



Extend Your Application's Reach from 32-bit to 64-bit Environments – Part 1: Porting Considerations

By Chandrashekhar M, Director & CTO, S7 Software solutions

Many existing 32-bit applications are outgrowing their 32-bit environment limitations, acquiring 64-bitness through porting efforts by their developers. Still other software marketers are realizing the value of porting to 64-bit systems and compiling for a more powerful processor microarchitecture. In a series of three articles, we'll provide background on 32- to 64-bit porting and describe some of the key considerations for porting; the things to watch for during the process, and then look at a couple applications ported from 32- to 64-bit environments. In this first article, we introduce the major differences of the environments and the possible impacts on porting.

Is 64-bit right for you?

Today, many 32-bit applications have opportunities beyond their current market. More often than not, these applications simply need more addressable memory than the 4GB limitation of 32 bits. But others need the processing power offered by processor microarchitectures designed for data-intensive 64-bit computing. These true 64-bit processors provide greater headroom with more registers, larger page file size, and other features for greater scalability and performance.

Database applications — particularly those that perform data mining — as well as Web caches and Web search engines, components of CAD/CAE simulation and modeling tools, and scientific computing are likely candidates for porting from 32-bit to 64-bit environments through a process called “code cleaning.”

Do you need to go through an involved porting process to reach the 64-bit market? Not necessarily. Most 32-bit code can run in a 64-bit environment, as long as the underlying operating environment can support your 32-bit code. Conversion is necessary only if one or more of the following is true for your program in the UNIX/Linux environment:

- Needs more than 4 GB of virtual address space
- Reads and interprets kernel memory using libkvm, /dev/mem, or/dev/kmem
- Uses /proc to debug 64-bit processes
- Uses a library that has only a 64-bit version
- Needs full 64-bit registers to do efficient 64-bit arithmetic

Figure 1 illustrates the two environments supported under UNIX. The 32-bit system supports only 32-bit applications and libraries. The 64-bit system supports the same 32-bit applications and libraries, but also supports 64-bit libraries and applications. To run



your existing 32-bit code, you need the 32-bit libraries your code depends on, plus 64-bit versions of any drivers the code must have.

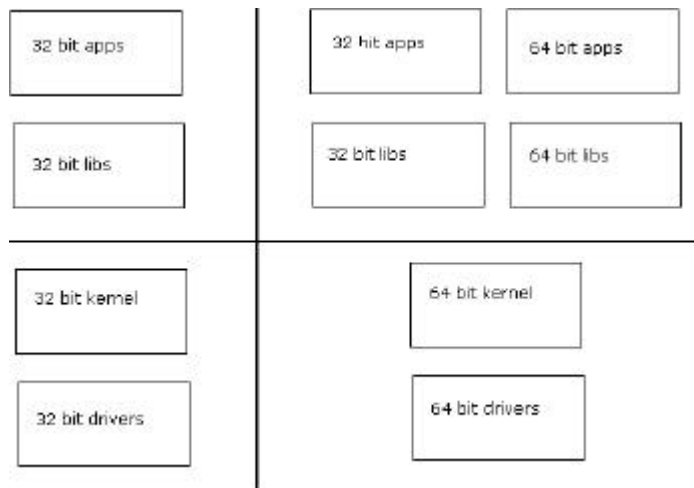


Figure 1. 32-bit and 64-bit system stacks

Major 64-bit Environment Features

The 64-bit environment supports:

- **Large virtual address space** – address up to 18 exabytes, about 4 billion times the current maximum of 32-bit addressing.
- **Large files** – If an application requires only support for large files, then it can remain 32-bit and use the Large Files interface. However, if portability is not a primary concern, it might be useful to convert the application to a 64-bit program to take full advantage of 64-bit capabilities with a coherent set of interfaces.
- **64-bit arithmetic** – This allows an application to take full advantage of the capabilities of the 64-bit CPU hardware.
- **Removal of certain system limitations** – The 64-bit system interfaces are inherently more capable than some of the 32-bit equivalents. For example, application programmers concerned about year 2038 problems (when 32-bit time_t runs out of time) can use the 64-bit time_t. While 2038 seems like a long way off, applications that do computations concerning future events, such as mortgages, might require the expanded time capability.

Porting Considerations

The major differences between the 32-bit and the 64-bit development environments are that 32-bit applications are based on the ILP32 data model, where ints, longs, and pointers are 32 bits, while 64-bit applications are based on the LP64 model, where longs and pointers are 64 bits and the other fundamental types are the same as in ILP32 (Table 1).



Table 1. Differences in data types for ILP32 and LP64

C data type	ILP32	LP64
Char	8	Unchanged
Short	16	Unchanged
Int	32	Unchanged
Long	32	64
Long long	64	Unchanged
Pointer	32	64
Enum	32	Unchanged
Float	32	Unchanged
Double	64	Unchanged
Long double	128	Unchanged

Sample Program

The following sample program illustrates the effect of LP64 and ILP32 data models. The same program can be compiled as either a 32-bit program or a 64-bit program.

```
% cat foo.c
#include <stdio.h>
int
main(int argc, char *argv[])
{
(void) printf("char is \t\t%lu bytes\n", sizeof (char));
(void) printf("short is \t\t%lu bytes\n", sizeof (short));
(void) printf("int is \t\t%lu bytes\n", sizeof (int));
(void) printf("long is \t\t%lu bytes\n", sizeof (long));
(void) printf("long long is \t\t%lu bytes\n", sizeof (long long));
(void) printf("pointer is \t\t%lu bytes\n", sizeof (void *));
return (0);
}
```

32-bit Compilation

```
% cc -O -o foo32 foo.c
% foo32
```

```
char is 1 bytes
short is 2 bytes
int is 4 bytes
long is 4 bytes
long long is 8 bytes
pointer is 4 bytes
```



64-bit Compilation

```
% cc -xarch=v9 -O -o foo64 foo.c  
% foo64
```

char is 1 bytes
short is 2 bytes
int is 4 bytes
long is 8 bytes
long long is 8 bytes
pointer is 8 bytes

Key Porting Considerations

It is not unusual for current 32-bit applications to assume that ints, pointers, and longs are the same size. Because the size of longs and pointers change in the LP64 data model, you need to be aware that this change alone can cause many 32-bit to 64-bit conversion problems. In addition, declarations and casts become very important in showing what is intended; how expressions are evaluated can be affected when the types change. The change in data-type sizes influences standard C conversion rules. To adequately show what is intended, you might need to declare the types of constants. Casts might also be needed in expressions to make certain that the expression is evaluated the way you intended. This is particularly true in the case of sign extension, where explicit casting might be essential to show intent.

In general, the migration of 32-bit source code to 64-bit will require the following changes:

- Pointers will become 64-bit. All direct or implied assignment or comparisons between pointers and "int" or "long" values must be examined. All casts to allow the compiler to accept assignment and comparison between pointers and integers must be removed. The type must be changed to a type of variable size (equal to pointer size).
- Longs will become 64-bit. All direct or implied assignment or comparisons between "int" or "long" values must be examined. All casts to allow the compiler to accept assignment and comparison between longs and integers must be examined to ensure validity. Better yet, if the two must remain of different sizes, then implement code that ensures that the 64-bit item does not exceed the maximum value of the 32-bit item.
- Declarations of variables, parameters, or function/method return types that must change to 64-bit size should be altered to use one of the size variant types.



- Functions with no prototype return an integer type in C/C++. Thus, functions returning a pointer must be declared, or the header with its prototype must be included.
- Code must be changed to use the new 64-bit APIs. On both 32- and 64-bit platforms, many APIs use data types the compiler will interpret as 64-bit in conflict with explicitly 32-bit data types in user code.
- Constants, especially hex or binary values are likely to be 32-bit specific. For example, a 32-bit constant 0x80000000 becomes 0x0000000080000000 in 64-bit. Depending on how it is being used, the results can be undesirable.
- Macros depending on 32-bit layout must be adjusted for the 64-bit environment.
- A variety of other issues can arise from sign extension, memory allocation sizes, shift counts, array offsets, and other factors.
- Printf/scanf formats might have to be modified. This is especially true for legacy applications that are printing out reports in a predefined format.

Summary

There are different reasons for porting applications. Porting 32-bit applications to 64-bit environments can expand your market. But porting from a 32- to 64-bit environment is not a trivial task. It requires careful consideration of your current code's use of sized constructs and the impact they can have in the new environment. You need to be aware of these before you engage in any porting effort. In Part 2 of our article, we'll look at specific guidelines that can help you with a successful port of your code.