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.

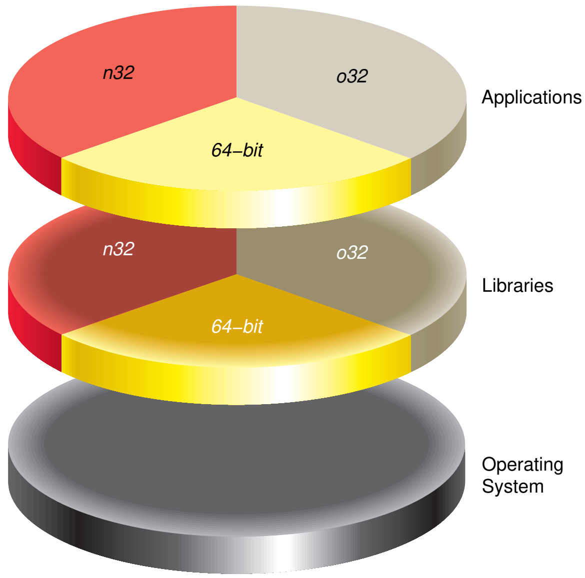

Figure 3-1 Application Support Under Different ABIs

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

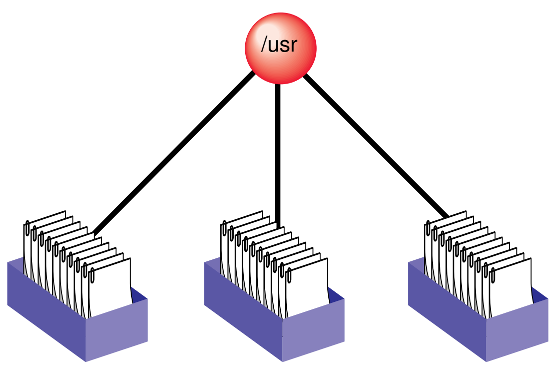

Figure 3-2 Library Locations for the Different ABIs

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
```