Chapter 6

Coding for 64-Bit Programs

This chapter provides information about ways to write/update your code so that you can take advantage of the Silicon Graphics implementation of the IRIX 64-bit operating system. Specifically, this chapter describes:

- "Coding Assumptions to Avoid"
- "Guidelines for Writing Code for 64-Bit Silicon Graphics Platforms"

Coding Assumptions to Avoid

Most porting problems come from assumptions, implicit or explicit, about either absolute or relative sizes of the **int**, **long int**, or **pointer** types in code.

To avoid porting problems, examine code that assumes:

- "sizeof(int) == sizeof(void *)"
- "sizeof(int) == sizeof(long)"
- "sizeof(long) == 4"
- "sizeof(void *) == 4"
- "Implicitly Declared Functions"
- "Constants With the High-Order Bit Set"
- "Arithmetic with long Types" (including shifts involving mixed types and code that may overflow 32 bits)

sizeof(int) == sizeof(void *)

An assumption may arise from casting pointers to **int**s to do arithmetic, from unions that implicitly identify **int**s and **pointer**s, or from passing **pointer**s as actual arguments to

functions where the corresponding formal arguments are declared as **int**s. Any of these practices may result in inadvertently truncating the high-order part of an address.

The compilers generally detect the first case and provide warnings. Also given ANSI C function prototypes, the compilers generally detect the last case. No diagnostic messages are provided for unions that implicitly identify **int**s and **pointer**s.

You can declare an integer variable that is required to be the size of a pointer with the type **ptrdiff_t** in the standard header file *stddef.h*, or with the types **__psint_t** and **__psunsigned_t** in the header file *inttypes.h*.

Also note that a cast of an **int** to a **pointer** may result in sign-extension, if the sign bit of the **int** is set when a **–64** compilation occurs.

sizeof(int) == sizeof(long)

Data that fits in an **int** or **long** on 32-bit systems will fit in an **int** on 64-bit systems. Expansion, in this case, has no visible effect. Problems may occur, however, where an unsigned **int** actual parameter is passed to a **long** (signed or unsigned) formal parameter without benefit of an ANSI prototype. In this case, the unsigned value is implicitly sign-extended in the register, and therefore is misinterpreted in the callee if the **sign** bit was set.

sizeof(long) == 4

A problem may occur in cases where long **int**s are used to map fields in data structures defined externally to be 32 bits, or where **union**s attempt to identify a **long** with four **char**s.

sizeof(void *) == 4

Problems with this code are similar to those encountered with `sizeof(long)==4`. However, mappings to external data structures are seldom a problem, since the external definition also assumes 64-bit pointers.

Implicitly Declared Functions

It is always risky to call a function without an explicit declaration in scope. Furthermore, be sure to declare with a compatible prototype any function defined with a prototype. Problems arise when mixing prototype and nonprototype declarations for the same function. For example, suppose you call a function (defined with a prototype to take a variable number of arguments) in a scope without a prototyped declaration. You may get unexpected results if a floating point argument is passed to it. This is a typical problem with calls to **printf** and after **stdio.h** routines. Therefore, always include **stdio.h** in any context where you use **stdio.h** facilities.

Constants With the High-Order Bit Set

A change in type sizes may yield some problems related to constants. Be careful about using constants with the high-order (**sign**) bit set. For instance, the hex constant **0xffffffff** yields different results in the expression:

```
long x;
... ( (long) ( x + 0xffffffff ) ) ...
```

In both modes, the constant is interpreted as a 32-bit unsigned **int**, with value 4,294,967,295. In 32-bit mode, the addition results in a 32-bit unsigned **long**, which is cast to type **long** and has value x-1 because of the truncation to 32 bits. In 64-bit mode, the addition results in a 64-bit **long** with value x+4,294,967,295, and the cast is redundant.

Arithmetic with `long` Types

Code that does arithmetic (including shifting), and code that may overflow 32 bits and assumes particular treatment of the overflow (for example, truncation), can exhibit different behavior, depending on the mix of types involved (including signedness).

Similarly, implicit casting in expressions that mix **int** and **long** values may produce unexpected results due to **sign/zero** extension. An **int** constant is sign- or zero-extended when it occurs in an expression with **long** values.

Solving Porting Problems

Once you identify porting problems, solve them by:

- changing the relevant declaration to one that has the desired characteristics in both target environments
- adding explicit type casts to force the correct conversions
- using function prototypes or using type suffixes (such as *l* or *u*) on constants to force the correct type

Guidelines for Writing Code for 64-Bit Silicon Graphics Platforms

The key to revising existing code and writing new code that is compatible with all of the major C data models is to avoid the assumptions described above in "Coding Assumptions to Avoid." Since all of the assumptions described sometimes represent legitimate attributes of data objects, you need to tailor declarations to the target machines' data models.

The following guidelines help you to produce portable code. Use these guidelines when you are developing new code or as you identify portability problems in existing code.

1. In a header file that you include in each of the program's source files, define (**typedef**) a type for each of the following functions:
   - For each specific integer data size required, that is, where exactly the same number of bits is required on each target, define a signed and unsigned type. For example:

     ```
     typedef signed char int8_t
     typedef unsigned char uint8_t
     ...
     typedef unsigned long long uint64_t
     ```

   - If you require a large scaling integer type, that is, one that is as large as possible while remaining efficiently supported by the target, define another pair of types. For example:

     ```
     typedef signed long intscaled_t
     typedef unsigned long uintscaled_t
     ```

   - If you require integer types of at least a particular size, but chosen for maximally efficient implementation on the target, define another set of types,

similar to the first but defined as larger standard types where appropriate for efficiency. The typedefs referred to above exist in the file *inttypes.h* (see "Using Typedefs").

After you construct the above header file, use the new **typedef** types instead of the standard C type names. You may need a distinct copy of this header file (or conditional code) for each target platform supported.

If you provide libraries or interfaces to be used by others, be careful to use these types (or similar application-specific types) chosen to match the specific requirements of the interface. Also, carefully choose the actual names used to avoid namespace conflicts with other libraries. Thus, your clients should be able to use a single set of header files on all targets. However, you will always need to provide distinct libraries (binaries) for the 32-bit compatibility mode and the 64-bit native mode on 64-bit Silicon Graphics platforms, although the sources can be identical.

2. Be sure to specify constants with an appropriate type specifier so that they will have the size required by the context with the values expected. Bit masks can be particularly troublesome in this regard: avoid using constants for negative values. For example, 0xffffffff may be equivalent to a -1 on 32-bit systems, but may be interpreted as 4,294,967,295 (signed or unsigned, depending on the mode and context) on most 64-bit systems. The *inttypes.h* header file provides *cpp* macros to facilitate this conversion. Defining constants that are sensitive to type sizes in a central header file may help in modifying them when a new port is done.

3. Where *printf/scanf* are used for objects whose types are **typedef**ed differently among the targets you must support, you may need to define constant format strings for each of the types defined in step 1. For example:

```
#define _fmt32 "%d"
#define _fmt32u "%u"
#define _fmt64 "%lld"
#define _fmt64u "%llu"
```

The *inttypes.h* header file also defines *printf/scanf* format extensions to standardize these practices. They are implemented by the first release of the 64-bit compilers and related tools.