Chapter 7

Porting Code to N32 and 64-Bit Silicon Graphics Systems

This section explains the levels of compatibility between the new 32-bit compilation mode (n32), the old 32-bit mode, and 64-bit programs. It also describes the porting procedure to follow and the changes you must make to port your application from old 32-bit mode to n32-bit mode.

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.
- "Porting Code to 64-Bit Silicon Graphics Systems," which describes data types, predefined types, typedefs, and use of large files on xFS.

This chapter 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 (for a list of n32 features, see Chapter 1).

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 7-1 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.

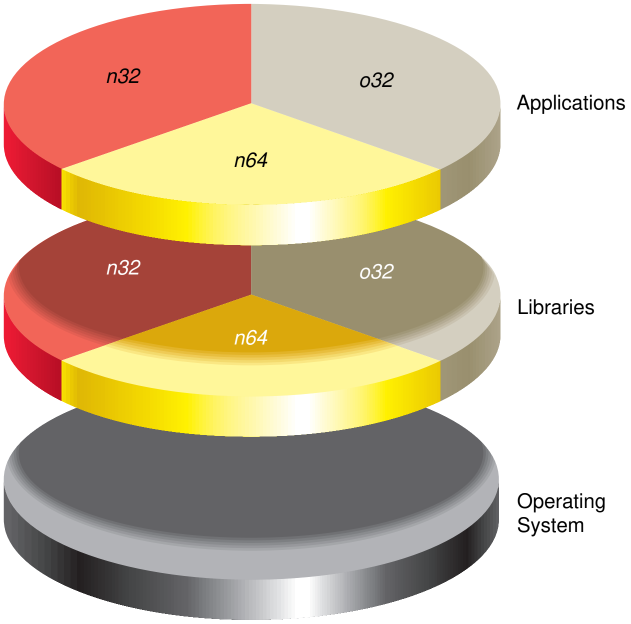

Figure 7-1 Application Support Under Different ABIs

Figure 7-2 illustrates the library locations for different ABIs.

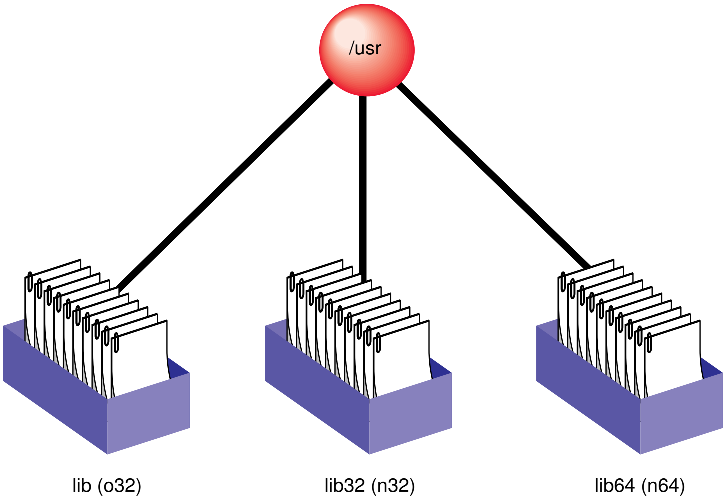

Figure 7-2 Library Locations for Different ABIs

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. You can also find additional information about n32 in the *MIPSpro N32 ABI Handbook*.

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 (*ragnarok*) 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.

Source Code Changes

Since no differences exist in the sizes of fundamental types between the old 32-bit mode 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 Chapter 3 of the *MIPSpro N32 ABI Handbook* in the section titled “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*.

Porting Code to 64-Bit Silicon Graphics Systems

This section covers porting code to 64-bit Silicon Graphics systems, including:

- "Using Data Types"
- "Using Predefined Types"
- "Using Typedefs"
- "Using Large Files With XFS"

You can find additional information about porting to 64-bit Silicon Graphics systems in the *MIPSpro Application Porting and Transition Guide*.

Using Data Types

Data types and sizes are listed in Table 7-1.

Table 7-1 Data Types and Sizes

Data Type	**32 Bit**	**64 Bit**
char	8	8
short	16	16
int	32	32
long	32	64
long long	64	64
pointer	32	64
float	32	32
double	64	64
long double*	64	128
void*	32	64

Note that in 64-bit mode, types **long** and **int** have different sizes and ranges; a **long** always has the same size as a **pointer**. A **pointer** (or address) has 64-bit representation in 64-bit mode and 32-bit representation in 32-bit mode. An **int** has a smaller range than a **pointer** in 64-bit mode.

Characteristics of integral types and floating point types are defined in the standard files *limits.h* and *float.h*.

Using Predefined Types

The *cc*, *CC*, and *as* compiler drivers produce predefined macros listed in Table 7-2. These macros are used in *sys/asm.h*, *sys/regdef.h*, and *sys/fpregdef.h*.

Table 7-2 Predefined Macros

32-Bit Executables	64-Bit Executables
–D_MIPS_FPSET=16	–D_MIPS_FPSET=32
–D_MIPS_ISA=_MIPS_ISA_MIPS1	–D_MIPS_ISA=_MIPS_ISA_MIPS3
–D_MIPS_SIM=_MIPS_SIM_ABI32	–D_MIPS_SIM=_MIPS_SIM_ABI64
–D_MIPS_SZINT=32	–D_MIPS_SZINT=32
–D_MIPS_SZLONG=32	–D_MIPS_SZLONG=64
–D_MIPS_SZPTR=32	–D_MIPS_SZPTR=64

_MIPS_FPSET describes the number of floating point registers. The 64-bit compilation mode makes use of the extended floating point registers available on the R4000 and R8000 processors.

MIPS_ISA determines the MIPS Instruction Set Architecture. MIPS_ISA_MIPS1 and MIPS_ISA_MIPS3 are the defaults for 32 bits and 64 bits, respectively. For example:

```
/* Define a parameter for the integer register size: */
#if (_MIPS_ISA == _MIPS_ISA_MIPS1 || _MIPS_ISA == _MIPS_ISA_MIPS2)
#define SZREG          4
#else
#define SZREG          8
#endif
```

MIPS_SIM determines the MIPS Subprogram Interface Model, which describes the subroutine linkage convention and register naming/usage convention.

_MIPS_SZINT, _MIPS_SZLONG, and _MIPS_SZPTR define the size of types **int**, **long**, and **pointer**, respectively.

The 64-bit MIPSpro compiler drivers generate 64-bit **pointer**s and **long**s and 32-bit **int**s. Therefore, assembler code that uses either **pointer** or **long** types must be converted to use

double-word instructions for MIPS III code(**–64**), and must continue to use **word** instructions for MIPS I and MIPS II code (**–32**).

Also, new subroutine linkage conventions and register naming conventions exist. The compiler predefined macro _MIPS_SIM enables macros in *sys/asm.h* and *sys/regdef.h*.

Eight argument registers exist: **$4** through **$11**. Four additional argument registers replace the **temp** registers in *sys/regdef.h*. These **temp** registers are not lost, however, as the argument registers can serve also as scratch registers, with certain constraints.

In the _MIPS_SIM_ABI64 model, registers **t4** through **t7** are not available, so any code using these registers does not compile. Similarly, registers **a4** through **a7** are not available under the _MIPS_SIM_ABI32 model.

If you are converting assembler code, the new registers **ta0**, **ta1**, **ta2**, and **ta3** are available under both _MIPS_SIM models. These alias with registers **t4** through **t7** in 32-bit mode, and with registers **a4** through **a7** in 64-bit mode.

Note that the caller no longer has to reserve space for a called function in which to store its arguments. The called routine allocates space for storing its arguments on its own stack, if desired. The NARGSAVE macro in *sys/asm.h* facilitates this.

Using Typedefs

This section describes **typedef**s that you can use to write portable code for a range of target environments, including 32- and 64-bit workstations as well as 16- and 32-bit PCs. These **typedef**s are enabled by compiler-predefined macros (listed in Table 7-2), and are in the file *inttypes.h*. (This discussion applies to C, although the same macros are predefined by the C++ compiler.)

Portability problems exist because an **int** (32 bits) is no longer the same size as a **pointer** (64 bits) and a **long** (64 bits) in 64-bit programs. **Typedef**s free you from having to know the underlying compilation model or worry about type sizes. In the future, if that model changes, the code should still work.

Typically, you want source code that you can compile either in 32- or 64-bit mode. (In this discussion, 32-bit mode means **–mips1** or **–mips2**; 64-bit mode means **–mips3** or **–mips4**.)

The following **typedef**s are defined in *inttypes.h*:

```
typedef signed char int8_t;
typedef unsigned char uint8_t;
typedef signed short int16_t;
typedef unsigned short uint16_t;
typedef signed int int32_t;
typedef unsigned int uint32_t;
typedef signed long long int int64_t;
typedef unsigned long long int uint64_t;
typedef signed long long int intmax_t;
typedef unsigned long long int uintmax_t;
typedef signed long int intptr_t;
typedef unsigned long int uintptr_t;
```

intmax_t and **uintmax_t** are guaranteed to be the largest integer type supported by this implementation. Use them in code that must be able to deal with any integer value.

intptr_t and **uintptr_t** are guaranteed to be exactly the size of a **pointer**.

Using Large Files With XFS

An application may create or encounter files greater than 2 gigabytes with XFS. If a program is doing sequential I/O and does not maintain internal byte counters, files greater than 2 Gb won't encounter problems.

However, if an application uses internal byte counters, then modifications are required.

Table 7-3 lists potential problems and modifications required to enable files greater than 2 Gb to run on XFS.

Table 7-3 Modifications for Applications on XFS

Application	**Modification**
Uses an internal byte counter while reading	Change to type **long long**
Uses certain system calls such as **lseek()** and **stat()** that use 32-bit off_t	Use **lseek64()**, **stat64()**, and so forth
Relies on internal features of EFS (such as reads the raw disk)	Rewrite the application (so it doesn't read the raw disk)

For more information about XFS, see *Getting Started With XFS Filesystems*.