64-bit C Porting Guide

MIPS Technologies / Silicon Graphics Computer Systems

October 13, 1993

WARNING: This guide is speculative and highly subject to change. It should be viewed only as an indication of possibilities to assist in planning a 64-bit transition.

Section 1 Overview of the Problem

Porting programs written in C to SGI's 64-bit MIPS architecture platforms, using the native 64-bit C compilers and related tools, should be straightforward. However, depending on assumptions made in writing the original C code, it may require some changes. This guide is intended to assist programmers involved in this process, addressing both the porting of existing code and guidelines for writing code which will be portable later.

The C integer data types have historically been sized based on matching desired functionality with the target architecture's ability to efficiently implement integers of various sizes. In Table 1 we identify five type models which are currently widespread, or are likely to become widespread as 64-bit architectures proliferate.

The first column, the 32BIT model, is the model supported on SGI's current 32-bit systems and on most other Unix workstations. It will also be supported for compatibility on SGI's 64-bit systems. The second, third, and fourth are models which will probably be supported on various 64-bit computer systems. They all have 64-bit pointers in common, and differ in the sizes of their **int** and **long int** data types. The SGI 64-bit platforms will support the LP64 model for *native* 64-bit programs. Finally, the PC

C type	32BIT	LP64	LLP64	ILP64	PC
char	8	8	8	8	8
short int	16	16	16	16	16
int	32	32	32	64	16
long int	32	64	32	64	32
long long int	64	64	64	64	
pointer	32	64	64	64	16/32

Table 1: C Data Type Models

model reflects many compilers available for IBM PC compatibles, based on Intel's 80x86 architectures.

In the sections below, we discuss what problems to expect in porting existing C code to the LP64 native model, and suggest approaches for avoiding porting problems in new code.

Section 2 Porting to the LP64 Model

For code which currently runs on SGI 32-bit platforms, porting to the LP64 model will normally be straightforward. (It may also be unnecessary. Unless the program requires a 64-bit address space, 64-bit data, or other native-only functionality, it may still be run as a 32-bit program.)

Porting will require, at a minimum, recompiling and relinking the program. You will need to specify a 64-bit target if you are doing this on a 32-bit workstation; on a 64-bit workstation this may be the default (and you would have to request the 32BIT compatibility model). In some cases, the differences between the models will imply changes in SGI-provided system header files and/or libraries; in such cases, your selection of the 32BIT or LP64 model will automatically select the correct version.

Within your code, most porting problems will arise from assumptions, implicit or explicit, about either absolute or relative sizes of the **int**, **long int**, or pointer types. The most common are likely to be:

- sizeof(int) == sizeof(void *)

This assumption may arise from casting pointers to ints to do arithmetic, from unions which implicitly identify ints and pointers, or from passing pointers as actual arguments to functions where the corresponding formal arguments are declared as ints. Any of these practices may result in inadvertantly truncating the high-order part of an address. Generally, an integer variable required to be the size of a pointer may be declared with the type **ptrdiff_t** in the standard header file **stddef.h**.

Use the SGI compilers will normally catch the first case (providing warnings). They will also, given ANSI C function prototypes, generally catch the last case.

- sizeof(int) == sizeof(long)

  This assumption arises in contexts similar to the previous one. It will not usually become visible, since data which fit in an int or long on 32-bit systems will usually still fit in an int on 64-bit systems, so that the truncation has no visible effect. It may, however, cause problems in the case where an unsigned int actual parameter is passed to a (signed or unsigned) long formal parameter without benefit of an ANSI prototype, since the unsigned value is implicitly sign-extended in the register and will therefore be misinterpreted in the callee if the "sign" bit was set.

- sizeof(long) == 4

  This assumption may arise in cases where long ints have been used to map fields in data structures defined externally to be 32 bits, or where unions attempt to identify a long with four chars, for example.

- sizeof(void *) == 4

  This assumption is analogous to the previous one. But mappings to external data structures should seldom be a problem, since the external definition should also assume 64-bit pointers in the LP64 model.

- constants

  The change in type sizes may yield some surprises related to constants. You should be especially careful about using constants with the high-order (sign) bit set. For instance, the hex constant **0xffffffff** will yield different results in the expression:

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

In both models, the constant is interpreted as a 32-bit unsigned int, with value **4,294,967,295**. In the 32BIT model, the addition result is a 32-bit unsigned long, which is cast to type long and has value x-1 because of the truncation to 32-bits. In the LP64 model, the addition result is a 64-bit long with value **x+4,294,967,295**, and the cast is redundant.

- arithmetic assumptions

  Related to some of the above cases, code which does arithmetic (including shifting) which may overflow 32 bits, and assumes particular treatment of the overflow (e.g. truncation), may exhibit different behavior in the LP64 model, depending on the mix of types involved (including signedness).

  Similarly, implicit casting in expressions which mix int and long values may behave unexpectedly due to sign/zero extension. In particular, remember that integer constants will be sign or zero extended when they occur in expressions with long values.

Once identified, each of these problems is easy to solve. Change the relevant declaration to one which has the desired characteristics in both target environments, add explicit type casts to force the correct conversions, use function prototypes, or use type suffixes (e.g. ‘l’ or ‘u’) on constants to force the correct type.

If you are also interested in the other 64-bit target environments, additional (but similar) problems may arise:

- sizeof(long) == sizeof(void *): LLP64

  Analogous to the sizeof(int) == sizeof(void *) case in LP64.

- sizeof(int) == 4: ILP64

  Analogous to the sizeof(long) == 4 case in LP64. Observe that if a 32-bit type is required, e.g. to map external data, use of an implementation-defined type (perhaps named __int32) will be required. To avoid distributed differences between the ILP64 code and code for targets without __int32 builtin types, we recommend using a typedef in a header file, so that only the header file needs to be changed for different targets.

- long is the largest efficient integer on the target: LLP64

  This assumption may no longer be valid. Whether it causes noticible problems will depend on whether the values encountered actually grow past 32 bits in the application.

  Fixing such instances in a fashion compatible between the various type models will require use of long long int declarations for the 64-bit target while retaining long int (or int) declarations for 32-bit targets, i.e. conditional code. Using an appropriate typedef can restrict the conditional code to a single header file.

As for the LP64 cases, fixing these porting problems individually will usually be easy. However, since the individual solutions may not be consistent between the various 64-bit models, we recommend that customers who must support other vendors' platforms move towards the more general recommendations in the next section.

Section 3 Writing Code Portable to 64-bit Platforms

The key to writing new code which is compatible with all of the major C data models described is just to avoid those problems described in the previous section. Since all of the assumptions described sometimes represent legitimate attributes of data objects, this requires some tailoring of declarations to the target machines' data models.

We suggest following the following guidelines to produce code without the more common portability problems. They can be followed from the beginning in developing new code, or adopted incrementally as portability problems are identified.

1. In a header file which can be included 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, i.e. where exactly the same number of bits is required on each target, define a signed and unsigned type[1], e.g.:

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

1. N.B.: A group of vendors, including all major UNIX workstation vendors, has agreed to a proposal to provide a standard header file named ***inttypes.h*** supporting new typedefs of this form, which may be appropriate underlying types for these typedefs. Using them directly in your code could conceivably result in portability problems for systems or compilers which do not support those types, or later if ANSI specifies a builtin facility for the same purpose.

- If you require a large *scaling* integer type, i.e. one which is as large as possible while remaining efficiently supported by the target, define another pair of types, e.g.:

    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.

Having constructed the above header file, use the new typedef'ed types instead of the standard C type names. You will need (potentially) a distinct copy of this header file (or conditional code) for each target platform supported.

As a special case of this, if you are providing libraries or interfaces to be used by others, be particularly careful to use these types (or similar application-specific types) chosen to match the specific requirements of the interface. Also in such cases, you will want to choose the actual names used to avoid namespace conflicts with other libraries doing the same thing. If this is done carefully, your clients should be able to use a single set of header files on all targets. However, you will generally need to provide distinct libraries (binaries) for the 32BIT compatibility model and the LP64 native model on 64-bit SGI platforms, though the sources may be identical.

2. Be careful that constants are specified with appropriate type specifiers so that they will extend to the size required by the context with the values that you require. 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 it will be interpreted as **4,294,967,295** (signed or unsigned, depending on the model and context) on most 64-bit systems. The ***inttypes.h*** header file provides ***cpp*** macros to facilitate this conversion.

   Defining constants which 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), e.g.:

```
#define _fmt32 "%d"
#define _fmt32u "%u"
#define _fmt64 "%ld"
#define _fmt64u "%lu"
```

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