



Extend Your Application's Reach from 32-bit to 64-bit Environments – Part 2: Porting Guidelines

By Chandrashekhar M, Director & CTO, S7 Software Solutions

In Part 1 of this article, we provided some background on 32- to 64-bit porting and described some of the key considerations for porting. In this article, we'll offer some porting guidelines. In Part 3, we'll look at a couple applications ported from 32- to 64-bit environments.

Guidelines for Converting to LP64

The following guidelines are a consolidation of our experience and the best practices of developers, plus suggestions by processor manufacturers.

Guideline: Pay close attention to compiler warnings.

Most of the recent compilers generate warnings whenever there is a type mismatch. It is always a good idea to turn on these warnings. The flag varies from compiler to compiler and from platform to platform; see the compiler's technical documentation.

Guideline: Do not assume int and Pointers are the same size

Since ints and pointers are the same size in the ILP32 environment, a lot of code relies on this assumption. Pointers are often cast to int or unsigned int for address arithmetic. Instead, pointers could be cast to long because long and pointers are the same size in both ILP32 and LP64 worlds. Rather than explicitly using unsigned long, use `uintptr_t`, because it expresses the intent more closely and makes the code more portable, insulating it against future changes.

Code Example

```
char *p;  
p = (char *) ((int)p & PAGEOFFSET);  
%  
warning: conversion of pointer loses bits
```

Suggested use:

```
char *p;  
p = (char *) ((uintptr_t)p & PAGEOFFSET);
```

Guideline: Do not assume int and long are the same size



Because ints and longs were never really distinguished in ILP32, a lot of existing code uses them indiscriminately while implicitly or explicitly assuming that they are interchangeable. Any code that makes this assumption must be changed to work for both ILP32 and LP64. While an int and a long are both 32 bits in the ILP32 data model, in the LP64 data model, a long is 64 bits.

Code Example

```
int waiting;
long w_io;
long w_swap;
...
waiting = w_io + w_swap;
%
```

warning: assignment of 64-bit integer to 32-bit integer

Guideline: Sign extension

Sign extension is a common problem when converting to 64–bits. It is hard to detect before the problem actually occurs because lint(1) does not warn you about it. Furthermore, the type conversion and promotion rules are somewhat obscure. To fix sign extension problems, you must use explicit casting to achieve the intended results. To understand why sign extension occurs, it helps to understand the conversion rules for ANSI C.

Two rules seem to cause the most sign extension problems between 32-bit and 64-bit integral values:

- **Integral promotion.** A char, short, enumerated type, or bit-field, whether signed or unsigned, can be used in any expression that calls for an int. If an int can hold all possible values of the original type, the value is converted to an int. Otherwise; it is converted to an unsigned int.
- **Conversion between signed and unsigned integers.** When a negative signed integer is promoted to an unsigned integer of the same or larger type, it is first promoted to the signed equivalent of the larger type, and then converted to the unsigned value.

For a more detailed discussion of the conversion rules, refer to the ANSI C standard. Also included in this standard are useful rules for ordinary arithmetic conversions and integer constants. When compiled as a 64-bit program, the addr variable in the following example becomes sign-extended, even though both addr and a.base are unsigned types.

Code Example



```
% cat test.c
struct foo {
  unsigned int base:19, rehash:13;
};
main(int argc, char *argv[])
{
  struct foo a;
  unsigned long addr;
  a.base = 0x40000;
  addr = a.base << 13; /* Sign extension here! */
  printf("addr 0x%lx\n", addr);
  addr = (unsigned int)(a.base << 13); /* No sign extension here! */
  printf("addr 0x%lx\n", addr);
}
```

This sign extension occurs because the conversion rules are applied as follows:

1. `a.base` is converted from an unsigned int to an int because of the integral promotion rule. Thus, the expression `a.base << 13` is of type int, but no sign extension has yet occurred.
2. The expression `a.base << 13` is of type int, but it is converted to a long and then to an unsigned long before being assigned to `addr`, because of signed and unsigned integer promotion rules. The sign extension occurs when it is converted from an int to a long.

```
% cc -o test64 -xarch=v9 test.c
% ./test64
addr 0xffffffff80000000
addr 0x80000000
%
```

When this same example is compiled as a 32-bit program, it does not display any sign extension:

```
% cc -o test32 test.c
% ./test32
addr 0x80000000
addr 0x80000000
%
```



Guideline: Use pointer arithmetic instead of address arithmetic

In general, using pointer arithmetic works better than address arithmetic because pointer arithmetic is independent of the data model, whereas address arithmetic might not be. It usually leads to simpler code as well.

Code Example

```
int *end;
int *p;
p = malloc(4 * NUM_ELEMENTS);
end = (int *)((unsigned int)p + 4 * NUM_ELEMENTS);
%
```

warning: conversion of pointer loses bits

Suggested use:

```
int *end;
int *p;
p = malloc(sizeof (*p) * NUM_ELEMENTS);
end = p + NUM_ELEMENTS;
```

Guideline: Repack structures

Internal data structures in applications should be checked for holes. Since any long or pointer fields will grow to 64 bits for LP64, you can use extra padding between fields in the structure to meet alignment requirements. In the 64-bit environment on Intel Itanium® 2 processor-based platforms, all types of structures are aligned to at least the size of the largest quantity within them. A simple rule for repacking the structure is to move the long and pointer fields to the beginning of the structure.

Code Example

```
struct bar {
int i;
long j;
int k;
char *p;
}; /* sizeof (struct bar) = 32 */
```

Suggested use:



```
struct bar {  
char *p;  
long j;  
int i;  
int k;  
}; /* sizeof (struct bar) = 24 */
```

Guideline: Check unions

Be sure to check unions, because their fields might have changed sizes between ILP32 and LP64.

Code Example

```
typedef union {  
double _d;  
long _l[2];  
} llx_t;
```

Suggested use:

```
typedef union {  
double _d;  
int _l[2];  
} llx_t;
```

Guideline: Specify type of constants

Lack of precision can cause loss of data in some constant expressions. These types of problems are very hard to find. Be explicit about specifying the type(s) in your constant expressions. Add some combination of {u,U,l,L} to the end of each integer constant to specify its type. You might also use casts to specify the type of a constant expression.

Code Example

```
int i = 32;  
long j = 1 << i; /* j will get 0 because RHS is integer */  
/* expression */
```

Suggested use:

```
int i = 32;  
long j = 1L << i;
```



Guideline: Beware of implicit declaration

The C compiler from Sun WorkShop assumes a type int for any function or variable that is used in a module and not defined or declared externally. Any longs and pointers used in this way are truncated by the compiler's implicit int declaration. The appropriate extern declaration for the function or variable should be placed in a header and not in the C module. Any C module that uses the function or variable should then include this header. If this is a function or a variable defined by the system headers, the proper header should still be included in the code.

Code Example

```
%
int
main(int argc, char *argv[])
{
char *s = malloc(10);

strcpy(s, "TEST");
printf("%s\n", s);
free(s);
return (0);
}
```

Compile it as: cc -g +DD64 test.c

Suggested use:

```
#include <stdio.h>
// #include <stdlib.h> /* uncomment this line to resolve the core dump on HP 11.23 */

struct bar {
int j;
};

struct bar *foo() {

struct bar *p = (struct bar *)malloc(sizeof(struct bar));
p->j = 1000;
return p;
}
```



```
}
```

```
int main() {  
    struct bar *x = foo();  
    printf("%d\n", x->j);  
}
```

Guideline: sizeof() is an unsigned long

In LP64, sizeof() has the effective type of an unsigned long. Occasionally sizeof() is passed to a function expecting an argument of type int, or is assigned or cast to an int. In some cases, this truncation might cause loss of data.

Guideline: Use casts to show your intentions

Relational expressions can be tricky because of conversion rules. You should be very explicit about how you want the expression to be evaluated by adding casts wherever necessary.

Guideline: Check format string conversion operation

The format strings for printf(3S), sprintf(3S), scanf(3S), and sscanf(3S) might need to be changed for long or pointer arguments. For pointer arguments, the conversion operation given in the format string should be %p to work in both the 32-bit and 64-bit environments.

Code Example

```
char *buf;  
struct dev_info *devi;  
...  
(void) sprintf(buf, "di%x", (void *)devi);  
%  
warning: function argument (number) type inconsistent with format  
sprintf (arg 3) void *: (format) int
```

Suggested use:

```
char *buf;  
struct dev_info *devi;  
...
```



Guideline: Check for side effects of changes

One problem to be aware of is that a type change in one area might result in an unexpected 64-bit conversion in another area. For example, in the case of a function that previously returned an `int` and now returns an `ssize_t`, all the callers need to be checked.

Guideline: Check whether literal uses of long still make sense

Because a `long` is 32 bits in the ILP32 model and 64 bits in the LP64 model, there might be cases where what was previously defined as a `long` is neither appropriate nor necessary. In this case, it might be possible to use a more portable derived type. Related to this, a number of derived types might have changed under the LP64 data model for the reason stated above. For example, `pid_t` remains a `long` in the 32-bit environment, but under the 64-bit environment a `pid_t` is an `int`.

Summary

Porting 32-bit code to a 64-bit environment is not a trivial undertaking, especially if the code was written without foresight of portability. The guidelines offered in this article should help you find and correct the more troublesome code usages. In Part 3, we will look at some examples of code ported to a 64-bit environment.