MIPSpro™ N32 ABI Handbook

Document Number 007-2816-002

CONTRIBUTORS

Written by George Pirocanac
Edited by Wendy Ferguson, Larry Huffman, and Bean Anderson
Production by Lorrie Williams
Engineering contributions by Steve Cobb, Jim Dehnert, Jay Gischer, W. Wilson Ho, Lilian Leung, and Ash Munshi

MIPSpro™ N32 ABI Handbook
Document Number 007-2816-002

Contents

List of Figures

Tables

Chapter 1

N32 ABI Overview

Welcome to the N32 ABI Handbook. This book describes the N32 High Performance 32-bit Application Binary Interface (ABI) for the MIPS architecture.

Contents of This Guide

As you continue reading this guide, you'll learn about N32. Topics include:

- Chapter 1 (this chapter), "N32 ABI Overview"
  - "What is N32?," which describes the n32 ABI and compares it with the other MIPS ABIs.
  - "Why We Need a New ABI," which lists the reasons why we need a new ABI.
  - "N32 Migration Requirements," which describes what is required of both SGI and its customers to use the n32 ABI.
- Chapter 2, "Calling Convention Implementations"
- Chapter 3, "N32 Compatibility, Porting, and Assembly Language Programming Issues"
- Chapter 4, "N32 Examples and Case Studies"

This document uses the following terminology:

o32	The current 32-bit ABI generated by the ucode compiler, that is, 32-bit compilers prior to IRIX 6.1 operating system.
n32	The new 32-bit ABI generated by the MIPSpro 64-bit compiler.
n64	The new 64-bit ABI generated by the MIPSpro 64-bit compiler.

What is N32?

N32 is a minor variation on the high performance 64-bit ABI. All the performance of the hardware is available to the program and existing 32-bit ABI programs are easily ported. Table 1-1 compares the various ABIs.

Table 1-1 ABI Comparison Summary

	o32	**n32**	**n64**
Compiler Used	ucode	MIPSpro	MIPSpro
Integer Model	ILP32	ILP32	LP64
Calling Convention	mips	new	new
Number of FP Registers	16 (FR=0)	32 (FR=1)	32 (FR=1)
Number of Argument Registers	4	8	8
Debug Format	mdbug	dwarf	dwarf
ISAs Supported	mips1/2	mips3/4	mips3/4
32/64 Mode	32 (UX=0)	64 (UX=1) *	64 (UX=1)

* UX=1 implies 64-bit registers and also indicates that MIPS3 and MIPS4 instructions are legal. N32 uses 64-bit registers but restricts addresses to 32 bits.

Why We Need a New ABI

The Application Binary Interface, or ABI, is the set of rules that all binaries must follow in order to run on an SGI system. This includes, for example, object file format, instruction set, data layout, subroutine calling convention, and system call numbers. The ABI is one part of the mechanism that maintains binary compatibility across all SGI platforms.

Until IRIX operating system version 6.1, Silicon Graphics supported two ABIs: a 32-bit ABI and a 64-bit ABI. IRIX 6.1 supports a new ABI, n32.

The following sections outline limitations of the old 32-bit ABI the 64-bit ABI. These issues form the motivation for the new n32 ABI. Specifically, topics covered include:

- "Limitations of the 32-bit ABI"
- "Limitations of the 64-bit ABI"
- "Motivation for the N32 ABI"

Limitations of the 32-bit ABI

The 32-bit ABI was designed essentially for the R3000. We can't extend the 32-bit ABI to use new performance-related features and instructions of the R4400 and beyond. For example:

- We can't use 16 of the 32 floating point registers.
- We can't use any 64-bit arithmetic instructions.
- We can't use any 64-bit data movement instructions.
- We can't use any MIPS4/R8000 instructions.

Because of this, we lose the performance available from the chip. Floating point intensive programs are especially hurt by these limitations; indeed some are 50%-100% slower!

Limitations of the 64-bit ABI

Although the 64-bit ABI exploits many performance-related features of the MIPS architecture, it also has issues that make using it problematical. These issues include the following:

- Many ISVs have been unable to port their code from the 32-bit ABI to the 64-bit ABI. This is not a simple port; in fact, it usually requires redesign and recoding.
- When ported from the 32-bit ABI to the 64-bit ABI, some C programs get significantly larger.

Motivation for the N32 ABI

Many ISVs and customers are finding it difficult to port to the 64-bit ABI. Clearly we need an ABI with all of the performance advantages of the 64-bit ABI, but with the same data type sizes as the 32-bit ABI to allow ease of porting.

N32 Migration Requirements

In order to implement n32, Silicon Graphics provides the following to our customers:

- A compiler that supports n32. (The 6.1 version of MIPSpro compilers supports n32).
- A kernel that supports n32. (IRIX 6.1 supports n32).
- N32 versions of each library.

To take advantage of the n32 ABI, our customers must:

- Install n32 OS, compiler, and all n32 libraries.
- Rewrite Assembly code to conform to n32 guidelines.
- Prototype C functions that use *varargs.*
- Recompile all the code with the compiler that supports n32 (use **–n32** on the command line). *Makefile* changes are needed only if explicit library paths are used.

Chapter 2

Calling Convention Implementations

This chapter describes the differences between o32, n32, and n64 ABIs with respect to calling convention implementations. Specifically, this chapter describes:

- "Native 64-Bit (N64) and N32 Subprogram Interface for MIPS Architectures" covers the 64-bit subprogram interface. This interface is also used in the n32 ABI.
- "Implementation Differences" identifies differences in the 32-bit and 64-bit implementations C programming language and explains why it's easier to port to n32 rather than to 64 bits.
- "ABI Attribute Summary" lists the important attributes for the o32 and n32/64-bit ABI implementations.

Native 64-Bit (N64) and N32 Subprogram Interface for MIPS Architectures

This section describes the internal subprogram interface for native 64-bit (n64) and n32 programs. This section assumes some familiarity with the current 32-bit interface conventions as specified in the MIPS application binary interface (ABI). The transition to native 64-bit code on the MIPS R8000 requires subprogram interface changes due to the changes in register and address size.

The principal interface for 64-bit native code is similar to the 32-bit ABI standard, with all 32-bit objects replaced by 64-bit objects. Note that square brackets [ ] indicate different n32-bit and o32-bit ABI conventions.

In particular, this implies:

- All integer parameters are promoted (that is, sign- or zero-extended to 64-bit integers and passed in a single register). Typically, no code is required for the promotion.
- All pointers and addresses are 64-bit objects. [Under n32, pointers and addresses are 32 bits.]

- Floating point parameters are passed as single- or double-precision according to the ANSI C rules. [This is the same under n32.]
- All stack parameter slots become 64-bit doublewords, even for parameters that are smaller (for example, floats and 32-bit integers). [This is also true for n32.]

In more detail, the 64-bit native calling sequence has the following characteristics.

- All stack regions are quadword aligned. [The 32-bit ABI specifies only doubleword alignment.]
- Up to eight integer registers (*$4 .. $11*) may be used to pass integer arguments. [The 32-bit ABI uses only the four registers *$4 .. $7*.]
- Up to eight floating point registers (*$f12 .. $f19*) may be used to pass floating point arguments. [The 32-bit ABI uses only the four registers *$f12 .. $f15*, with the odd registers used only for halves of double-precision arguments.]
- The argument registers may be viewed as an image of the initial eight doublewords of a structure containing all of the arguments, where each of the argument fields is a multiple of 64 bits in size with doubleword alignment. The integer and floating point registers are distinct images, that is, the first doubleword is passed in either *$4* or *$f1*, depending on its type; the second in either *$5* or *$f1*; and so on. [The 32-bit ABI associates each floating point argument with an even/odd pair of integer or floating point argument registers.]
- Within each of the 64-bit save area slots, smaller scalar parameters are right-justified, that is, they are placed at the highest possible address (for big-endian targets). This is relevant to float parameters and to integer parameters of 32 or fewer bits. Of these, only **int** parameters arise in C except for prototyped cases – **floats** are promoted to **doubles**, and smaller integers are promoted to **int**. [This is true for the 32-bit ABI, but is relevant only to prototyped small integers since all the other types were at least register-sized.]
- 32-bit integer (*int*) parameters are always sign-extended when passed in registers, whether of signed or unsigned type. [This issue does not arise in the 32-bit ABI.]
- Quad-precision floating point parameters (C **long double** or Fortran **REAL*16**) are always 16-byte aligned. This requires that they be passed in even-odd floating point register pairs, even if doing so requires skipping a register parameter and/or a 64-bit save area slot. (The 32-bit ABI does not consider long double parameters, since they were not supported.)
- **Structs**, **unions**, or other composite types are treated as a sequence of doublewords, and are passed in integer or floating point registers as though they were simple

scalar parameters to the extent that they fit, with any excess on the stack packed according to the normal memory layout of the object. More specifically:

- Regardless of the **struct** field structure, it is treated as a sequence of 64-bit chunks. If a chunk consists solely of a double float field (but not a **double**, which is part of a **union**), it is passed in a floating point register. Any other chunk is passed in an integer register.
- A **union**, either as the parameter itself or as a **struct** parameter field, is treated as a sequence of integer doublewords for purposes of assignment to integer parameter registers. No attempt is made to identify floating point components for passing in floating point registers.
- Array fields of **structs** are passed like **unions**. Array parameters are passed by reference (unless the relevant language standard requires otherwise).
- Right-justifying small scalar parameters in their save area slots notwithstanding, **struct** parameters are always left-justified. This applies both to the case of a **struct** smaller than 64 bits, and to the final chunk of a **struct** which is not an integral multiple of 64 bits in size. The implication of this rule is that the address of the first chunk's save area slot is the address of the **struct**, and the **struct** is laid out in the save area memory exactly as if it were allocated normally (once any part in registers has been stored to the save area). [These rules are analogous to the 32-bit ABI treatment – only the chunk size and the ability to pass double fields in floating point registers are different.]

- Whenever possible, floating point arguments are passed in floating point registers regardless of whether they are preceded by integer parameters. [The 32-bit ABI allows only leading floating point (FP) arguments to be passed in FP registers; those coming after integer registers must be moved to integer registers.]
- Variable argument routines require an exception to the previous rule. Any floating point parameters in the variable part of the argument list (leading or otherwise) are passed in integer registers. Several important cases are involved:
  - If a **varargs** prototype (or the actual definition of the callee) is available to the caller, it places floating point parameters directly in the integer register required, and no problems occur.
  - If no prototype is available to the caller for a direct call, the caller's parameter profile is provided in the object file (as are all global subprogram formal parameter profiles), and the linker (*ld/rld*) generates an error message if the linked entry point turns out to be a **varargs** routine.

Note: If you add **–TENV:varargs_prototypes=off** to the compilation command line, the floating point parameters appear in both floating point registers and integer registers. This decreases the performance of not only **varargs** routines with floating point parameters, but also of any unprototyped routines that pass floating point parameters. The program compiles and executes correctly; however, a warning message about unprototyped **varargs** routines still is present.

  - If no prototype is available to the caller for an indirect call (that is, via a function pointer), the caller assumes that the callee is not a **varargs** routine and places floating point parameters in floating point registers (if the callee is **varargs**, it is not ANSI-conformant).

- The portion of the argument structure beyond the initial eight doublewords is passed in memory on the stack and pointed to by the stack pointer at the time of call. The caller does not reserve space for the register arguments; the callee is responsible for reserving it if required (either adjacent to any caller-saved stack arguments if required, or elsewhere as appropriate.) No requirement is placed on the callee either to allocate space and save the register parameters, or to save them in any particular place. [The 32-bit ABI requires the caller to reserve space for the register arguments as well.]
- Function results are returned in **$2** (and **$3** if needed), or **$f0** (and **$f2** if needed), as appropriate for the type. Composite results (**struct**, **union**, or **array**) are returned in **$2**/**$f0** and **$3**/**$f2** according to the following rules:
  - A **struct** with only one or two floating point fields is returned in **$f0** (and **$f2** if necessary). This is a generalization of the Fortran COMPLEX case.
  - Any other *struct* or *union* results of at most 128 bits are returned in **$2** (first 64 bits) and **$3** (remainder, if necessary).
  - Larger composite results are handled by converting the function to a procedure with an implicit first parameter, which is a pointer to an area reserved by the caller to receive the result. [The 32-bit ABI requires that all composite results be handled by conversion to implicit first parameters. The MIPS/SGI Fortran implementation has always made a specific exception to return COMPLEX results in the floating point registers.]
- There are eight callee-saved floating point registers, **$f24..$f31** for the 64-bit interface. There are six for the n32 ABI, the six even registers in **$f20..$f30**. [The 32-bit ABI specifies the six even registers, or even/odd pairs, **$f20..$f31**.]
- Routines are not be restricted to a single exit block. [The 32-bit ABI makes this restriction, though it is not observed under all optimization options.]

There is no restriction on which register must be used to hold the return address in exit blocks. The **.mdebug** format was unable to cope with return addresses in different places, but the DWARF format can. [The 32-bit ABI specifies **$3**, but the implementation supports **.mask** as an alternative.]

PIC (position-independent code, for DSO support) is generated from the compiler directly, rather than converting it later with a separate tool. This allows better compiler control for instruction scheduling and other optimizations, and provides greater robustness.

In the 64-bit interface, **gp** becomes a callee-saved register. [The 32-bit ABI makes **gp** a caller-saved register.]

Table 2-1 specifies the use of registers in native 64-bit mode. Note that “Caller-saved” means only that the caller may not assume that the value in the register is preserved across the call.

Table 2-1 Native 64-Bit and N32 Interface Register Conventions

Register Name	Software Name	Use	Saver
$0	zero	Hardware zero	
$1 or $at	at	Assembler temporary	Caller-saved
$2..$3	v0..v1	Function results	Caller-saved
$4..$11	a0..a7	Subprogram arguments	Caller-saved
$12..$15	t4..t7	Temporaries	Caller-saved
$16..$23	s0..s7	Saved	Callee-saved
$24	t8	Temporary	Caller-saved
$25	t9	Temporary	Caller-saved
$26..$27	kt0..kt1	Reserved for kernel	
$28 or $gp	gp	Global pointer	Callee-saved
$29 or $sp	sp	Stack pointer	Callee-saved

Table 2-1 (continued) Native 64-Bit and N32 Interface Register Conventions

Register Name	Software Name	Use	Saver
$30	s8	Frame pointer (if needed)	Callee-saved
$31	ra	Return address	Caller-saved
hi, lo		Multiply/divide special registers	Caller-saved
$f0, $f2		Floating point function results	Caller-saved
$f1, $f3		Floating point temporaries	Caller-saved
$f4..$f11		Floating point temporaries	Caller-saved
$f12..$f19		Floating point arguments	Caller-saved
$f20..$f23 (32-bit)		Floating point temporaries	Caller-saved
$f24..$f31 (64-bit)		Floating point	Callee-saved
$f20..$f31 even (n32)		Floating point temporaries	Callee-saved
$f20..$f31 odd (n32)		Floating point	Caller-saved

Table 2-2 shows several examples of parameter passing. It illustrates that at most eight values can be passed through registers. In the table note that:

- **d1..d5** are double precision floating point arguments
- **s1..s4** are single precision floating point arguments

- **n1..n3** are integer arguments

Table 2-2 Native 64-Bit and N32 C Parameter Passing

Argument List	Register and Stack Assignments
d1,d2	$f12, $f13
s1,s2	$f12, $f13
s1,d1	$f12, $f13
d1,s1	$f12, $f13
n1,d1	$4,$f13
d1,n1,d1	$f12, $5,$f14
n1,n2,d1	$4, $5,$f14
d1,n1,n2	$f12, $5,$6
s1,n1,n2	$f12, $5,$6
d1,s1,s2	$f12, $f13, $f14
s1,s2,d1	$f12, $f13, $f14
n1,n2,n3,n4	$4,$5,$6,$7
n1,n2,n3,d1	$4,$5,$6,$f15
n1,n2,n3,s1	$4,$5,$6, $f15
s1,s2,s3,s4	$f12, $f13,$f14,$f15
s1,n1,s2,n2	$f12, $5,$f14,$7
n1,s1,n2,s2	$4,$f13,$6,$f15
n1,s1,n2,n3	$4,$f13,$6,$7
d1,d2,d3,d4,d5	$f12, $f13, $f14, $f15, $f16
d1,d2,d3,d4,d5,s1,s2,s3,s4	$f12, $f13, $f14, $f15, $f16, $f17, $f18,$f19,stack
d1,d2,d3,s1,s2,s3,n1,n2,n3	$f12, $f13, $f14, $f15, $f16, $f17, $10,$11, stack

Implementation Differences

This section lists differences between the 32-bit and the 64-bit C implementations. Because all of the implementations adhere to the ANSI standard, and because C is a rigorously defined language designed to be portable, only a few differences exist between the 32-bit, n32, and 64-bit compiler implementations. The areas where differences can occur are in data types (by definition) and in areas where ANSI does not define the precise behavior of the language. In this area the n32 ABI is like the current 32-bit ABI. Thus, it is easier to port to the n32 ABI than to the 64-bit ABI.

Table 2-3 summarizes the differences in data types under the 32-bit and 64-bit data type models.

Table 2-3 Differences in Data Type Sizes

C type	32-bit and N32	64-bit
char	8	8
short int	16	16
int	32	32
long int	32	64
long long int	64	64
pointer	32	64
float	32	32
double	64	64
long double[a]	64 (128 in n32)	128

a. On ucode 32-bit compiles the **long double** data type generates a warning message indicating that the *long* qualifier is not supported. It is supported under n32.

As you can see in Table 2-3, **long ints**, **pointers**, and **long doubles** are different under the two models.

ABI Attribute Summary

Table 2-4 summarizes the important attributes for the o32 and n32/64-bit ABI implementations.

Table 2-4 ABI Attribute Summary

Attribute	o32	N32/64-bit
Width of integer parameters in registers	32 bits	64 bits
Stack parameter slot size	32 bits	64 bits
Types requiring multiple registers or stack slots	(long) double, long long	long double
Stack region alignment	16 byte	16 byte
Integer parameter registers	$4..$7	$4..$11
Floating point parameter registers (single/double precision)	$f12, $f14	$f12 .. $f19
Floating point parameters in Floating point registers (not varags)	first two only, not after integer parameters	any of first eight
Floating point parameters in Floating point registers (varags)	first two only, not after integer parameters	prototyped parameters only
Integer parameter register depends on earlier floating point parameter	Yes	No
Justification of parameters smaller than slot	integer: left float: N/A	integer: left float: Undecided
Placement of long double parameters	register: $f12/$f14 memory: aligned	register: even/odd memory: aligned

Table 2-4 (continued) ABI Attribute Summary

Attribute	o32	N32/64-bit
Sizes of structure components that are passed by registers	32 bits	64 bits
Are structure fields of type double in floating point registers?	Never	If not unioned
Justification of structs in partial registers	left	left
Who saves area for parameter registers	caller	callee, only if needed
Structure results limited to one or two FP fields in registers	FORTRAN COMPLEX only	Always
All types of structure results in registers	never	up to 128 bits
Structure results via first parameter result in $2	yes	no
Callee-saved FP registers	$f20..$f31 pairs	$f24..$f31 all (64-bit) $f20..$f31 even (n32)
Single exit block?	yes, sometimes ignored	no (option)
Return address register	ABI: $31 .mask support	any
GP register	caller-saved	callee saved
Use of odd FP registers	double halves	arbitrary
Use of 64-bit int registers	never (MIPS 1)	arbitrary

Chapter 3

N32 Compatibility, Porting, and Assembly Language Programming Issues

This chapter explains the levels of compatibility between o32, n32, and 64-bit programs. It also describes the porting procedure to follow and the changes to make when porting your application from o32 to n32.

Specifically, this chapter discusses the following topics:

- "Compatibility," which describes compatibility between o32, n32, and 64-bit programs.
- "N32 Porting Guidelines," which explains guidelines for porting high-level languages.
- "Assembly Language Programming Guidelines," which provides guidelines for writing portable assembly language code.

Compatibility

In order to execute different ABIs, support must exist at three levels:

- The operating system must support the ABI
- The libraries must support the ABI
- The application must be recompiled with a compiler that supports the ABI

Figure 3-1 shows how applications rely on library support to use the operating system resources that they need.

Note: Each o32, n32, and n64 application must be linked against unique libraries that conform to its respective ABI. As a result, you CANNOT mix and match objects files or libraries from any of the different ABIs.

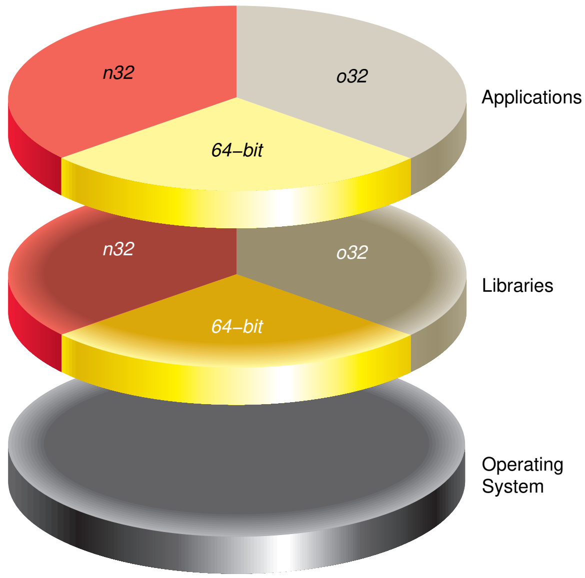


Figure 3-1 Application Support Under Different ABIs

Figure 3-2 illustrates the locations of the different libraries.

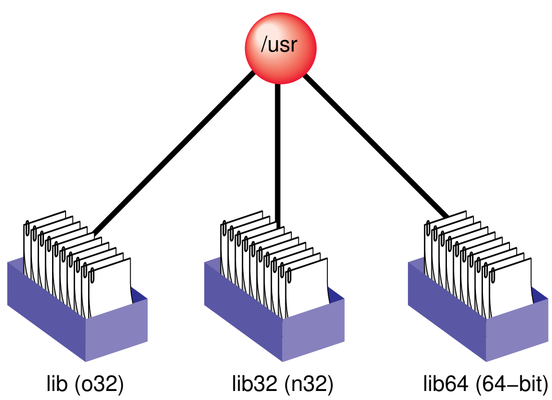


Figure 3-2 Library Locations for the Different ABIs

An operating system that supports all three ABIs is also needed for running the application. Consequently, all applications that want to use the features of n32 must be ported. The next section covers the steps in porting an application to the n32 ABI.

N32 Porting Guidelines

This section describes the guidelines/steps necessary to port IRIX 5.*x* 32-bit applications to n32. Typically, any porting project can be divided into the following tasks:

- Identifying and creating the necessary porting environment (see "Porting Environment")
- Identifying and making the necessary source code changes (see "Source Code Changes")
- Rebuilding the application for the target machine (see "Build Procedure")
- Analyzing and debugging runtime issues (see "Runtime Issues")

Each of these tasks is described below.

Porting Environment

The porting environment consists of a compiler and associated tools, *include* files, libraries, and makefiles, all of which are necessary to compile and build your application. Version 6.1 of the MIPSpro compiler supports the generation of n32 code. To generate this code, you must:

- Check all libraries needed by your application to make sure they are recompiled n32. The default root location for n32 libraries is */usr/lib32*. If the n32 library needed by your application does not exist, recompile the library for n32.
- Modify existing makefiles (or set environment variables) to reflect the locations of these n32 libraries, if they use **-L** to specify library locations.

Source Code Changes

Since no differences occur in the sizes of fundamental types between o32 and n32, porting to n32 requires no source code changes for applications written in high-level languages such as C, C++, and Fortran.

The only exception to this is that C functions that accept variable numbers of floating point arguments must be prototyped.

Assembly language code, however, must be modified to reflect the new subprogram interface. Guidelines for following this interface are described in "Assembly Language Programming Guidelines."

Build Procedure

Recompiling for n32 involves either setting the *-n32* argument in the compiler invocation or running the compiler with the environment variable *SGI_ABI* set to *-n32*. That's all you must do after you set up a native n32 compilation environment (that is, all necessary libraries and **include** files reside on the host system).

Runtime Issues

Applications that are ported to n32 may get different results than their o32 counterparts. Reasons for this include:

- Differences in algorithms used by n32 libraries and o32 libraries
- Operand reassociation or reduction performed by the optimizer for n32.
- Hardware differences of the R8000 (madd instructions round slightly differently than a multiply instruction followed by an add instruction).

For more information refer to Chapter 5 of the *MIPSpro 64-bit Porting and Transition Guide*.

Assembly Language Programming Guidelines

This section describes techniques for writing assembler code that can be compiled and run as either an o32 or n32 executable. These techniques are based on using certain predefined variables of the compiler, and on macros defined in <*sys/asm.h*> and <*sys/regdef.h*>, which rely on those compiler predefines. Together, they enable a fairly easy conversion of existing assembly code to run under the n32 ABI. They also allow retargeted assembler code to look fairly uniform in the way it is converted.

Predefined Variables

The predefined variables are set by the compiler or assembler when they are invoked. These variables will have different values depending on which switches are used on the command line. These variables can then be used by conditional compilation directives such as **#ifdef** to determine which code gets compiled or assembled for a particular ABI. You can see the values of these predefined variables by adding the *-show* switch to a compilation command. The variable that can help distinguish between on32 and n32 compilations are the following:

```
For MipsI o32 executables:
    -D_MIPS_FPSET=16
    -D_MIPS_ISA=_MIPS_ISA_MIPS1
    -D_MIPS_SIM=_MIPS_SIM_ABI32


For MipsIV N32 executables:
    -D_MIPS_FPSET=32
    -D_MIPS_ISA=_MIPS_ISA_MIPS4
    -D_MIPS_SIM=_MIPS_SIM_ABIN32
```

The explanation of these predefines is as follows:

- *MIPS_ISA* is Mips Instruction Set Architecture. *MIPS_ISA_MIPS1* and *MIPS_ISA_MIPS*4 are the most common variants for assembler code. *MIPS_ISA_MIPS4* is the ISA for R8000 applications.
- *MIPS_SIM* denotes the Mips Subprogram Interface Model.This describes the subroutine linkage convention and register naming/usage convention. It indicates o32 n32 or n64.
- *_MIPS_FPSET* describes the number of floating point registers. The Mips IV compilation model makes use of the extended floating point registers available on the R4000 and beyond.

The following code fragment shows an example of the use of these macros:

```
#if (_MIPS_ISA == _MIPS_ISA_MIPS1 || _MIPS_ISA == _MIPS_ISA_MIPS2)
#define SZREG           4
#endif

#if (_MIPS_ISA == _MIPS_ISA_MIPS3 || _MIPS_ISA == _MIPS_ISA_MIPS4)
#define SZREG           8
#endif
```

N32 Implications for Assembly Code

Four implications to writing assembly language code for n32 are:

- The first requires you to use a different convention to save the global pointer register (*$gp*) as explained in “Caller $gp (o32) vs. Callee Saved $gp (n32 and n64).”
- The second deals with different register sizes as explained in “Different Register Sizes.”
- The third requires you to use a different subroutine linkage convention as explained in “Using a Different Subroutine Linkage.”
- The fourth restricts your use of *lwc1* instructions to access floating point register pairs but allows you to use more floating point registers as described in “Using More Floating Point Registers.”

Caller $gp (o32) vs. Callee Saved $gp (n32 and n64)

The *$gp* register is used to point to the Global Offset Table (GOT). The GOT stores addresses of subroutines and static data for runtime linking. Since each DSO has its own GOT, the *$gp* register must be saved across function calls. Two conventions are used to save the *$gp* register.

Under the first convention, called caller saved *$gp*, each time a function call is made, the calling routine saves the *$gp* and then restores it after the called function returns. To facilitate this two assembly language pseudo instructions are used. The first, *.cpload*, is used at the beginning of a function and sets up the *$gp* with the correct value. The second, *.cprestore*, saves the value of *$gp* on the stack at an offset specified by the user. It also causes the assembler to emit code to restore *$gp* after each call to a subroutine.

The formats for correct usage of the *.cpload* and *.cprestore* instructions are shown below:

.cpload reg	*reg* is *t9* by convention
*.cprestore offse*t	*offset* refers to the stack offset where *$gp* is saved

Under the second convention, called callee saved *$gp*, the responsibility for saving the *$gp* register is placed on the called function. As a result, the called function needs to save the *$gp* register when it first starts executing. It must also restore it, just before it returns. To accomplish this the *.cpsetup* pseudo assembly language instruction is used. Its usage is shown below:

.cpsetup reg, offset, proc_name
reg is *t9* by convention
offset refers to the stack offset where *$gp* is saved
proc_name refers to the name of the subroutine

Note: You must create a stack frame by subtracting the appropriate value from the *$sp* register before using the directives which save the *$gp* on the stack.

In order to facilitate writing assembly language code for both conventions several macros have been defined in <*sys/asm.h*>. The macros **SETUP_GP, SETUP_GPX, SETUP_GP_L,** and **SAVE_GP** are defined under o32 and provide the necessary functionality to support a caller saved *$gp* environment. Under n32, these macros are null. However, *SETUP_GP64*, *SETUP_GPX64*, *SETUP_GPX64_L*, and *RESTORE_GP64* provide the functionality to support a callee saved environment. These same macros are null for o32. An example of the use of these macros and their definition can be found at the end of the fourth section.

Different Register Sizes

Under n32, registers are 64 bits wide; under o32, they are 32 bits wide. In order to properly manipulate these register under n32, you must use the 64-bit forms of the basic load, store, and arithmetic operation instructions. To allow the same source to be assembled for either o32 or n32, a set of macros has been defined in <*sys/asm.h*>. These macros use the correct instruction form for 32-bit or 64-bit operation. These macros include the following:

- REG_S expands to *sw* for o32 and to *sd* for n32.
- REG_L expands to *lw* for o32 and to *ld* for n32.
- PTR_L expands to *lw* for o32 and to *ld* for n32.
- PTR_S expands to *sw* for o32 and to *sd* for n32.
- PTR_SUBU expands to *subu* for o32 and to *dsubu* for n32.
- PTR_ADDU expands to *addu* for o32 and to *daddu* for n32.

Using a Different Subroutine Linkage

Under n32, more registers are used to pass arguments to called subroutines. The registers that are saved by the calling and called subroutines are also different under this convention, which is described in detail in Chapter 2, "Calling Convention Implementations." As a result, a different register naming convention exists. The compiler predefine *_MIPS_SIM* enables macros in <*sys/asm.h*> and <*sys/regdef.h*>. Some important ramifications of the subroutine linkage convention are outlined below.

The *_MIPS_SIM_NABI32* model (n32), defines 4 additional argument registers for a total of 8 argument registers: **$4 .. $11**. The additional 4 argument registers come at the expense of the temp registers in <*sys/regdef.h*>. In this model, there are no registers **t4 .. t7**, so any code using these registers does not compile under this model. Similarly, the register names **a4 .. a7** are not available under the *_MIPS_SIM_ABI32* model. (Note that those temporary registers are not lost -- the argument registers can serve as scratch registers also, with certain constraints.)

To make it easier to convert assembler code, the new names **ta0**, **ta1**, **ta2**, and **ta3** are available under both *_MIPS_SIM* models. These alias with **t4 .. t7** in the o32 ABI, and with **a4 ..a7** in the n32 ABI.

Another facet of the linkage convention is that the caller no longer has to reserve space for a called function in which to store its arguments. The called routine allocates space for storing its arguments on its own stack, if desired. The NARGSAVE define in <*sys/asm.h*> helps with this.

The following example handles assembly language coding issues for n32 and KPIC (KPIC requires that the *asm* coder deals with PIC issues). It creates a template for the start and end of a generic assembly language routine.

The template is followed by relevant defines and macros from <*sys/asm.h*>.

```
#include <sys/regdef.h>
#include <sys/asm.h>
#include <sys/fpregdef.h>

LOCALSZ= 7     # save gp ra and any other needed registers
/* For this example 7 items are saved on the stack */
/* To access the appropriate item use the offsets below */
FRAMESZ= (((NARGSAVE+LOCALSZ)*SZREG)+ALSZ)&ALMASK
RAOFF=  FRAMESZ-(1*SZREG)
GPOFF=  FRAMESZ-(4*SZREG)
```

```
A0OFF=  FRAMESZ-(5*SZREG)
A1OFF=  FRAMESZ-(6*SZREG)
T0OFF=  FRAMESZ-(7*SZREG)

NESTED(asmfunc,FRAMESZ,ra)
        move t0, gp   # save entering gp
                      # SIM_ABI64 has gp callee save
                      # no harm for SIM_ABI32
        SETUP_GPX(t8)
        PTR_SUBU sp,FRAMESZ
        SETUP_GP64(GPOFF,_sigsetjmp)
        SAVE_GP(GPOFF)
/* Save registers as needed here */
        REG_S ra,RAOFF(sp)
        REG_S a0,A0OFF(sp)
        REG_S a1,A1OFF(sp)
        REG_S t0,T0OFF(sp)

/* do real work here */
/* safe to call other functions */

/* restore saved regsisters as needed here */
        REG_L ra,RAOFF(sp)
        REG_L a0,A0OFF(sp)
        REG_L a1,A1OFF(sp)
        REG_L t0,T0OFF(sp)

/* setup return address, $gp and stack pointer */
REG_L    ra,RAOFF(sp)
RESTORE_GP64
PTR_ADDU sp,FRAMESZ

        bne      v0,zero,err
        j        ra

        END(asmfunc)


/* The following macro definitions are */
/* from /usr/include/sys/asm.h */

#if (_MIPS_SIM == _MIPS_SIM_ABI32)
/*
 * Set gp when at 1st instruction
 */
```

```
#define SETUP_GP     \
            .set noreorder;    \
            .cpload t9;     \
            .set reorder

/* Set gp when not at 1st instruction */
#define SETUP_GPX(r)     \
            .set noreorder;    \
            move r, ra;  /* save old ra */ \
            bal 10f;  /* find addr of cpload */\
            nop;      \
10:       \
            .cpload ra;     \
            move ra, r;     \
        .set reorder;

#define SETUP_GPX_L(r,l)    \
        .set noreorder;    \
        move r, ra;  /* save old ra */ \
        bal l;  /* find addr of cpload */\
        nop;      \
l:       \
        .cpload ra;     \
        move ra, r;     \
        .set reorder;

#define SAVE_GP(x)     \
        .cprestore x; /* save gp trigger t9/jalr conversion */

#define SETUP_GP64(a,b)
#define SETUP_GPX64(a,b)
#define SETUP_GPX64_L(cp_reg,ra_save, l)
#define RESTORE_GP64
#define USE_ALT_CP(a)

#else /* (_MIPS_SIM == _MIPS_SIM_ABI64) || (_MIPS_SIM ==
_MIPS_SIM_NABI32) */
/*
 * For callee-saved gp calling convention:
 */
#define SETUP_GP
#define SETUP_GPX(r)
#define SETUP_GPX_L(r,l)
#define SAVE_GP(x)
```

```
#define SETUP_GP64(gpoffset,proc)   \
        .cpsetup t9, gpoffset, proc

#define SETUP_GPX64(cp_reg,ra_save)   \
        move ra_save, ra;     /* save old ra */ \
        .set noreorder;    \
        bal 10f;      /* find addr of .cpsetup */ \
        nop;      \
10:       \
        .set reorder;    \
        .cpsetup ra, cp_reg, 10b;  \
        move ra, ra_save

#define SETUP_GPX64_L(cp_reg,ra_save, l)  \
        move ra_save, ra;     /* save old ra */ \
        .set noreorder;    \
        bal l;      /* find addr of .cpsetup */ \
        nop;      \
l:       \
        .set reorder;    \
        .cpsetup ra, cp_reg, l;   \
        move ra, ra_save

#define RESTORE_GP64     \
        .cpreturn

#define USE_ALT_CP(reg)     \
        .cplocal reg     /* use alternate register for  context
pointer */

#endif /* _MIPS_SIM != _MIPS_SIM_ABI32 */

/*
 * Stack Frame Definitions
 */

#if (_MIPS_SIM == _MIPS_SIM_ABI32)
#define NARGSAVE 4 /* space for 4 arg regs must be alloc*/
#endif
#if (_MIPS_SIM == _MIPS_SIM_ABI64 || _MIPS_SIM == _MIPS_SIM_NABI32)
#define NARGSAVE 0 /* no caller responsibilities */
#endif

#define ALSZ  15 /* align on 16 byte boundary */
#define ALMASK  ~0xf
```

```
#if (_MIPS_ISA == _MIPS_ISA_MIPS1 || _MIPS_ISA == _MIPS_ISA_MIPS2)
#define SZREG  4
#endif

#if (_MIPS_ISA == _MIPS_ISA_MIPS3 || _MIPS_ISA == _MIPS_ISA_MIPS4)
#define SZREG  8
#endif
```

Using More Floating Point Registers

On the R4000 and later generation MIPS microprocessors, the FPU provides:

- 16 64-bit Floating Point registers (FPRs) each made up of a pair of 32-bit floating point general purpose register when the FR bit in the Status register equals 0, or
- 32 64-bit Floating Point registers (FPRs) each corresponding to a 64-bit floating point general purpose register when the FR bit in the Status register equals 1

For more information about the FPU of the R4000 refer to Chapter 6 of the *MIPS R4000 User's Manual*.

Under o32, the FR bit is set to 0. As a result, o32 provides only 16 registers for double precision calculations. Under o32, double precision instructions must refer to the even numbered floating point general purpose register. A major implication of this is that code written for the MIPS I instruction set treated a double precision floating point register as an odd and even pair of single precision floating point registers. It would typically use sequences of the following instructions to load and store double precision registers.

```
lwc1 $f4, 4(a0)
lwc1 $f5, 0(a0)
...
swc1 $f4, 4(t0)
swc1 $f5, 0(t0)
```

Under n32, however, the FR bit is set to 1. As a result, n32 provides all 32 floating point general purpose registers for double precision calculations. Since *$f4* and *$f5* refer to different double precision registers, the code sequence above will not work under n32. It can be replaced with the following:

```
l.d $f14, 0(a0)
...
s.d $f14, 0(t0)
```

The assembler will automatically generate pairs of LWC1 instructions for MIPS I and use the LDC1 instruction for MIPS II and above.

On the other hand, you can use these additional odd numbered registers to improve performance of double precision code.

The following example taken form <*libm43/z_abs.s*> can be assembled for o32 or n32. When assembled **–n32**, it uses odd double precision floating point registers as well as the macros from <*sys/asm.h*> to adhere to the subroutine interface convention.

```
#include <regdef.h>
#include <sys/asm.h>

        PICOPT
        .text

.weakext  z_abs_, __z_abs_
#define z_abs_  __z_abs_

.extern __hypot

LOCALSZ = 10
FSIZE = (((NARGSAVE+LOCALSZ)*SZREG)+ALSZ)&ALMASK
RAOFF= FSIZE - SZREG
GPOFF= FSIZE - (2*SZREG)

#if (_MIPS_SIM == _MIPS_SIM_ABI64 || _MIPS_SIM == _MIPS_SIM_NABI32)

NESTED(z_abs_,FSIZE,ra)

       PTR_SUBU sp,FSIZE
       SETUP_GP64(GPOFF,z_abs_)
       REG_S    ra, RAOFF(sp)
       l.d      $f12, 0(a0)
       l.d      $f13, 8(a0)
       jal      __hypot
       REG_L    ra, RAOFF(sp)
       RESTORE_GP64
       PTR_ADDU sp, FSIZE
       j        ra
END(z_abs_)

#elif (_MIPS_SIM == _MIPS_SIM_ABI32)
```

```
NESTED(z_abs_,FSIZE,ra)

       SETUP_GP
       PTR_SUBU sp,FSIZE
       SAVE_GP(GPOFF)
       REG_S   ra, RAOFF(sp)
       l.d     $f12, 0(a0)
       l.d     $f14, 8(a0)
       jal     hypot
       REG_L   ra, RAOFF(sp)
       PTR_ADDU sp, FSIZE
       j       ra

END(z_abs_)

#endif
```

Chapter 4

N32 Examples and Case Studies

This chapter provides examples and case studies of programs that have been converted from o32 to n32. Each step in the conversion is presented and examined in detail. Examples include:

- "An Example Application"
- "Building and Running the o32 Application"
- "Porting Issues"
- "Building and Running the N32 Application"
- "Building Multiple Versions of the Application"

An Example Application

An examination of the following application, *app1*, illustrates the steps necessary to port from o32 to n32. As you can see, *app1* is trivial in functionality, but it is constructed to point out several of the issues involved in converting code from o32 to n32.

App1 contains the following files:

- *main.c*, which contains the function *main()*.
- *foo.c,* which contains *foo()* a varargs function.
- *gp.s*, which contains the assembly language leaf routine, *get_gp()*. This function returns the value of the global pointer register (*$gp*).
- *regs.s*, which contains the assembly language function **regs()**. This function is linked separately into its own DSO. The function **regs()** returns the value of *$gp*, the return address register (*$ra*), and the stack pointer (*$sp*). This function also makes calls to the *libc* routines **malloc()** and **free()** as well as calculating the sum of two double precision values passed to it as arguments and returns the sum through a pointer that is also passed to it as an argument.

Figure 4-1 shows a call tree is for the *app1* program. It illustrates that **main()** calls *get_gp()*, *foo()* and *printf()*. The function **foo()** calls **regs()** and **printf()**, while **regs()** calls **malloc()** and **free()**. The figure also shows that **app1** is linked against two shared objects, *libc.so* and *regs.so*.

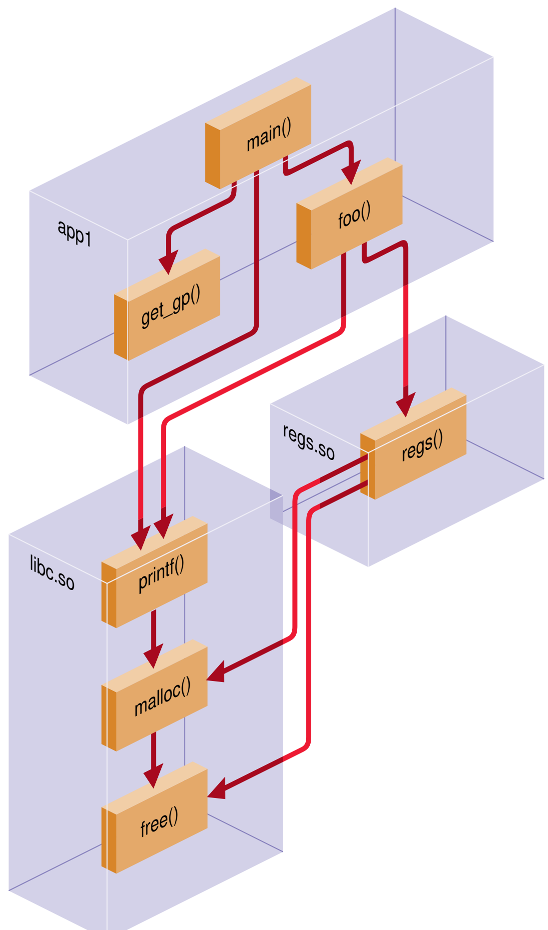


Figure 4-1 Call Tree for App1

The source code for the original versions of *main.c*, *foo.c*, *gp.s*. and *regs.s* are shown below.

```
/* main.c */
extern void foo();
```

```
main()
{
   unsigned gp,ra,sp, get_regs();
   double d1 = 1.0;
   double d2 = 2.0;
   double res;

   gp = get_gp();
   printf(“gp is 0x%x\n”, gp);
   foo(7, 3.14, &gp, &ra,
       &sp, d1, &d2, &res);

}

/* foo.c */

#include <stdarg.h>

void foo(int narg, ...)
{
  va_list ap;
  double d1;
  double daddr1, *daddr2, *resaddr;
  unsigned *gp, *ra, *sp;

  va_start(ap, narg);
  printf(“Number of Arguments is: %d\n”,narg);

  d1 = va_arg(ap, double);
  printf(“%e\n”,d1);

  gp = va_arg(ap, unsigned*);
  ra = va_arg(ap, unsigned*);
  sp = va_arg(ap, unsigned*);

  daddr1 = va_arg(ap, double);
  daddr2 = va_arg(ap, double*);
  resaddr = va_arg(ap, double*);

  printf(“first  double precision argument is %e\n”,daddr1);
  printf(“second double precision argument is %e\n”,*daddr2);

  regs(gp, ra, sp, daddr1, daddr2, resaddr);
```

```
  printf("Back from assembly routine\n");
  printf("gp is 0x%x\n",*gp);
  printf("ra is 0x%x\n",*ra);
  printf("sp is 0x%x\n",*sp);
  printf("result of double precision add is %e\n",*resaddr);

  va_end(ap);
}

/* gp.s */
#include <regdef.h>
#include <asm.h>

LEAF(get_gp)
        move  v0, gp
        j     ra
        .end  get_gp

/* regs.s */
#include <regdef.h>

        .text
        .globlregs            # make regs external
        .entregs 2
regs:
        .set noreorder
        .cploadt9             # setup gp
        .set reorder
        subu   sp, 32         # create stack frame
        sw     ra, 28(sp)     # save return address
        .cprestore     24     # for caller saved gp
                              # save gp 24(sp)
        sw     gp, 0(a0)      # return gp in first arg
        sw     ra, 0(a1)      # return ra in second arg
        sw     sp, 0(a2)      # return sp in third arg

        li     a0, 1000       # call libc routines
        jal    malloc         # for illustrative purposes
        move   a0, v0         # to make regs
        jal    free           # a nested function

        lw     t0, 56(sp)     # get fifth argument from stack
        lwc1   $f4, 4(t0)     # load it in fp register
        lwc1   $f5, 0(t0)     # fp values are stored in LE
                              # format
```

```
    lwc1   $f6, 52(sp)  # get fourth argument from stack
    lwc1   $f7, 48(sp)  # fp values are stored in LE
                        # format

    add.d  $f8, $f4, $f6  # do the calculation
    lw     t0, 60(sp)     # get the sixth argument
                          # from the stack
    swc1   $f8, 4(t0)     # save the result
    swc1   $f9, 0(t0)     # fp values are stored in LE

lw     ra, 28(sp)     # get return address
addu   sp, 32         # pop stack
j      ra             # return to caller
.end regs
```

Building and Running the o32 Application

The commands used to build *app1* are shown below. As mentioned previously, *regs.s* is compiled and linked separately into its own DSO, while *main.c*, *foo.c* and *gp.s* are compiled and linked together.

```
%cc -32 -O -shared -o regs.so regs.s

%cc -32 -O -o app1 main.c foo.c gp2.s regs.so
```

In order to run the application, the *LD_LIBRARY_PATH* environment variable must be set to the directory where regs.so resides.

```
%setenv LD_LIBRARY_PATH .
```

Running the application produces the following results. Note that the value of *$gp* is different when code is executing in the *regs.so* DSO.

```
%app1

   gp is 0x100090f0

   Number of Arguments is: 7

   3.140000e+00

   first  double precision argument is 1.000000e+00

   second double precision argument is 2.000000e+00

   Back from assembly routine

   gp is 0x5fff8ff0
```

```
ra is 0x400d10
sp is 0x7fff2e28
result of double precision add is 3.000000e+00
```

Porting Issues

If the files *foo.c* and *main.c* were recompiled for n32, the resulting executable would not work for a variety of reasons. Each reason is examined below and a solution is given. The resulting set of files will work when compiled either for o32 or for n32. This section covers:

- "Varargs Routines"
- "Assembly Language Issues"

Varargs Routines

Attempting to recompile *main.c foo.c* **–n32** results in two sets of warnings shown below:

```
%cc -n32 -O -o app1 main.c foo.c gp2.s
    foo.c
    !!! Warning (user routine 'foo'):
    !!! Prototype required when passing floating point parameter to
    varargs routine: printf
    !!! Use '#include <stdio.h>' (see ANSI X3.159-1989, Section 3.3.2.2)
    ld32: WARNING 110: floating-point parameters exist in the call for
    "foo", a VARARG function, in object "main.o" without a prototype --
    would result in invalid result.  Definition can be found in object
    "foo.o"
    ld32: WARNING 110: floating-point parameters exist in the call for
    "printf", a VARARG function, in object "foo.o" without a prototype
    -- would result in invalid result.  Definition can be found in
    object "/usr/lib32/mips4/libc.so"
```

The first warning points out that **printf()** is a varargs routine that is being called with floating point arguments. Under these circumstances, a prototype must exist for **printf()**. This is accomplished by adding the following line to the top of *foo.c*:

```
#include <stdio.h>
```

The second warning points out that *foo()* is also a varargs routine with floating point arguments and must also be prototyped. This is fixed by changing the declaration of *foo()* in *main.c* to:

```
foo(int, ...)
```

For completeness, <*stdio.h*> is also included in *main.c* to provide a prototype for *printf()* should it ever use floating point arguments.

As a result of these small changes, the C files are fixed and ready to be compiled **–n32**. The new versions are shown below.

```
/* main.c */
#include <stdio.h>
extern void foo(int, ...);

main()
{
   unsigned gp,ra,sp, get_regs();
   double d1 = 1.0;
   double d2 = 2.0;
   double res;

   gp = get_gp();
   printf("gp is 0x%x\n", gp);

   foo(7, 3.14, &gp, &ra,
       &sp, d1, &d2, &res);

}


/* foo.c */
#include <stdio.h>
#include <stdarg.h>

void foo(int narg, ...)
{

  va_list ap;
  double d1;
  double daddr1, *daddr2, *resaddr;
  unsigned *gp, *ra, *sp;

  va_start(ap, narg);
```

```
  printf("Number of Arguments is: %d\n",narg);

  d1 = va_arg(ap, double);
  printf("%e\n",d1);

  gp = va_arg(ap, unsigned*);
  ra = va_arg(ap, unsigned*);
  sp = va_arg(ap, unsigned*);

  daddr1 = va_arg(ap, double);
  daddr2 = va_arg(ap, double*);
  resaddr = va_arg(ap, double*);

  printf("first  double precision argument is %e\n",daddr1);
  printf("second double precision argument is %e\n",*daddr2);

  regs(gp, ra, sp, daddr1, daddr2, resaddr);
  printf("Back from assembly routine\n");
  printf("gp is 0x%x\n",*gp);
  printf("ra is 0x%x\n",*ra);
  printf("sp is 0x%x\n",*sp);
  printf("result of double precision add is %e\n",*resaddr);

  va_end(ap);
}
```

Assembly Language Issues

Since **get_gp()** is a leaf routine that is linked in the same DSO where it is called, no changes are required to port it to n32. However, you have to recompile it.

Since **get_gp()** is a leaf routine that is linked in the same DSO where it is called, no changes are required to port it to n32. However, you have to recompile it.

On the other hand, **regs()** requires a lot of work. The issues that need to be addressed are detailed below.

gp register

As explained throughout this book, the o32 ABI follows the convention that *$gp* (global pointer register) is caller saved. This means that the global pointer is saved before each function call and restored after each function call returns. This is accomplished by using the *.cpload* and *.cprestore* assembler pseudo instructions respectively. Both lines are present in the original version of *regs.s*.

The n32 ABI, on the other hand, follows the convention that *$gp* is callee saved. This means that *$gp* is saved at the beginning of each routine and restored right before that routine itself returns. This is accomplished through the use of *.cpsetup*, an assembler pseudo instruction.

The recommended way to deal with these various pseudo instructions is to use the macros provided in <*sys/asm.h*>. The macros below will provide correct use of these pseudo instructions whether compiled for o32 or for n32.

- SETUP_GP expands to the *.cpload t9* pseudo instruction for o32. For n32 it is null.
- SAVE_GP(GPOFF) expands to the *.cprestore* pseudo instruction for o32. For n32 it is null.
- SETUP_GP64(GPOFF, regs) expands to the *.cpsetup* pseudo instruction for n32. For o32 it is null.

Register Size

Under o32, registers are 32 bits wide. Under n32, they are 64 bits wide. As a result, assembly language routines must be careful in the way they operate on registers. The following macros defined in <*sys/asm.h*> are useful because they expand to 32-bit instructions under o32 and to 64-bit instructions under n32.

- REG_S expands to *sw* for o32 and to *sd* for n32.
- REG_L expands to *lw* for o32 and to *ld* for n32.
- PTR_SUBU expands to *subu* for o32 and to *dsubu* for n32.
- PTR_ADDU expands to *addu* for o32 and to *daddu* for n32.

Argument Passing

The **get_regs()** function in *regs.s* is called with six arguments. Under o32, the first three are passed in registers *a0* through *a2*. The fourth argument (a double precision parameter) is passed at offset 16 relative to the stack. The fifth and sixth arguments are passed at offsets 24 and 28 relative to the stack, respectively. Under n32, however, all of the arguments are passed in registers. The first three arguments are passed in registers *a0* through *a2* as they were under o32. The next parameter is passed in register *$f15*. The last two parameters are passed in registers *a4* and *a5* respectively. Table 4-1 summarizes where each of the arguments are passed under the two conventions.

Table 4-1 Argument Passing

Argument	o32	n32
argument1	a0	a0
argument2	a1	a1
argument3	a2	a2
argument4	$sp+16	$f15
argument5	$sp+24	a4
argument6	$sp+28	a5

Note: Under o32, there are no *a4* and *a5* registers, but under n32 they must be saved on the stack because they are used after calls to an external function.

The code fragment that illustrates accessing the arguments under n32 is shown below:

```
        mov.d   $f4,$f15            # 5th argument in 5th fp
                                    # arg. register
        l.d     $f6,0(a4)           # fourth argument in
                                    # fourth arg. register
        s.d     $f8,0(a5)           # save in 6th arg. reg
```

Extra Floating point Registers

As explained in Chapter 3, "N32 Compatibility, Porting, and Assembly Language Programming Issues," floating point registers are 64 bits wide under n32. They are no longer accessed as pairs of single precision registers for double precision calculations. As a result, the section of code that uses the pairs of *lwc1* or *swc1* instructions must be changed. The simplest way to accomplish this is to use the *l.d* assembly language instruction. This instruction expands to two *lwc1* instructions under **-mips1**; under **-mips2** and above, it expands to the *ldc1* instruction.

Putting it together

The new version of *regs.s* is shown below. It is coded so that it will compile and execute for either o32 or n32 environments.

```
/* regs.s */
#include <sys/regdef.h>
#include <sys/asm.h>

.text

LOCALSZ=5            # save ra, a4, a5, gp, $f15

FRAMESZ= (((NARGSAVE+LOCALSZ)*SZREG)+ALSZ)&ALMASK

RAOFF=FRAMESZ-(1*SZREG)     # stack offset where ra is saved
A4OFF=FRAMESZ-(2*SZREG)     # stack offset where a4 is saved
A5OFF=FRAMESZ-(3*SZREG)     # stack offset where a5 is saved
GPOFF=FRAMESZ-(4*SZREG)     # stack offset where gp is saved
FPOFF=FRAMESZ-(5*SZREG)     # stack offset where $f15 is
                            # saved
                            # a4, a5, and $f15 don't have to
                            # be saved, but no harm done in
                            # doing so
NESTED(regs, FRAMESZ, ra)
                            # define regs to be a nested
                            # function
    SETUP_GP                # used for caller saved gp
    PTR_SUBU sp,FRAMESZ     # setup stack frame
    SETUP_GP64(GPOFF, regs) # used for callee saved gp
    SAVE_GP(GPOFF)          # used for caller saved gp

    REG_S   ra, RAOFF(sp)   # save ra on stack

#if (_MIPS_SIM != _MIPS_SIM_ABI32)
                            # not needed for o32
    REG_S   a4, A4OFF(sp)   # save a4 on stack (argument 4)
    REG_S   a5, A5OFF(sp)   # save a5 on stack (argument 5)
    s.d     $f15,FPOFF(sp)  # save $f15 on stack (argument 6)
#endif /* _MIPS_SIM != _MIPS_SIM_ABI32 */

    sw      gp, 0(a0)       # return gp in first arg
    sw      ra, 0(a1)       # return ra in second arg
    sw      sp, 0(a2)       # return sp in third arg

    li      a0, 1000        # call malloc
    jal     malloc          # for illustration purposes only

    move    a0, v0          # call free
    jal     free            # go into libc.so twice
```

```
                                # this is why a4, a5, $f15
                                # had to be saved

#if (_MIPS_SIM != _MIPS_SIM_ABI32)
                                # not needed for o32
        l.d     $f15,FPOFF(sp)  # restore $f15 (argument #6)
        REG_L   a4, A4OFF(sp)   # restore a4   (argument #4)
        REG_L   a5, A5OFF(sp)   # restore a5   (argument #5)
#endif /* _MIPS_SIM != _MIPS_SIM_ABI32 */

#if (_MIPS_SIM == _MIPS_SIM_ABI32)
                                # for o32 arguments will
                                # need to be pulled from the
                                # stack
        lw      t0,FRAMESZ+24(sp) # fifth argument is 24
                                  # relative to original sp
        l.d     $f4,0(t0)         # use l.d for correct code
                                  # on both mips1 & mips2
        l.d     $f6,FRAMESZ+16(sp)  # fourth argument is 16
                                    # relative to original sp
        add.d   $f8, $f4, $f6       # do the calculation
        lw      t0,FRAMESZ+28(sp)   # sixth argument is 28
                                    # relative to original sp
        s.d     $f8,0(t0)           # save the result there
#else
                                    # n32 args are in regs
        mov.d   $f4,$f15            # 5th argument in 5th fp
                                    # arg. register
        l.d     $f6,0(a4)           # fourth argument in
                                    # fourth arg. register
        add.d   $f8, $f4, $f6       # do the calculation
        s.d     $f8,0(a5)           # save in 6th arg. reg
#endif /* _MIPS_SIM != _MIPS_SIM_ABI32 */

        REG_L   ra, RAOFF(sp)     # restore return address
        RESTORE_GP64              # restore gp for n32
                                  # (callee saved)
        PTR_ADDU sp,FRAMESZ       # pop stack
        j        ra               # return to caller
.endregs
```

Building and Running the N32 Application

The commands for building an n32 version of *app1* are shown below. The only difference is in the use of the **–n32** argument on the compiler command line. If *app1* was a large application using many libraries, the command line or makefile would possibly need to be modified to refer to the correct library paths. In the case of *app1* the correct *libc.so* is automatically used as a result of the **–n32** argument.

```
%cc –n32 –O –shared –o regs.so regs.s
%cc –n32 –O –o app1 main.c foo.c gp.s regs.so
```

In order to run the application, the *LD_LIBRARY_PATH* environment variable must again be set to the directory where *regs.so* resides.

```
%setenv LD_LIBRARY_PATH .
```

Running the application produces the following results. Note that the values of some of the returned registers are different from those returned by the o32 version of *app1*.

```
%app1
   gp is 0x100090e8
   Number of Arguments is: 7
   3.140000e+00
   first  double precision argument is 1.000000e+00
   second double precision argument is 2.000000e+00
   Back from assembly routine
   gp is 0x5fff8ff0
   ra is 0x10000d68
   sp is 0x7fff2e30
   result of double precision add is 3.000000e+00
```

Building Multiple Versions of the Application

Following the procedure above generates new n32 versions of *app1* and *regs.so*; however, they overwrite the old o32 versions. To build multiple versions of *app1*, use one of the following methods:

- Use different names for the n32 and o32 versions of the application and DSO. This method is simple, but for large applications, you must rename each DSO.
- Create separate directories for the o32 and n32 applications and DSOs, respectively. Modify the commands above or makefiles to create *app1* and *reg.so* in the appropriate directory. This method offers more organization than the approach above, but you must set the *LD_LIBRARY_PATH* accordingly.
- Create separate directories as specified above, but add the **–rpath** argument to the command line that builds *app1*.

Index

O

P

Q

R

S

T

U

V

Tell Us About This Manual

As a user of Silicon Graphics documentation, your comments are important to us. They help us to better understand your needs and to improve the quality of our documentation.

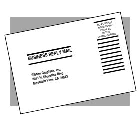
Any information that you provide will be useful. Here is a list of suggested topics to comment on:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

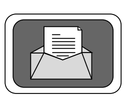
Please include the title and part number of the document you are commenting on. The part number for this document is 007-2816-002.

Thank you!

Three Ways to Reach Us

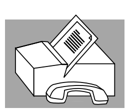


The **postcard** opposite this page has space for your comments. Write your comments on the postage-paid card for your country, then detach and mail it. If your country is not listed, either use the international card and apply the necessary postage or use electronic mail or FAX for your reply.



If **electronic mail** is available to you, write your comments in an e-mail message and mail it to either of these addresses:

- If you are on the Internet, use this address: techpubs@sgi.com
- For UUCP mail, use this address through any backbone site: *[your_site]*!sgi!techpubs



You can forward your comments (or annotated copies of manual pages) to Technical Publications at this **FAX** number:

415 965-0964