSD)		Get/set unique identifier of current host
gethostnamadr (3N–BSD)	gethostbyname, gethostbyaddr, gethostent, sethostent, endhostent	Get network host entry
gethostname, sethostname	2–BSD	Get/set name of current host
getnetent, getnetbya, getnetbynm	getnetbyaddr, getnetbyname, setnetent, endnetent 3N–BSD	Get network entry
getpeernm (2–BSD)	getpeername	Get name of connected peer
getprotoe (3N–BSD)	getprotoent, getprotobynumber, getprotobyname, setprotoent, endprotoent	Get protocol entry
getrlimit, setrlimit (2–BSD)		Control maximum system resource consumption
getrusage (2–BSD)		Get information about resource utilization

Table 3.1 libbsd.a Routines (cont.)

libbsd.a name	libc.a name	Description
getservent, getservbyport, getservbyname, setservent, endservent (3N–BSD)		Get service entry
getsocknm (2–BSD)	getsockname	Get socket name
getsockopt, setsockopt (2–BSD)		Get and set options on sockets
gettimeofday, settimeofday (2–BSD)		Get/set date and time
getwd (3–BSD)		Get current working directory pathname
htonl, htons, ntohl, ntohs (3N–BSD)		Convert values between host and network byte order
in_addr (3N–BSD)	inet_addr	Internet address manipulation routines
in_lnaof (3N–BSD)	inet_lnaof	Internet address manipulation routines
in_mkaddr (3N–BSD)	inet_mkaddr	Internet address manipulation routines
in_netof (3N–BSD)	inet_netof	Internet address manipulation routines
in_network (3N–BSD)	inet_network	Internet address manipulation routines
in_ntoa (3N–BSD)	inet_ntoa	Internet address manipulation routines
insque, remque (3–BSD)		Insert/remove element from a queue
listen (2–BSD)		Listen for connections on a socket
opendir, readdir, telldir, seekdir, rewinddir, closedir (3–BSD)		Directory operations
random, srandom, initstate, setstate (3–BSD)		Better random number generator; routines for changing generators
rcmd, rresvport, ruserok (3–BSD)		Routines for returning a stream to a remote command
recv, recvfrom, recvmsg (2–BSD)		Receive a message from a socket
rexec (3–BSD)		Return stream to a remote command
rindex (3–BSD)		string operations
scandir, alphasort (3–BSD)		scan a directory

Table 3.1 libbsd.a Routines (cont.)

libbsd.a name	libc.a name	Description
select (2–BSD)		Synchronous I/O multiplexing
send, sendto, sendmsg (2–BSD)		Send a message from a socket
setregid (2–BSD)		Set real and effective group ID
setreuid (2–BSD)		Set real and effective user ID's
setuid, seteuid, setruid, setgid, setegid, setrgid (3–BSD)		Set user and group ID
shutdown (2–BSD)		Shut down part of a full–duplex connection
socket (2–BSD)		Create an endpoint for communication
syslog, openlog, closelog, setlogmask (3–BSD)		Control system log

3.3.1 RISC/os Differences

This section describes some differences between RISC/os 4.0 and regular System V UNIX.

Symbolic Links The presence of symbolic links can cause problems if you use the UNIX function *chdir()*. Programs should avoid using relative paths to change directories, because the sequence

```
chdir("./subdir"); chdir("..")
```

may not set the directory back to the original place.

File Name Size The maximum size of a file name is 255 characters, not 14. This causes problems when programs expect truncation. For example, a program could reject a user–supplied filename that is larger than 14 characters because it assumes that 14 characters is the limit. In addition, if a program declares an array to hold a filename, the array may be too small, especially if it is declared to be 15 characters long. MAXNAMLEN in */usr/include/dirent.h* and MAXPATHLEN in */usr/include/sys/nami.h* are appropriate to use instead.

Directory Access The MIPS file system does not allow your program to directly read a directory. Programs that do this are considered to be non–portable to RISC/os even though they may work on many versions of System V UNIX.

longjmp() Programs that call the function *longjmp()* from the signal handlers may need special work before being optimized with **–O2**, since global variables may be placed in the registers and the values may not be restored properly. Declaring appropriate variables as volatile solves this problem. To get around this problem, use the **–volatile** compiler option, which causes all objects to be treated

as volatile. Keep in mind that using this option significantly reduces the amount of optimization that is done.

COFF Format Programs that work with object and executable files need some work, as the MIPS COFF format is not completely compatible with System V.

Dereferencing a Pointer Address 0 is an invalid address. On many BSD systems, this address contains a 0, and C programs may depend on being able to dereference pointers with this value. Dereferencing a pointer with 0 is incorrect according to all C standards.

3.4 Dereferencing nil Pointers

Programs that contain errors sometimes go undetected on machines where dereferencing a zero pointer yields zero. Typically, the programmer meant to write:

```
int *c;
if (c != 0 && *c) ...;
```

but actually wrote:

```
int *c;
if (*c) ...
```

On most VAX UNIX systems, the error goes undetected; on most MC68000 implementations and on the MIPS RISComputer systems, this causes a segmentation violation.

3.5 Where Text and Data Lie in Memory

Figure 4.1 illustrates how text and data are arranged in memory on MIPS machines and VAX machines. Refer to this figure as you read the paragraphs that follow it.

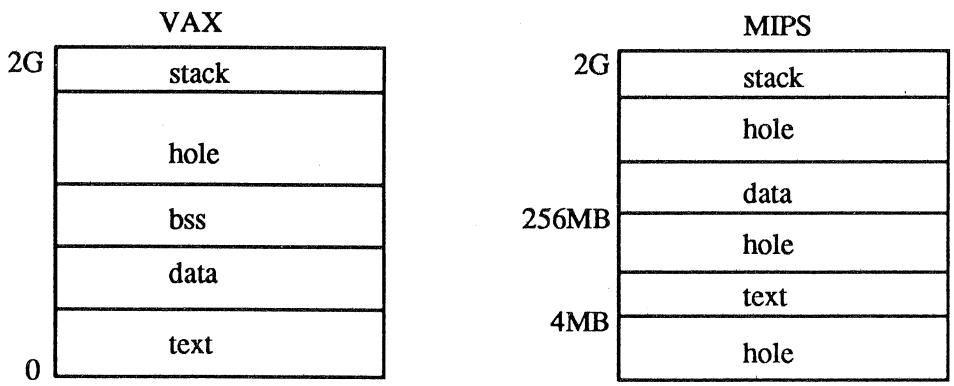


Figure 4.1. A comparison of VAX and MIPS address space

You may encounter problems when you port a program if you assume that the link editor–defined symbol *etext* indicates the beginning of the data section as well as the end of the text section. As illustrated in Figure 4.1, *etext* would work on a VAX because data and text are located next to each other. However, on a

MIPS machine they are not. To solve this problem, the MIPS link editor provides other symbols for the beginning and end of each text section; for more information see the **end(3)** manual page.

Some sophisticated UNIX programs such as GNU's emacs assume that the program text (that is, the executable code) or data starts at a low address in memory. The program can for example, store tag information in the high–order bits of a pointer and mask out the tag just before dereferencing the pointer. Unless you specify otherwise with the link editor **–T** and **–D** options, program text on RISComputer systems starts at approximately 0x400000, program data approximately 0x10000000, and the program stack at approximately 0x80000000.

3.6 Porting from Other Operating Systems

This section discusses the issues that you need to be aware of when porting programs from systems other than UNIX.

3.6.1 General

In general, if you are porting from operating systems other than UNIX, you need a working knowledge of both RISC/os and the operating system that you are porting from.

Programs written with no operating system dependencies should port easily. Such programs use the standard I/O routines for the language in which they are written rather than using UNIX system calls are more likely to port with little or no modification. C programs that follow the ANSI C Standard should work as expected.

3.6.2 Porting FORTRAN Programs from VAX

Library functions that provide an interface to RISC/os 4.0 (similar to those provided by the C library) are available to MIPS–FORTRAN programs. Also, intrinsic subroutine and functions used to interface VAX systems are available to provide the same functional interface to RISC/os from MIPS–FORTRAN programs. **Chapter 4**, Part I, of the *MIPS–FORTRAN Programmer's Guide* and Language Reference describes these system functions and subroutines.

4
Hardware–Related Considerations

This chapter discusses specific MIPS RISComputer implementation that you need to consider when porting programs.

4.1 Floating Point Arithmetic

In 1985, ANSI/IEEE 754–1985 defined a standard floating point representation and arithmetic. MIPS RISComputers conform to this standard. While you might expect conformance to this standard to eliminate problems with porting floating point programs, there are still significant differences that can hinder an implementation's portability.

Floating point differences manifest themselves by:

- Producing slightly different results
- Producing incorrect results
- Slow execution
- Faulting

4.1.1 General IEEE 754

Table 4.1 lists the IEEE floating point format. The explicit use of extended precision formats available on some IEEE 754 floating point implementations makes programs non–portable, because there is no simple or efficient way to get the range or accuracy of IEEE extended on a machine whose highest precision is double. To avoid this problem, try using double precision instead; however, using double may give you incorrect results. Therefore, before you substitute double for extended, analyze why your program is using extended.

Some compilers use extended precision even when your program does not specify it. This is because some hardware such as Motorola 68881, 68882, and Intel 80387, makes this the only efficient thing to do. When an expression is evaluated using extended precision, you may get a slightly different answer than if it were evaluated in double precision.

The implementation of library math functions differs from machine to machine, so you will see slightly different results when you run programs on the MIPS RISComputer.

Table 4.1 IEEE Floating Point Format

Format	Size	Radix	Approximate Range	Rounding	Exceptions
single format	32 bit	2 with 24 bits of precision	10^{-45} to 10^{38}	round to nearest, round .5 to even	for overflows, divide by zero, and so on, do not fault, but instead return special symbols
double format	64 bit	2 with 53 bits of precision	10^{-324} to 10^{308}	same as above	same as above
extended	80 + bit	2 with 64 bits of precision	10^{-4951} to 10^{4931}	same as above	same as above

4.1.2 DEC VAX

Table 4.2 lists the DEC VAX floating point format. If you are porting a program that uses VAX floating point to a MIPS RISComputer, which uses IEEE floating point, keep in mind that the IEEE floating point format is much more likely to cause problems between single– and double–precision in loosely typed languages like FORTRAN.

For example, the two VAX single– and double–precision formats are identical except that the double–precision format provides additional precision. Therefore, if you reference something that is single that is really double, it has little effect on the value. Because IEEE 32–bit and 64–bit formats are different, if you make the same mistake on a RISComputer, you could produce data that does not even resemble the data produced from the original machine.

Use of H–format (REAL*16) is non–portable to RISComputers, because their floating point does not have the range or accuracy of H–format. Using IEEE double precision will likely give you incorrect answers.

The default double precision, D–format, has more precision but less exponent range than IEEE double, thus precision–sensitive programs may give different results.

Table 4.2 VMS Floating Point Format

Format	Size	Radix	Approximate Range	Rounding	Exceptions
F–format	32 bits	2 with 24 bits of precision	10^{-38} to 10^{38}	round to nearest round .5 and up	for overflows, divide by zero
D–format*	64 bits	2 with 56 bits of precision	10^{-38} to 10^{38}	same as above	same as above
G–format *	64 bits	2 with 53 bits of precision	10^{-308} to 10^{307}	same as above	same as above
H–format	128 bits	2 with 112 bits of precision	10^{-4933} to 10^{4931}	same as above	same as above

* F–format is similar to IEEE single format and G–format is similar to IEEE double format.

4.1.3 IBM 370

Table 4.3 lists the IBM 370 floating point formats. Programs that depend on the larger single–precision exponent range are non–portable. MIPS RISComputers generally provide better accuracy, and therefore different results.

Table 4.3 IBM 370 Floating Point Format

Format	Size	Radix	Approximate Range	Rounding
single format	32 bits	16 with 6 radix–16 digits precision	10^{-73} to 10^{75}	chopped
double format	64 bits	16 with 14 radix–16 digits precision	10^{-73} to 10^{75}	same as above
Real *16	128 bits	16 with 30 radix–16 digits precision implemented in software	10^{-73} to 10^{75}	same as above

4.1.4 Cray

Figure 4.4 lists the Cray floating point format. Because Cray's single–precision is a 64–bit format, it is generally necessary to switch to MIPS double precision to get the same results. Also, if your program depends on a large exponent range or 128–bit precision, program modifications are required. MIPS RISComputers provide better accuracy than Cray's 64–bit format, and therefore different results occur.

Table 4.4 Cray Floating Point Format

Format	Size	Radix	Approximate Range	Rounding
single format	64 bit	48 bits of precision	10^{-2460} to 10^{2460}	chopped or worse
double format	128 bit	95 bits of precision implemented in software	10^{-2460} to 10^{2460}	chopped or worse

4.3.5 Math Library Accuracy

Besides basic floating point format and accuracy issues, each implementation typically differs in the algorithms and characteristics of its math library, Even IEEE 754–1985 machines that are otherwise identical may produce different results due to differences in math libraries. See **math(3M)** for additional information on the MIPS math library. The algorithms are generally from *Cody and Waite*, with some additions and replacements from 4.3 BSD.

4.2 Endianness

Machines that number the bytes from right to left and the least significant byte is zero, within a 32–bit integer are called *little–endian*; machines that number the bytes from left to right and the least significant byte is 3 are called *big–endian*. See **Appendix D** and Byte Ordering options in **Chapter 1** of the *Language Programmer's Guide* for more information on these byte ordering schemes. Although the MIPS R2000 chip can operate either way, MIPS RISComputer systems use the big–endian byte ordering scheme.

You may create porting problems by placing small objects side by side to make a bigger object, or splitting a big object into small objects. For example, the following code that reads and compares a pair of shorts is machine–dependent, because on some machines the 0th element of the array represents the high–order half of the word rather than the low–order half:

```
char carray[BUFSIZ];
err = read(0, carray, 4);
if ((carray[0] | (carray[1] << 8)) >
(carray[2] | (carray[3] << 8))) ...
```

There is never a problem if you use the correct data type and let the compiler deal with the order of the bytes:

```
short sarray[BUFSIZ];
err = read(0, (char *) sarray, 2 * sizeof(short));

if (sarray[0] >sarray[1]) ...
```

Similarly, the following code to print four characters stored within an integer is machine–dependent because it assumes the first character is at the low–order end of the integer:

```
unsigned i;
printf("%c%c%c%c\n", i & 0xff, (i >>8) & 0xff, (i
>>16) & 0xff,
(i >>24) & 0xff);
```

A better solution is:

```
unsigned i;
printf("%.4s\en", (char *) &i);
```

4.3 Alignment

RISComputer architecture requires that each piece of data in memory be aligned on a boundary appropriate to its size. For example, an n–byte integer can be aligned on a boundary whose address is a multiple of n–bytes, up to a maximum of 8 bytes. This restriction permits the memory system to run much faster.

Ordinarily alignment has no effect on correctly written programs, because the compiler inserts unused space (''padding'') between variables wherever necessary to conform to the rules. Language standards almost always permit such padding, and in the rare cases where the language forbids it, the compiler conforms to the language requirements by loading and storing objects in special ways (see **Section 8.2** in this manual for information on how this applies to FORTRAN programs).

However, a program that follows the rules of its language usually doesn't encounter problems. To avoid alignment problems, declare the fields of a structure in descending order by size.

Even a program that follows the rules given above may have trouble when writing data on one machine and reading it on another. In fact, padding is only one of many problems: machines differ with regard to endianness, floating–point formats, the size of the integer, and the width of a character or word. There are two collective solutions to these problems: if I/O speed is not important, use ASCII files rather than binary ones; otherwise, consider using the **xdr(3N)** subroutine package for external data representation.

See **Section 8.2** in this manual for a discussion of the extensions to the compiler system for dealing with misaligned data as they apply to FORTRAN programs.

You may choose one of the following three command–line arguments to deal with various degrees of misalignment:

–align8 Permits objects larger than 8 bits to be aligned on 8–bit boundaries. This option requires the greatest amount of space; however, it is the most complete solution; 16–bit padding is not inserted for **integer*2** objects within common blocks.

–align16 Permits objects larger than 16 bits to be aligned on 16–bit boundaries; 16–bit objects must still be aligned on 16–bit boundaries (MC68000–like alignment rules); 16–bit padding is not inserted for **integer*2** objects within common blocks.

–align32	Permits objects larger than 32 bits to be aligned on 32–bit boundaries; 16–bit objects must still be aligned on 16–bit boundaries, and 32–bit objects must still be aligned on 32–bit boundaries. This option requires the least amount of space, but isn't a complete solution; 16–bit padding is inserted for **integer*2** objects within common blocks.

4.4 Uninitialized Variables

Whenever possible, initialize local variables. The **lint(1)** C program checker and the RISCompilers issue warning messages about uninitialized data in certain instances. However, because the system can see only the static characteristics of a program, it cannot warn about all instances of uninitialized data.

If your program's failures vary with the input data, but the variances are not logically related to the failing code, look for uninitialized variables.

In addition, if your program works when compiled with the default **–O1** optimization, but fails when compiled with **–O2** optimization, then the fault may be caused by uninitialized variables rather than the optimizer. In an **–O2** optimization, the optimizer may allocate an uninitialized variable to a register, creating an error that would not have occurred in an **–O1** optimization.

On the MIPS RISComputer, uninitialized variables can degrade the performance of a program that otherwise runs correctly. The hardware performs most IEEE operations, but software is invoked for operations on denormalized numbers. If, in performing computations on an uninitialized floating–point variable, an uninitialized variable happens to be a denormalized IEEE value, then the algorithm in your program could continue to function properly even with a non–zero variable, provided it remains close to zero. This situation could deteriorate the performance and accuracy of your program.

If you suspect this problem, use the **time(1)** command. For most programs, the system CPU time is small compared to the user CPU time. If the system time is unexpectedly high but not high enough to account for the overall slowdown, that's a good indication of denormalized arithmetic. The system time does not account for the entire slowdown, because not all of the emulation time is charged against your program.

Another aid in diagnosing these problems is the **fpi(3)** floating point interrupt analyzer. The **fpi** routines count the instances of floating–point emulation and print a summary.

5
Undefined Language Elements

Language standards deliberately avoid defining certain language constructs, thus causing inconsistencies among different implementations of the same language. This section explains how the RISCompiler system defines some of these constructs, which you may need to alter in the program being ported.

5.1 The Value of nil

C, Pascal, and PL/I do not specify the value that the compiler must use to represent a nil (or "null") pointer. However, C does dictate that the compiler must recognize a zero in the source program as the notation for a nil pointer and convert it into whatever value does represent nil.

The MIPS RISCompiler system uses zero to represent "nil". Few UNIX programs encounter any difficulty with this, but other operating systems use other values like "–1" or "– maxint – 1". A portable program shouldn't depend on this value, but for convenience the MIPS PL/I compiler does recognize an option that permits you to change the value:

```
pl1 -Wf,-setnull,-1 ...
```

5.2 Order of Evaluation

The order in which program statements are evaluated can cause problems as shown below.

For example, the expression in the following Pascal statement can cause trouble if the programmer hoped that it would invoke the decrement function on both variable x and variable y:

```
if (decrement(x) < 0) and (decrement(y) < 0) then
...
```

As another example of the side effects, neither language specifies the order in which the compiler evaluates an actual argument list:

```
foo(decrement(x), x+ y);
```

Another example is the C statement:

```
foo(*p++, *p++);
```

The best way to control the order of evaluation in a program being ported to a RISComputer system is to introduce temporary variables. Because of global optimization, this usually costs nothing (apart from forcing the intended order and degree of evaluation), because the compiler attempts to allocate all of the objects to registers:

```
temp1 = decrement(x);
temp2 = decrement(y);
if (temp1 < 0 and temp2 < 0) then ...

temp1 = decrement(x);
call foo(temp1, x+ y);
```

5.3 Inter–Language Interfaces

The allocation of variables in memory, the rules of argument–passing, and the mapping of source–language identifiers onto assembly–level symbols all pose problems that appear when you stop programming entirely within one language and start calling routines written in another language.

For example, Pascal specifies that the **ord** function must return zero for a *false* boolean and one for a *true* boolean; but Pascal does not specify whether a boolean value is stored in memory as a single bit, a byte, or a full word. In fact, Pascal permits a compiler to implement *true* by setting the sign bit of a word, or even by setting all bits to 1, provided the **ord** function performs the appropriate conversion. As long as you program entirely in Pascal, you need never know these details, but when Pascal code passes a boolean to a C subroutine, the latter must know whether to expect a **char**, a **short**, or an **int**, and what value constitutes *true*.

For information on interfaces between C and Pascal programs, see **Chapter 4** in the *Language Programmer's Manual.* For information on the interfaces between FORTRAN and C programs, and FORTRAN and Pascal programs, see the *MIPS–FORTRAN Programmer's Guide.*

6
C Programming Language

MIPS C conforms to the *de facto* standard established by the Kernighan and Ritchie text and the AT&T portable C compiler. It provides certain extensions, such as prototype declarations, suggested by the draft ANSI C standard. See **Appendix A** in the *Language Programmer's Guide* for more information on C extensions.

6.1 Using the C Preprocessor

To maintain your program on both an old system and on the RISComputer system, consider using the **#ifdef** conditional–compilation facility provided by the **cpp** preprocessor. The C and Pascal RISCompilers provide this feature by default; the FORTRAN compiler provides it if you either use the **–cpp** driver option, or give your source file a name ending in *.F* rather than *.f*. Using **cpp**, you can include the following conditional statements in your program:

```
#ifdef MY_OLD_MACHINE
x := #ff5a;
#endif /* MY_OLD_MACHINE */
#ifdef MIPS
x := 16#ff5a;
#endif /* MIPS */
```

Then, you can use the **–D** option to select the appropriate version. For example, to generate a MIPS–specific version of a Pascal program, you would specify:

```
pc -DMIPS myprog.p -o myprog
```

To translate *myprog.p* into a source file *myprog.i* suitable for compilation on your old machine, you would use the **–P** option as follows:

```
pc -P -DMY_OLD_MACHINE myprog.p
rcp myprog.i my_old_machine:myprog.p
```

On most machines, including RISComputers, the **–D** option is unnecessary if you use a name that is automatically defined for you. MIPS compiler drivers predefine the following automatically:

```
mips
host_mips
MIPSEB
MIPSEL
LANGUAGE_C
LANGUAGE_PASCAL
LANGUAGE_FORTRAN
```

```
LANGUAGE_ASSEMBLY
LANGUAGE_PL1
LANGUAGE_COBOL
unix
SYSTYPE_BSD
SYSTYPE_SYSV
```

Note: Typically, you use *#ifdef mips* for differences that are hardware related or os related and *#ifdef MIPS* for differences due to other programs or preferences.

6.2 Using the Lint Program Checker

The **lint** program checker tries to find areas in the source code of C programs that are unportable or that are likely to cause errors. See the **lint(1)** manual page in the *User's Reference Manual* for reference information. Here are some guidelines to follow when using **lint**:

- Instead of running **lint** on your source files one by one, run it a single time, specifying the names of all the source files. The **lint** command detects such problems as argument–list mismatches more thoroughly when it processes the entire source program at once.
- Use the same **–D** and **–I** options (if any) that you specify when you compile.
- Analyze **lint** error or warning messages carefully before changing your code; make sure you understand why **lint** is creating the errors. For example, suppose **lint** indicates that a function result is incompatible with its use:

```
double d;
d = atof("1.23");
```

You could satisfy **lint** by putting a cast in front of the function call:

```
d = (double) atof("1.23");
```

but in fact you would be masking the problem rather than fixing it. The correct solutions are to either include *math.h* in your program or declare *atof* so that the compiler knows that it returns a **double** value rather than an **int** as follows:

```
double d;
extern double atof();

d = atof("1.23");
```

6.3 Memory Allocation

The interface to the C library memory allocator **malloc** is standard, but the implementation varies. RISC/os 4.0 uses the 4.3 BSD **malloc**, rather than the System V.3 **malloc**, because the former is significantly faster.

BSD **malloc** allows for allocation errors in that it rounds up the requested block size to a power of two, thus making programs that write more than they allocate work. While the power of true allocation is fast, it is inappropriate for large data block sizes.

Note that UNIX memory allocators use more memory than the programs request. If you plan to allocate memory in large chunks and never free them during execution, consider using **sbrk(2)**.

If you suspect a problem caused by memory allocation, try a different allocator and see if the problem disappears or changes. UMIPS provides the following memory allocators in addition to the standard **malloc** version:

- an optional **malloc**, which you can obtain by specifying the **–lmalloc** option during compile/link edit.
- an additional allocator with routines **xmalloc**, **xfree**, and **xrealloc** resides in */usr/lib/libp.a* (in release 1.31 or earlier, specify **–lxmalloc**). You can allocate the routines using the **–lp** option during compile/link edit. This allocator's interface is identical with that of **malloc**, **free**, and **realloc**.

Even if using a different memory allocator solves the problem, you should still fix it to prevent a recurrence. Here are some approaches you can take:

1. Replace all calls to **malloc** and **realloc** with a wrapper routine that initializes the newly–allocated block (or the yet–unused portion of the reallocated block) to zero. If the problem disappears, look for code that erroneously assumes that newly allocated memory is initialized to zero.
2. Replace all calls to **malloc** and **realloc** with a wrapper that calls those routines, allocating one more byte than you ask for. If the problem disappears, this experiment may hide the problem by altering the order of blocks in memory. It is also likely that (in Pascal or Fortran) the program is confused about whether a character array originates at 0 or 1, or that (in C) the program did not leave space for the "null" byte that terminates a string.
3. Replace all calls to **malloc** and **realloc** with a wrapper that calls those routines, allocating four or eight more bytes than you ask for. If the problem disappears, then a zero–origin problem with an integer, real, or double–precision array exists.

4. Experimentally replace all calls to **free** with an empty routine. If the problem disappears, the experiment may have masked the true problem by rearranging blocks in memory. However, dangling pointers to reused space may be causing the problem. Make sure that the program does not retain pointers to any data structure whose address may change due to a call on **realloc**.

6.4 Signed chars

Like AT&T 3B compilers, but unlike most VAX and MC68000 compilers, the MIPS RISCompiler System interprets **char** to mean **unsigned char**. The **–signed** command–line option, however, reverses this.

To understand the consequences of unsigned characters, consider that the character *0xff* is not the same as *–1*; and a loop like

```
char c;
for (c = '\ '; c >= 0; c--) ...;
```

never terminates because the variable c can never be negative.

The MIPS C compiler, and others that have adopted features of the proposed ANSI draft standard, permit you to specify either **signed char** or **unsigned char** explicitly in a declaration. Alternatively, you can use masks or shifting to eliminate or propagate bits.

Lint detects such problems by printing the diagnostic message *degenerate unsigned comparison.*

6.5 Bitfields

For a bit field declaration within a structure, the MIPS C compiler uses signed or unsigned bitfields depending on your declaration. The Kernighan and Ritchie definition of the language permits a compiler to ignore these attributes and always use signed arithmetic or always use unsigned arithmetic; some compilers take advantage of this.

6.6 Short, Int, and Long Variables

On a RISComputer system, a **short** variable is 16 bits wide; an **int** variable is 32 bits wide; and a **long** variable is also 32 bits wide. Some microcomputer compilers allocate only 16 bits for **int** and 8 bits for **short**, and some programs may rely on this. In general, manipulating 32–bit objects with the RISComputer architecture is as fast as or faster than manipulating 16–bit objects.

6.7 Leading "_"

Like AT&T 3B compilers, and unlike the BSD UNIX VAX compiler, the MIPS C compiler system doesn't prepend an underscore to the name of a C–compiled symbol.

6.8 Varargs

To improve performance, RISCompilers pass certain procedure arguments in registers. This process is normally transparent to you, except for functions that use variable–length argument lists. These lists must use the macros provided in */usr/include/varargs.h* or */usr/include/stdarg.h*. The functions must not assume that the arguments all appear in memory and can be accessed by taking the address of the first argument and incrementing it. Both the ANSI draft standard and the Kernighan and Ritchie definition of the language warn that programs attempting to implement variable argument lists without using varargs may not be portable.

Even *varargs* cannot deal with a situation where the argument list varies in type as well as length. Consider the following rather common practice of assuming that all C data types are equivalent for purposes of parameter–passing:

```
error(s, a, b, c, d, e)
char *s;
int a, b, c, d, e;
{
fprintf(stderr, s, a, b, c, d, e);
}

double d;

error("Value %g should be between %g and %g\en",
d, 1.2, 6.5);
```

The problem with this routine isn't that the variable argument list is variable, but rather that the routine declares arguments *a* through *e* as integers when in fact the routine plans to supply floating point numbers. This violates both the Kernighan and Ritchie definition of the language and the ANSI draft standard. In addition, it has dire consequences, because RISComputer architecture uses two separate sets of registers to pass **integer** and floating point arguments, and because it imposes rules on the alignment of data types. The **fprintf** can accept variably typed arguments because it determines the types at execution time and references them appropriately; but the routine in the above example tells the compiler to emit a single version of ''error'' that always references them all as type **int**.

The following code fragment is the best method to use for a program being ported to MIPS–C. Any other system that implements *fprintf* using a routine called *vprintf* can also use this routine system:

```
/* VARARGS 1 */
void
error(s, va_alist)
char *s;
va_dcl
{
va_list ap;
va_start(ap);
vprintf(s, ap, stderr);
fputs("\en", stderr);
exit(1);
}

error("Value %g should be between %g and %g\en",
d, 1.2, 6.5);
```

However, a solution using a macro would make this routine more portable:

```
#define error(_s,_a,_b,_c,_d,_e) \e
fprintf(stderr, _s, _a, _b, _c, _d, _e); \e
exit(1);

error("Value %g should be between %g and %g\en",
d, 1.2, 6.5, 0, 0);
```

6.9 typedef Names

ANSI C provides prototypes that in one instance conflict with Kernighan and Ritchie usage. ANSI C makes it illegal for a typedef name to appear in the argument list for a function definition. For example, in the following code:

```
typedef int P;
function(P);
{
}
```

the occurrence of P in the argument list is illegal since the compiler expects an identifier after the type 'P'. MIPS C conforms to the ANSI standard in this case.

6.10 Functions Returning Float

Functions that are declared as returning **float** actually return **float** rather that double as in some older implementations of C. If the result is then used in a context requiring promotion to double; it is promoted after returning from the function call.

6.11 Casting

Casting is not permitted on the left hand side of an assignment. If you are porting a program that currently runs on a Sun Workstation, you may have problems with this because Sun allows casting on the left hand side.

6.12 Dollar Sign in Identifier Names

The dollar '$' sign is not a legal character in an identifier name. Because VAX and Sun compilers allow you to use '$' as a legal character, MIPS provides the command line argument:

```
-Wf, -Xdollar
```

6.13 Additional Keywords

MIPs will eventually conform to the ANSI C standard; therefore, the compiler treats *const*, *signed*, and *volatile* as keywords.

6.14 alloca()

The current RISCoperating system does not support *alloca()*.

6.15 Unsigned Pointers

MIPS RISCompiler treats pointers as unsigned rather than signed integers. For example, the following code:

```
extern  char  * sbrk();
char *p
p = sbrk(4090)

   if (p < 0) error("out of memory");
```

does not work as expected because MIPS RISCompilers use unsigned pointer comparisons, and nothing unsigned is less than zero. The *sbrk* routine does not work because it returns –1 if it fails. The proper way to test for failure is:

```
if (p == (char *) -1) error("out of memory");
```

7
Pascal Programs

MIPS Pascal conforms to the IEEE standard, which is similar to the original Wirth–Jensen report, rather than to the ISO standard. It also provides a number of extensions, but not UCSD string support or ISO conferment arrays. See **Appendix B** in the *Language Programmer's Guide* for more information on Pascal extensions.

If you wish to maintain your program on both an old system and on the RISComputer system, refer to **Section 6.1** in this manual.

7.1 Runtime Checking

When possible, compile your program with runtime–checking using the –C option, which generates code that checks that subscripts don't exceed the range specified for them in the program. Storing one byte past the end of an array of characters may be harmless on one system if the compiler decides not to use the byte for anything but causes an execution error on another system if a compiler decides to store something such as a subroutine return address there.

7.2 Pascal Dynamic Memory Allocation

The MIPS Pascal compiler responds to the much–requested dynamic allocation extensions to IEEE Pascal. The compiler provides a new generic data type, **pointer**, which is type–compatible with any standard Pascal pointer type.

The new capability does not allow you to directly take the address of an arbitrary variable or directly dereference a generic pointer. However, you can take the address of any object in the Pascal heap, or you can use the C library function **malloc** to return a generic pointer. Once you have a generic pointer containing the desired address, you can use any Pascal pointer type as a "template" to dereference that pointer.

Here is an example of one approach that uses **malloc**:

```
(* Declare interface to C library function for dynamic
allocation *)

function malloc(number_of_bytes: integer): pointer; ex-
tern;

(* Declare interface to C library function for rapidly
setting a block of memory to a fixed value *)

procedure memset(destination: pointer; value: char;
number_of_bytes: integer); extern;
```

```
(* Two examples: a string, and an array of real numbers
*)
type
   big_char_array = packed array [0 ..  maxint] of char;
   string = record
      length: integer;
      data: ^big_char_array;
      end;
   big_real_array = packed array [0 ..  maxint] of real;
   matrix2d = record
      rows, columns: integer;
      data: ^big_real_array;
      end;

var
      s: string;
      m: matrix2d;
      i, j: integer;
      Begin

/* To read a string of length "i" from the input: */

   s.length = i;
   s.data = malloc(i * sizeof(char));
   if s.data = nil then
   ...handle allocation error here...
   for j := 0 to i - 1 do
   begin;
      s.data^[j] := input^;
      get(input);
      end;

   m.rows := 5;
   m.columns := 7;
   m.data := malloc(m.rows * m.columns *
                    sizeof(real));
   if m.data = nil then
   ...handle allocation error here...
   (* Clear the array *)
   memset(m.data, chr(0), m.rows * m.columns *
                             sizeof(real));
   for i := 0 to m.rows - 1 do
      for j := 0 to m.columns - 1 do
      m.data^[i * m.columns+ j] := 1.0;
```

For reasons explained in **Chapter 10** of this manual, you should refrain from using the generic–pointer facility with variables which lie in local or global memory rather than in the heap or the **malloc** area. For example, while the following trick does permit you to take the address of any character array, it is unsafe when used with ordinary local or global variables.

```
In one module:

function char_addr(p: pointer): pointer;
begin
     char_addr := p;
end;

In other modules:

function char_addr(var c: char): pointer; extern;
function mung_strings(p, q: pointer); extern;
var
   x: packed array [1 ..  10] of char;
   y: packed array [1 ..  100] of char;
   p, q: pointer;

Begin
   p := char_addr(x[1]);
   q := char_addr(y[1]);
   mung_strings(p, q);
   end
```

8
Fortran Programming Language

This chapter describes MIPS–FORTRAN, which contains full American National Standard (ANSI) Programming Language FORTRAN (X6.9–1978) plus MIPS extensions that provide full VMS FORTRAN compatibility to the extent possible without the VMS operating system or VAX data representation. MIPS–FORTRAN also contains extensions that provide partial compatibility with programs written in SVS FORTRAN and FORTRAN 66.

MIPS–FORTRAN is a superset of VMS FORTRAN; the MIPS compiler system can convert source programs written in VMS FORTRAN into machine programs executable under the UMIPS operating system.

See the *MIPS–FORTRAN Language Reference* and the *MIPS–FORTRAN User's Guide* for more information on language extensions.

If you wish to maintain your program on both an old system and on the RISComputer system, read **Section 6.1** in this manual.

8.1 Static Versus Automatic Allocation

For fastest program execution, the FORTRAN compiler uses **–automatic** allocation by default. If your program requires static allocation, you could use the **–static** driver option when you compile, however, program execution speed is sacrificed in most cases. A better solution is to use the ANSI FORTRAN 77 SAVE statement to specify the particular variables that must be statically allocated to make the program work correctly.

One symptom of a program that uses **–static** is repeated program failures because uninitialized local variables are used.

Neither ANSI FORTRAN 66 nor FORTRAN 77 permits a program to assume that local variables are automatically initialized to zero, or that local variables retain their values from the time a subroutine returns until the next time that subroutine is invoked.

Many older compilers use static allocation; that is, they allocate a location in global memory for each local variable in each subroutine. Because each local variable has its own fixed location, it starts out with a value of zero and retains its value even when the subroutine that declared it is not active. Applications on various systems often make use of this inadvertantly.

Automatic allocation uses a stack to implement local variables. It has several advantages.

First, because current local variables reside near the current stack pointer, the compiler can address them with short–offset load and store instructions, which execute more rapidly than large–offset instructions.

Second, local variables get popped from the stack when a subroutine returns, the total memory required for the program is less, and subroutines which are never active simultaneously can share memory for their local variables.

Third, automatic allocation permits the global optimizer to more effectively allocate local variables to registers within a subroutine. This is because the optimizer does not need to do either of the following:

- load the initial values of the variables from global memory at the start of the subroutine
- restore their final values to memory when the subroutine returns.

8.2 Retention of Data

MIPS–FORTRAN does not support the retention of data passed as parameters in previous calls to different entry points of a subroutine. This effect is not allowed by the FORTRAN standard and is error prone. However it is supported by some FORTRAN implementations and is required by some FORTRAN progrrams. Consider this example, your program calls an entry point to a subprogram with certain arguments, it then calls the subprogram again to a different entry point or the subprogram itself, the second call assumes that the arguments to the first call remain valid.MIPS FORTRAN does not support this usage.However, you can do one of the following to retain the data:

- set the arguments to local variables in the subprogram and use the **–static** switch to retain the values of the local variables.
- place the variable in a global common.

8.3 Variable Length Argument Lists

MIPS–FORTRAN does not support variable length argument lists, so your program can't call a routine the first time with fifteen arguments and a second time with two arguments.

8.4 Runtime Checking

Compile your program with runtime–checking using the –C option. The –C option generates code to check that subscripts do not exceed the range specified in the program. Performance is impacted once the program is debugged. Removing the –C option solves this problem. Storing one byte past the end of an array of characters may be harmless on one system if the compiler decided not to use the byte for anything. However, an execution error may occur on another system if the compiler tries to store something such as a subroutine address. This does not work if array parameters are declared as one element, which is common in older programs. To get around this, use the Fortran 77 "*" declaration.

8.5 Alignment of Data Types

RISComputer architecture imposes certain rules governing how data may be aligned in memory. Basically, a variable of size *n* bytes must be aligned on a boundary whose address is a multiple of *n* bytes, up to a maximum of 8 bytes. For example, because a half–word occupies two bytes, its address must be a multiple of two.

High–level languages also impose rules about where you can assume data appears in memory. In most cases, the language rules forbid the same things that the architecture forbids.

Occasionally, the rules conflict. For example, the ANSI X6.9–1978 standard for FORTRAN explicitly permits certain double–precision (8–byte) variables to lie on the same boundary as any real (4–byte) variable; but the RISComputer architecture requires the double–precision variable to be aligned on an 8–byte boundary.

The RISCompiler system supports the alignment rules imposed by each of its languages, even when they are more permissive than the architecture. FORTRAN, for example, deliberately avoids performing double–word load or store operations on certain double–precision variables.

Some extensions such as **integer*2**, which are not part of any language standard, cause problems. For example, consider the following common block:

```
common /x/i, j, k, l
integer*2 j, l, q(6)


integer*4 i, k
equivalence (q(1), i)
```

The compiler normally inserts a half–word of padding between *j* and *k* to conform to alignment rules, but that prevents *q(6)* from lying atop *l*.

Modifying your programs to align data according to the rules of the RISComputer architecture improves their performance. In the previous example, reversing the order of *j* and *k* within the common block eliminates the need for padding at the cost of changing the relationship between the array *q* and the scalar variables.

Rearranging the order of variables within a common block is not practical. However, you can use certain ''hidden'' options of the compiler system to generate code which tolerates misalignments but degrades performance. When uncertain if an object will be misaligned, the compiler generates slower code sequences.

You may choose one of the following three options to deal with various degrees of misalignment:

–align8 Permits objects larger than 8 bits to be aligned on 8–bit boundaries. This option requires the greatest amount of space; however, it is the most complete solution; 16–bit padding is not inserted for **integer*2** objects within common blocks.

–align16 Permits objects larger than 16 bits to be aligned on 16–bit boundaries; 16–bit objects must still be aligned on 16–bit boundaries (MC68000–like alignment rules); 16–bit padding is not inserted for **integer*2** objects within common blocks.

–align32 Permits objects larger than 32 bits to be aligned on 32–bit boundaries; 16–bit objects must still be aligned on 16–bit boundaries, and 32–bit objects must still be aligned on 32–bit boundaries. This option requires the least amount of space, but isn't a complete solution; 16–bit padding is inserted for **integer*2** objects within common blocks.

You should also use the following option no matter which above option you choose, unless experimentation proves this impossible:

–align_common Assumes that all common blocks are aligned properly, even though objects within the common blocks may be misaligned. This option generates better code. Without it, the assembler assumes that all global objects in languages like C and FORTRAN may be misaligned, even though they appear to be aligned, because they might be aliased against initialized objects in other modules to force the link editor to misalign them.

Pass these options specifically to the FORTRAN and assembly phases of the compiler system, by preceding them with **–wfb**, as shown:

```
f77 -Wfb,-align_common,-align16 ...
```

Two problems are not solved by these options:

Your program must not perform I/O directly on misaligned objects or perform any other operation which requires passing them by reference to runtime library routines, that have not been compiled with the **–align** flags.

You can circumvent this problem by copying misaligned objects to or from aligned temporaries before performing I/O. If the misaligned data is accessed only within libraries, and not by the kernel, you can circumvent the problem by using a runtime fix–up package which traps unaligned references and repairs them dynamically. See the **unaligned(3)** manual page for more information.

Keep in mind that trapping is expensive in terms of execution time.

8.6 Inconsistent Common–Block Sizes

ANSI FORTRAN requires that a named common block has the same size but not necessarily the same constituent variable each time it occurs in a program. However, programs often declare only the amount needed, thus making the length of the common block vary. For example:

```
subroutine foo
common /gdata/ theta
...
end
subroutine bar
common /gdata/ theta, omega, radius
...
end
```

The FORTRAN compiler allows uneven block sizes when possible by allocating the space required by the largest instance of the common block. If, however, the varying size causes one instance of a common block to fall below the **–G** threshold while another instance of the same common block is too big to fit into the *$gp* area, a problem results. At best the link editor prints error messages and the compiler system makes less than optimal use of the *$gp* area. At worst, a falsely small instance of the common block causes the compiler to overflow the *$gp* area. For a more detailed discussion of the **–G** option, see **Chapter 10** in this manual.

Use the link editor to report conflicting common block sizes by taking the name of each common block, converting it to lower case, prepending a **–y**, and appending an _. When you link your program, pass the above names to the link editor. The names must precede the first object file specified in the command line. For example, if the common blocks are named *gdata*, *rsb31*, and *xtrnls*, then type in:

```
f77 -ygdata_ -yrsb31_ -yxtrnls_ *.o -o myprog
```

The link editor reports the size that each common block has every time it occurs in an object file. The link editor also reports additional information about each common block; however, for common block problems, only size matters:

```
stdrfl.o: definition of common rsbg31_ size 1012
arstdr.o: definition of common rsbg31_ size 4
```

8.7 Multiple Initializations of Common blockdata

ANSI FORTRAN 77 requires that you use a DATA statement on a named common block only within a blockdata subprogram. An ordinary subroutine may initialize only local variables, not common variables; MIPS compiler system does not enforce this restriction.

However, the MIPS compiler does enforce the ANSI FORTRAN 77 restriction which requires that you initialize each common block within exactly one subprogram. A variety of messages appear when you violate this restriction, including error messages from the link editor citing multiply defined symbols or messages from earlier phases of the compiler citing *illegal init* or *illegal space*. For example:

```
ugen: internal  : line 6345 : ../symbol.p, line
270
illegal inits
```

To diagnose such problems, use the utility **fsplit** to split your program into many small files, with one subprogram per file. Then link the program and collect the multiply defined messages in a file. For each multiply–defined symbol, prepend a *–y*, and relink the program with these options preceding the list of object files in the command line. For example, if the link editor issues error messages for *gdata_* and *rsb31_*, then relink with:

```
f77 -ygdata_ -yrsb31_ *.o -o myprog
```

The link editor uses the phrase *definition of external data* every time an object file initializes a symbol:

```
bstimr.o: definition of external data gdata_
zuxeng.o: definition of external data rsb31_
stdrfl.o: definition of common rsbg31_ size 1012
cmflo1.o: definition of external data rsb31_
cmflow.o: definition of external data gdata_
arstdr.o: definition of common gdata_ size 4032_
```

The phrase *definition of common* can appear repeatedly for a particular common block, but the phrase *definition of external data* must appear only once for each common block.

Once you realize that two *.o* files are initializing the same common block, transfer the appropriate DATA statements from one to the other (or, preferably, to a blockdata subprogram), then recompile and relink.

8.8 Endianness and integer*2

Special problems exist for porting FORTRAN programs between big– and little–endian machines in addition to those discussed in **Section 4.4**. Although FORTRAN programs pass arguments by reference (they pass the address of the argument rather than the argument itself), they cannot declare the formal arguments of a subroutine. Consider the following call:

```
call foo(0.314159e1, 0.628318d1, 1234, 2468)
```

Clearly the first argument is type real or **real*4**, and the second argument is type double precision or **real*8**. But the types the third and fourth argument, which can be either **integer*2** or **integer*4**, are unknown to the compiler. Thus, the compiler allocates four bytes for each of these variables.

On a little–endian machine, where the address of an integer is the address of its low–order byte, this code works correctly even if subroutine foo expects the arguments to be **integer*2**, because the address is the same in either case. On a big–endian machine, where the address of an integer is the address of its high–order byte, this code fails: if a four–byte integer is passed to a subroutine which expects a two–byte integer, then the subroutine recognizes only the two upper bytes of the four–byte integer.

There are two solutions:

- If all of the formal arguments in your program are two–byte integers, and you also wish the compiler to use two–byte integers wherever you have declared variables as **integer** rather than **integer*4**, then you can use the **–i2** option when you compile your program, and all literal integers will use only two bytes.
- If it is not possible to use **–i2**, then you must use temporary variables of type **integer*2** to pass literal numbers to two–byte arguments:

```
integer*2 temp1, temp2

temp1 = 1234
temp2 = 2468
call foo(0.314159e1, 0.628318d1, temp1, temp2)
```

9
RISCwindows

This chapter describes the issues that you need to be aware of when porting an X program to RISCwindows. RISCwindows is MIPS' implementation of version 11 of the X Window System. RISCwindows 3.0 contains one toolkit : the X toolkit intrinsics and the Athena Widgets from the X Consortium. Other companies may have their own widget set and intrinsics; either may be incompatible with MIPS'. Refer to the *RISCwindows Reference Manual* and the *Release Notes* for more information.

9.1 Environment

You can program RISCwindows using either the SystemV or BSD version of RISC/os by using the command line arguments **–systype sysv** (the default) or **–systype bsd43**. See **Chapter 3** in this manual for more information.

9.2 System V Issues

Headers and Defines

If your program uses #ifdef SYSV or the header file X11/Xos.h, then the command line flags **–DSYSV**, **–DMIPS**, and **–lbsd** are required when you compile. Always include these flags, since the library X11/Xos.h, may be included by another header file.

Sigset

X client programs should use the "sigset" family of signal management system calls rather than "signal", because Xlib uses "sigset" and the two cannot be mixed.

Linking

To link a program using Xlib, **–lbsd** is required. For programs using the "Load" toolkit widget, **–lmld** is also required.

9.3 BSD Issues

If you are compiling a RISCwindow program under BSD, then you need only include the command line flag **–DMIPS** when you compile.

9.4 Hardware Issues

Refer to the technical reference manual that accompanies your MIPS workstation for specific information on the number of pixels per inch, color table, number of bit planes, and so on.

10 PL/I

This chapter describes the issues that you need to be aware of when porting a PL/I program to a MIPS computer. You should be aware of the MIPS–PL/I implementation for the RISCompiler System. This implementation is described in *Part I: Programmer's Guide* of the *MIPS–PL/I Programmer's Guide and Language Reference Manual.*

10.1 PL/I Extensions

MIPS PL/I conforms to a subset of the full ANSI PL/I called subset G. Some extensions have also been added which are discussed in **Appendix G** of the *MIPS–PL/I Language Programmer's Guide.* You should be familiar with the differences between the full PL/I language and the G subset before you port your program.

10.2 Alignment of Data in Memory

The way data is aligned in memory from system to system is such that you will probably have to write a program to convert the data to a usable format.

If your program specifies aligned data, but sends unaligned data, it causes problems. To solve the problem, the option;

```
-wk, -force, -unalign
```

causes the compiler to treat all formal and actual arguments of type 'bit' as if they are unaligned. This degrades the execution speed of the compiled program, but relaxes somewhat the requirement that formal and actual bit parameter types match exactly.

10.3 The ADDR() Function

The ADDR() function returns a pointer to the storage referenced by a specified variable *x*. The variable x must be a reference to a parameter whose corresponding argument is an array that is a member of a dimensioned structure because the storage of such an array is fragmented and cannot be accessed by a pointer and a based variable.

On many implementations, x must not be an unaligned bit–string or a structure consisting entirely of unaligned bit–strings.

11 RISCompiler Components

11.1 Introduction

This chapter discusses considerations for debugging, programming checking, compiling, and link editing your programs; the chapter discusses the following topics:

Debugging Procedures. You compile programs for debugging using the **–g** option of the driver command that compiles your program and then executing the resulting object with the dbx debugger.

Programming Checking. Several program checking tools are available to check the correctness of your program.

Optimization. The optimizer can significantly improve the performance of your object program. The optimizer is invoked using one of the several **–O** options of the driver command. You should consider levels of optimization higher than the standard default once your program is successfully debugged.

Link Editor Features. Several link editor options and techniques should be considered. These options are invoked by either a driver command (**cc, pc, f77, pl1, cob**) or the link editor **ld** command.

In addition to the information provided in this chapter, you may need to refer to the *Languages Programmer's Guide* and the manual page for the driver, **dbx**, or **ld** in the *RISC/os User's Reference Manual.*

11.2 Debugging

This section gives a suggested procedure to follow when debugging your ported program. For a complete description of the debugger **dbx**, refer to the *Languages Programmer's Guide.*

If a program fails and you wish to use **dbx** to debug the failed program, do the following:

1. Recompile the program using the following compiler options:
   - the **–g** debugging option, which causes the compiler system to generate the symbol table required by **dbx**.

- the **–O1** optimizing options (the default), which causes the compiler system to minimally optimize the resulting object. (Once program is successfully debugged, you may want to recompile it using a higher level of optimization.)
- the **–signed** and **–varargs** options (for C programs only).
- the **–static** option (for FORTRAN programs only.)

2. Execute the program.

3. If a segmentation fault, bus error, or other error causes the program to default, then use **dbx** to isolate the problem. Do a stack trace using the dbx **where** command to locate the point of failure.

4. If you know the approximate location of where the problem occurs, then do the following:

   - Use the dbx **stop** command to set a breakpoint just before the suspected problem location.
   - Use the dbx **where** command to display the current values contained in the pertinent variables
   - Use the dbx **next** or **step** command to incrementally execute the instructions after the breakpoint. Display and check the values of the variables as you execute each instruction.

5. Use binary search techniques, as discussed in step 4, when you are trying to track down the source of corrupted data. You can also make a change to data or code to see what happens; understand the code before you do this. For example, sometimes all you need to do is to check for the symptom that results in a problem, and bypass the code that would be executed. A classic example of this is programs that get segmentation faults for doing the following:

```
if(*sp=='a'){
                    ...
}
```

   If sp is 0, then a segmentation fault occurs, but the code works as expected if it is changed to:

```
if(sp && *sp == 'a'){
                    ...
}
```

11.3 Program Checking

A correct program is not necessarily a portable program as it may run successfully on one system, but not another. Debugging alone does not guarantee correctness. In fact, no tool can completely guarantee the correctness of a program; however, a few tools can help check whether a program is operating correctly. These tools are appropriate to use either when porting a program from another system to a RISComputer, or when writing a program on a RISComputer intended to be portable to other systems.

11.3.1 Lint

One such tool is Lint, a static program checker for the C programming language. Lint provides the sort of checking that is typically performed by compilers in other programming languages. Its use for C programs is highly recommended. See the **Language Programmer's Reference** for more information.

11.3.2 Subscript Range Checks

Another tool is subscript range checking. It is not uncommon for a program to reference an array outside of the declared bounds. An error of this sort may go undetected if, for example, the location referenced exists, but is otherwise unused. When the program is ported to another system the incorrect reference may instead access a critical location, and the program will fail to operate correctly.

To detect subscript range errors, your program may be compiled with a special option that generates extra code to verify that the indexes to array references are within the declared bounds of the array. This option is available in Pascal and FORTRAN. It is the default in Ada. For C, the language and its style of use, does not make subscript range checking useful, so no compiler option is provided.

A Pascal program compiled without subscript range checking would run:

```
% pc -q -o example example.p
```

However, if you compiled the same program with subscript checks, you would receive a subscript error during run time.

```
% pc -c -q -o example
% ./example
  Trace/BPT trap (core dumped)
```

At this point, you could use **dbx(1)** to locate the source line with the subscript range error.

The –C compile option also works for a FORTRAN program. Older FORTRAN programs require some modification to work with subscript range checking turned on. It was once common in FORTRAN to declare array parameters to have dimension 1 when the actual size was passed as a separate parameter:

```
subroutine zero(a, n)
real a(1)
do 10 i = 1, n
10    a(i) = 0
end
```

In FORTRAN 77 the declaration could correctly be written as:

```
real a(n)
```

or

```
real a(*)
```

if the array size is not passed as a parameter.

11.3.3 Dynamic Storage Allocation

Just as programs sometimes reference outside the bounds of an array, a common error is to call a dynamic storage allocator and reference outside of the allocated block. Since the compiler often does not know the size of the block when a pointer based reference is made, it cannot generate code to verify the access, as with subscript range checking. However, a special version of the standard dynamic storage allocation routines malloc(), free(), and realloc() called **malloccheck(1)** is available that checks for incorrect uses of dynamic storage. Add **–lmalloccheck** to your link command line to use this version. It checks for the following:

Writing beyond an allocated block. A common error is to write beyond the end of an allocated block. The malloccheck allocator allocates extra space both before and after the block it returns to you and initializes this space to special bit patterns. A write outside the block will usually affect these pattern words. When the block is freed, the pattern words are checked, and if modified a warning is given.

Freeing a block twice. Another error is to free a block twice. Malloccheck does not re–use storage after it is freed, but instead simply marks it as such. A second free to the same block generates a warning.

Referencing a block after it is freed. Another error is to reference a block after it is freed. This often works because the freed storage is not immediately re–used. Malloccheck's free routine overwrites the data when it is free, which usually causes subsequent references to return unexpected results, leading to a detectable program failure later.

Initializing allocated storage to zero. Some programs inadvertently assume the allocated storage is initialized to zero, even though the standard malloc() and free() routines do not guarantee this. Malloccheck initializes the allocated storage to non–zero so that such assumptions lead to program failure.

Malloccheck's primary checking is done when blocks are freed. An error may go undetected if a block is never freed, or if the error occurs after it is freed. Also, an inconsistency detected by free may be difficult to trace to an error made long before. For all of these reasons, malloccheck provides the malloc_status() subroutine, which checks the entire dynamic storage allocation area. Calls to malloc_status() can be inserted in the program as necessary to locate the source of an error. During program development a single call to malloc_status() at the end of the program is useful. The argument to malloc_status() specifies the level of checking:

```
malloc_status(0);
```

checks for errors and prints some summary statistics. A level of 1:

```
malloc_status(1);
```

checks for errors and prints some summary statistics and lists all blocks that remain in use. This is useful for finding blocks that the program failed to free. Failure to free storage can lead to eventual memory exhaustion and program failure on a large run.

For example if you compile and link example.c with the default allocator, example runs:

```
% cc -g -o example example.c
```

However if you link example with malloccheck, it it finds the following errors.

```
% cc -g -o example example.c -lmalloccheck
% ./example

 Error: check word at 10003834 preceding block at
 10003838 bashed from 87ccccc1 to 87cccc01.
 Error: pad byte at 100038b8 of block at 100038b0
 bashed from 5a to 02.
 Error: freeing block at 100038d0 again.
 Error: check word at 100038ec following block at
 100038f8 bashed from 87ccccc2 to 03ccccc2.
 Error: check word at 10003908 following block at
 10003910 bashed from 87ccccc3 to 04ccccc3.
 Error: trailer size word at 10003934 for block at
 10003928 bashed to 05000004.
 Error: realloc(NULL, 20).
 Error: realloc of free block at 10003968.
 Warning: malloc(268435456). Will return NULL.
 Warning: sbrk(8388640) failed. Will return NULL.
 malloc_status(1):
 Error: check word at 10003834 preceding block at
 10003838bashed from 87ccccc1 to 87cccc01.
```

11.4 Optimization

The MIPS optimizer is vulnerable to human error, for example, incorrectly specifying the size of a variable or the nature of a formal argument. In the following Pascal code, the optimizer may move the *if* statement to precede the loop, since *name_changer* is declared to receive only one character, therefore *name[5]* cannot change during the loop:

```
type
   array5 = packed array [1 ..  5] of char;
var
   i: integer;
   name: array5;
procedure name_changer(var c: char); extern;

for i := 1 to 10 do
   begin
   if name[5] >'9' then goto 5;
   name_changer(name[1]);
   writeln(name);
   end;
```

This assumption is true if *name_changer* is coded in Pascal and the formal argument agrees with the actual argument. If it is coded in C, and the formal argument is *char *c*, then *name_changer* may alter *name[5]* during the loop. To solve this problem be specific. Don't specify *var c: char* if it is actually *var c: array5* from the point of view of the external procedure.

Similar problems arise in FORTRAN programs that assume declaring a formal argument or common block to be an array of one element is the same as declaring it specifically:

```
common /x/ ary(1)

call matset(ary)
```

If a common declaration in another program unit specify *ary(100)*, then the variable *ary* becomes 100 elements large when you link the program; but in this particular section, the optimizer behaves as if the variable had only one element. This problem can be solved as follows:

- Use consistent common declarations.
- Use an ANSI FORTRAN 77 declaration in the form of *integer parm(*)* rather than the traditional trick of *integer parm(1)* when the size of a formal parameter may vary.

11.5 The Link Editor

This section describes the special features of the link editor that you should be aware of when porting a program. For information on the link editor and its libraries, refer to the **ld(1)** manual page in the *RISC/os (UMIPS) User's Reference Manual.*

11.5.1 The –G option

The RISCompiler system sets up one register called ***gp*** to point to a 64Kbyte block of global memory that can be addressed in half the number of instructions required for a normal global access. It allocates by default to the gp area any global variable up to a maximum size of eight bytes. You can change the default size using the driver **–G** option (see the *Language Programmer's Guide* or the associated compiler manual page in the *User's Reference Manual*).

There are three kinds of *gp*–related problems:

1. The *gp* area overflows because *gp*–relative data doesn't fit into its allocated 64Kbytes of memory.

   If this problem occurs, the link editor prints a prediction of the best value to use as a maximum size in the **–G** option. The "best value" places as many global variables as possible into the 64Kbyte area to improve performance, but excludes enough variables to prevent the area from overflowing.

   However, the "best value" is merely a prediction and may not produce successful results. To make sure that no *gp* area overflow occurs, and to produce an executable object immediately, note the best value provided by the link editor, and then recompile and relink your program using the **–G 0** option. You can then move that copy to a safe place and recompile and relink using the recommended best value.

   If your program does not fail, but you want to improve performance, then use the **–bestGnum** option. This option causes the link editor to predict a best value. Recompile and relink with the new value. However, you should first debug the program at the default setting, save a working copy, and then experiment with the best number prediction.

2. A variable larger than the maximum specified size is in the *gp* area.

   This problem can happen when two program modules disagree about the data type of an object. For example, one program sees the data as a small variable and addresses it within the *gp* area, and the other sees it as a large variable.

The link editor retains the larger size of the variable, when possible, and places it into the *gp* area with a warning error message. This may cause the *gp* area to overflow. If the *gp* area overflows, then use the **—G 0** option or (preferably) reconcile the conflicting declarations so as to retain the advantages of using the *gp* area for other variables.

Sometimes the link editor cannot put the large variable into the *gp* area because it is a synonym for some other object that cannot be addressed relative to the *gp* register. If this is the case, you must reconcile the conflicting declarations. For example, suppose one module defines an object as a function, which cannot be addressed relative to the gp register:

```
int foo();

bar(foo);
```

and another defines it as a small data item:

```
int foo, *ptr;

ptr = &foo;
```

Most inconsistently sized declarations are caused by a violation of the ANSI standard with regard to FORTRAN common blocks. See **Chapter 8** in this manual for details.

3. The link editor believes that the *gp* register isn't initialized.

   This problem can occur when you use your own start up code, rather than the runtime startup code in *crt0.o* or *crt1.o* provided when a RISCompiler driver (**cc**, **f77**, **pc**, **cobol**, or **pl1**) links your program.

   The runtime startup code loads a link editor–defined symbol called *_gp* into the *gp* register. If you use your own startup code instead, load *_gp* into some register ($0 is acceptable) even if you load *gp* with some other value that you have calculated yourself; otherwise, the link editor issues an error message.

Two details may help you in reconciling inconsistently sized declarations:

1. If a common variable is declared but not referenced in a module, then the compiler allocates it outside the *$gp* area regardless of its size. This allocation reduces possible problems. Therefore, you should explicitly initialize unreferenced variables to zero, to ensure that they are placed within the *$gp* area.
2. In C, you can force a scalar variable to be referenced as if it lay outside the $gp area by declaring it to be an array of unspecified size and referencing the first element (for example, *int[] j;* and *j[0]* rather than *int j* and *j*).

11.5.2 Forcing Library Extractions

The RISCompiler system link editor opens and searches only one library at a time in the order you specify. This can cause problems as the following example shows. Suppose you try to link a program *p.o* with two libraries, *l1.a* and *l2.a*, as follows:

```
cc -o p p.o l1.a l2.a
```

The components that the program and libraries contain or need are:

File/Library	Contains	Imports/Exports
p.o		imports l2proc
l1.a	l1.o	export l1proc, import l3proc
l2.a	l2.o	export l2proc, import l1proc
l2.a	16.o	exports l3proc

When the program is compiled:

1. The link editor sees that it needs to import *l2proc* for *p.o*
2. It searches *l1.a* for *l2proc*, and does not find it
3. The link editor closes *l1a* and opens *l2a*
4. It finds the *l2proc* but cannot find the *l1proc* because *l1.a* is closed

If you specify *l1.a* and *l2.a* in the opposite order, then the link editor fails to obtain *l3proc*.

The standard UNIX solution to this problem in which you assemble a file *kludge.s* containing:

```
.globl l1proc
```

and link *kludge.o* prior to *l1.a* to import *l1proc* does not work on the RISCompiler system. The RISCompiler assembler notices that *kludge.s* does not really use *l1proc*, and as an optimization removes the request to import it. To solve this problem, edit *kludge.s* so that it defines *l1proc*:

```
.extern l1proc
   .data
   .word l1proc
```

Simpler solutions are to:

1. Correct the problem on the command line by having the link editor search the *l1.a* library twice:

```
cc -o p p.o l1.a l2.a l1.a
```

2. Extract the object file and directly include it in the command line.

```
ar x l1.a l1.o
cc -o p p.o l1.o l1.a l2.a
```

11.5.3 The Semantics of a Library Search

Some programs assume that the link editor searches linearly within a library for symbols that it wishes to import. The RISCompiler link editor libraries use a hashed symbol table for faster linking, so the order in which *.o* files are added to a *.a* file is insignificant.

The link editor does not consider a "common" declaration to be a request to import every module that issues an identical "common" declaration. For example, a declaration of *int errno* in a C–coded main program does not cause the link editor to import every module that similarly declares *int errno*; those modules are imported only if they specifically export some symbol that your program specifically imports using a function definition or initialized data definition

However, a ''common'' in the library can satisfy an import request without actually adding the library module to the program. For example, if your main program declares *extern int errno*, the occurrence of *int errno* in a module *foo.o* in the library would create a common ''int errno'' in the linked program, without necessarily adding *foo.o* to the linked program. This rather exotic behavior makes our link editor compatible with the one provided by the standard BSD UNIX distribution.

11.5.4 Libraries Versus Object Files

If you want to bundle together a group of infrequently–changed object files because it is more convenient to specify a single name when you link, it is faster to use *ld –r* to bundle them into a *.o* file than to use *ar –r* to add them to a *.a* file.

A

B

C

D

E

F

I

L

M