PUBLISHED IN THE
SEPTEMBER/OCTOBER 1995 ISSUE OF *The X Journal*.

IS YOUR X CODE READY FOR 64-BIT?

Mark J. Kilgard *
Silicon Graphics Inc.
$Revision: 1.8$

May 24, 1995

While Microsoft is busy trying to move the 16-bit DOS-based personal computer world to the 32-bit world of Windows 95 and NT, innovative X workstation vendors are already transitioning their 32-bit product lines to 64-bit architectures. 64-bit computing means these newer machines can operate on data in 64-bit chunks for more efficient processing and permit programs to utilize huge virtual address spaces. And usually, these systems are designed to handle enormous files far in excess of two gigabytes. The move to 64-bit makes workstations more like traditional supercomputers.

While workstation vendors have often claimed near supercomputer performance from their machines, supercomputing is more than fast arithmetic. To be a supercomputer, fast computations must be coupled with the ability to operate on the voluminous quantities of data that supercomputing applications use and generate. 64-bit systems make true supercomputing and hugely data intensive applications feasible on computers that are more like conventional Unix workstations than highly specialized and expensive supercomputers. These more cost-effective 64-bit systems open up new computing possibilities beyond traditional supercomputing applications. Huge-scale databases with sophisticated query engines, real-time video services, and faster conventional workstation applications are all possible with these new 64-bit systems.

The two X workstation vendors that already deliver 64-bit systems are Digital Equipment and Silicon Graphics, Inc. (SGI). Digital's Alpha-based workstations and servers run a 64-bit version of OSF/1, recently renamed Digital Unix. SGI's R8000-based Power Challenge, Power Onyx, and Power Indigo^2 machines run the 64-bit IRIX 6.0.1 operating system. The next release of IRIX will provide 64-bit system support for some configurations using the R4000 family of processors. Other major workstations vendors have plans for 64-bit systems in the future.

Just because a computer has a 64-bit processor does not make it a 64-bit system. There are three fundamental requirements a computer system must meet to be called a 64-bit system. It must have:

- A processor that efficiently supports 64-bit data types including pointers. This implies the primary data paths for the system are (at least) 64-bits wide.
- A compilation system that can produce 64-bit executables.
- An operating system that can run 64-bit programs.

Digital made a wholesale switch to 64-bit systems in their Unix product line when Digital adopted the Alpha processor. This approach avoids much of the baggage of carrying forward 32-bit compatibility.

While SGI has been shipping workstations using the R4000 family of 64-bit processors for over three years, these machines shipped with a 32-bit operating system. Even so, by shipping 64-bit hardware, SGI is in a significantly better position than other vendors who continue to grow their 32-bit installed hardware base. Transitioning SGI's product line to true 64-bit systems is easier when much of SGI's installed base of machines already have 64-bit capable hardware.

The material presented here focuses on how the coming transition to 64-bit in the workstation world affects X programmers coding in C or C++. The transition to 64-bit will be much easier if you are aware of how 64-bit systems affect the X Window System and the X code you write. Even if you have no immediate plans to use 64-bit systems, the code you write today may be ported to future 64-bit systems. If written correctly, the necessary porting effort can be greatly eased.

This article is structured in three sections. The first section explains the options and advantages for using 64-bit systems. In particular, how a 64-bit programming model changes the sizes of fundamental C data types is discussed. Then, the second section explores the kinds of general portability problems introduced by 64-bit data types. The last section details how 64-bit systems

*Mark graduated with B.A. in Computer Science from Rice University and is a Member of the Technical Staff at Silicon Graphics. He can be reached by electronic mail addressed to `mjk@sgi.com`

affect the X Window System and how to avoid 64-bit portability pitfalls specific to X programming interfaces.

1 General 64-bit Considerations

Before migrating any code to a 64-bit system, make sure that the transition is justified. Some 64-bit versions of Unix support both a 64-bit Application Binary Interface (ABI) as well as 32-bit ABI for backward compatibility with existing applications. IRIX 6.0.1 is an example of an operating system that supports both 64-bit and 32-bit applications. Because Digital's Unix broke with the past, all applications *must* be compiled as 64-bit programs.

If you have the option of 32-bit compatibility, it may not be worthwhile to migrate existing code to 64-bit. Converting code to 64-bit makes sense if you plan on using huge files or a huge address space. Converting to 64-bit also makes sense if you can utilize efficient 64-bit integer types or other 64-bit processor features and performance that would be otherwise unavailable.

Keep in mind that there are also downsides to 64-bit programs that result from the increased program memory usage because many basic data types expand from 32-bit to 64-bit quantities. Also, you may need to test and support both a 32-bit and 64-bit version of your code when a single 32-bit version would work as well.

For most existing X applications, unless porting to 64-bit is required, using 32-bit compatibility is an appropriate option. For libraries, the choice of whether to support 64-bit is based on the needs of the library customers. Since a 64-bit application may require various libraries, providing 64-bit library implementations is generally a good idea even if not currently needed.

1.1 The 64-bit Programming Model

Once you have decided that you need a 64-bit version of your code, you need to consider what changes when you compile your code to generate 64-bit executables. Even if you have no immediate plans to port code to 64-bit, you should adopt the coding practices described here to avoid pitfalls in future code.

The most important difference between a 32-bit and a 64-bit system is the sizing of fundamental C data types. The source of most problems with code being recompiled for 64-bit is due to assumptions about data type sizes. So how do the size of C data types change in 64-bit?

There are a couple different answers. So far, the term "64-bit" has been used informally. In general, 64-bit means that pointers are represented as 64-bit values[1] in contrast to the 32-bit pointers on 32-bit systems. While 64-bit systems use 64-bit pointers, the sizes of other C data types are open for interpretation.

The C language specification leaves it to compiler implementors to define the sizes for C data types for a particular target architecture. How many bytes make up a `long`, `int`, or `short` is not defined by the C language specification. The sizes of the C integer types are sized based on the target architecture's ability to efficiently implement integers of the various sizes.

For 64-bit systems, there is more than one reasonable way to size the basic C data types. Table 1 lists some of the possible C data type sizes for 64-bit and 32-bit systems. The **32BIT** and **PC** models are not 64-bit models but are listed for comparison.

The **PC** model is the standard for Intel's x86 architecture. The **32BIT** model is the standard for 32-bit workstations. The nice thing about the **32BIT** model is that the pointer, `int`, and `long` data types all share the same 32-bit representation. If you are unfamiliar with the `long long` data type, it is an invention of some compiler implementors to support a larger than 32-bit integer type without resizing the **32BIT** `long` and without adding any new keywords to the C language. Not all compilers implement `long long`.

The **LP64**, **LLP64**, and **ILP64** models are all 64-bit models. The names are mnemonic for the types that are represented as 64-bit values in each model. For example, **ILP64** indicates that pointer, `int`, and `long` data types are represented as 64-bit values. Each model has advantages and disadvantages for compatibility with existing code, interoperability with 32-bit processes, functionality, and performance.

The choice of which 64-bit data type model to use is generally made by your system or C compiler vendor. Most 64-bit Unix workstation vendors are adopting the **LP64** model, but be aware that other models exist. In particular, Cray uses the **ILP64** model.

For floating point types, the size of the `float` and `double` type generally is 32-bit and 64-bit respectively. This is not a change from the **32BIT** model. Larger than 64-bit floating pointer types are possible though.

Since the existing 64-bit X workstation vendors have adopted the **LP64** model, that model is assumed for the rest of this article though many observations that follow are relevant to the other 64-bit models. The source of most 64-bit **LP64** transition problems is that that the pointer and `long` data types are sized differently than the `int` data type in the **32BIT** model. Existing C code written for the **32BIT** model is liable to break if it is based on assumptions about the `int`, `long`, and pointer types being the same size.

1.2 Advantages of 64-bit

While moving to 64-bit can expose portability problems in your code, coding for 64-bit systems can be to your advantage.

The ability to efficiently perform logical operations on 64-bit values can increase the speed of applications like image processing and circuit simulation where large bit vectors or matrices are manipulated. If these operations are found in the performance critical sections of such applications, minor changes to use 64-bit `long` types can potentially double application performance. Usually, this type of tuning can be done with changes localized to the performance critical code.

[1] While 64-bit pointers are manipulated as 64-bit quantities, in the foreseeable future no machine will this entire range which could address 16 *billion* gigabytes of data.

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

Table 1: C data type models for 32-bit and 64-bit systems.

The efficiency of arithmetic operations on 64-bit integers is potentially more than double when using 64-bit operations (as compared to emulating such operations in a 32-bit environment). As an example, encryption algorithms can benefit from 64-bit arithmetic and logical operations. Software changes may be required to utilize 64-bit operations, but the changes to exploit 64-bit integers are typically localized.

A 32-bit address space can be restrictive for some of today's large data base, engineering, and scientific programs. Typically, 32-bit systems can provide up to two gigabytes of usable main memory to applications, but this may not be enough for data intensive applications. A 64-bit address space may increase data base performance by permitting a huge data cache. A 64-bit engineering or scientific simulation permits problems to be solved that would otherwise be too large for a 32-bit address space.

But be aware that 64-bit programs have secondary costs that may increase their memory and cache requirements compared to a 32-bit version. Because pointers are represented as 64-bit values, the memory to store pointers will double.

Migration of supercomputer applications to Unix workstations is a case where porting is easier for 64-bit systems than 32-bit systems. Supercomputer applications are already adapted to 64-bit architectures. Running a supercomputer application on a 64-bit workstation may be less expensive than running it on a supercomputer and may provide a higher quality development environment and ready access to 3D graphics for visualization of results.

64-bit operating systems are very likely to support 64-bit file systems. Some application areas like geophysics and particle physics may require file sizes well in excess of two gigabytes (the general limit for 32-bit file systems).

Using 64-bit systems may also have miscellaneous benefits common to 64-bit processors and systems such as use of extra registers (particularly for floating point), improved subroutine argument passing, and faster memory to memory data movement.

Some of these advantages may be available in the 32-bit mode of 64-bit systems, though they are generally most conveniently used when running true 64-bit programs. For example, efficient 64-bit logical and arithmetic operations may be available to 32-bit programs.

2 Generic 64-bit Portability Problems

There are a number of things to watch for when porting existing 32-bit code to 64-bit. If you are used to writing code for 32-bit machines, misplaced assumptions may be invalid for 64-bit code.

2.1 Listen to Your Compiler

The best advice for making sure code will operate correctly in a 64-bit environment is to pay attention to the warnings generated by your compiler. Because 64-bit compiler writers anticipate 32-bit code will be ported to 64-bit systems, compilers for 64-bit systems tend to generate portability warnings about coding practices that are not portable or risky in 64-bit environments. Even modern 32-bit compilers generate reasonable warnings relevant to 64-bit porting. If your compiler does not generate particularly comprehensive warnings, try `lint` or a better compiler.

When you compile 32-bit code on a 64-bit system, turn on the maximum useful warnings and study the output for the kinds of problems discussed below. Definitely use ANSI C or C++ prototypes to maximize the error checking. Prototypes verify correct type matching across function boundaries.

2.2 Bad Assumptions about Data Types

So what types of error should concern you? Most of the problems result from assumptions, implicit or explicit, about either the absolute or relative sizes of the `int`, `long`, and pointer data types. Here are common faulty assumptions that undermine 64-bit porting:

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

  This assumption occurs when a pointer is cast to an `int` to perform pointer arithmetic. The assumption can also occur when a union is used to hold both an `int` and a pointer, or when an `int` or pointer is passed as a parameter to a routine actually requiring the opposite type.

  Assuming prototypes are used, then casting and parameter passing problems are detected by good compilers. The union problem is subtler and harder for compilers to detect.

One way to avoid the pointer arithmetic problem is to use the ANSI C `ptrdiff_t` type found in the `stddef.h` header file. This integer type is defined to be large enough to hold an integer representation of a pointer and is expressly designed for portably performing pointer arithmetic. For example, instead of:

```
int
ptr_dist(void *a, void *b) {
  return (int) a - (int) b;
}
```

write:

```
#include <stddef.h>
ptrdiff_t
ptr_dist(void *a, void *b) {
  return (ptrdiff_t) a
         - (ptrdiff_t) b;
}
```

- `sizeof(int) == sizeof(long)`

  This assumption is similar the previous assumption. In the **32BIT** model, these two types are both logically integers and are of the same size, so it is common for a routine to expect, for instance, an `int` and still work correctly when passed a `long`. Be particularly careful when mixing signed and unsigned versions of `long` and `int` since an unsigned value may be unintentionally sign-extended. Again, proper prototypes catch or avoid most such errors.

- `sizeof(long) == 4`

  This assumption manifests itself when a `long` (often embedded in a structure) is used to map external data representations intended to be 32 bits. This can often happen when reading or writing binary data files or encoding or decoding protocols. The following 32-bit code fragment would not be portable to a 64-bit system for this reason:

```
#include <stdio.h>
/* header struct changes size
   when compiled LP64! */
struct header {
  long tag;
  long length;
};
main(int argc, char **argv) {
  FILE *file;
  struct header info;
  file = fopen("data", "r");
  fread(&info, sizeof(info), 1, file);
}
```

The `info` variable has a different size on a **LP64** system than on a **32BIT** system, implying that different data would be read, and the `tag` and `length` members would get a different value on each system when reading the same file. Unfortunately, this type of error is not caught by the compiler.

Also, do not assume a `long` is four bytes when using unions. The following example is not portable:

```
union {
  char c[4];
  long l;
} combo;
main(int argc, char **argv) {
  combo.c[0] = 'c';
  combo.c[1] = 'a';
  combo.c[2] = 't';
  combo.c[3] = '\0';
  /* wrong if sizeof(long) != 4 */
  if(combo.l == 0x63617400)
    printf("Big endian\n");
  else
    printf("Little endian\n");
}
```

This code could be used to determine the byte order of a 32-bit system, but would not work correctly on a 64-bit **LP64** system since a `long` is not four bytes long.

- `sizeof(void*) == 4`

  This assumption is analogous to the previous one. Because use of pointers in file and protocol formats is dubious, this assumption is rare.

- Assumptions about constants and arithmetic.

  Size changes to the integer types can result in unexpected results from arithmetic using constants due to conversions between signed and unsigned types. Be particularly careful when using constants with the high-order bit set. For example:

```
long x, y;
x = 3;
/* 32BIT truncates y to 32 bits,
   LP64 does not */
y = x + 0xffffffff;
```

In the **32BIT** model, the result is 2. In the **LP64** model, the result is 4,294,967,298. The 0xffffffff constant is treated as an unsigned constant in both cases, but in the 32-bit model, the result is truncated to 32 bits.

Beware of other assumptions about how the results of integer operations are truncated. This fragment demonstrates how different models handle a shift operation:

```
unsigned long a = 0xff000000;
if(a << 8 == 0)
```

```
    printf("32BIT, PC, or LLP64");
  else
    printf("LP64 or ILP64");
```

In the **LP64** case, the shifted set bits are not truncated as in the 32-bit case. Compilers can warn you about most of these assumptions.

Be aware that you can use the `L` and `U` suffixes (lowercase `l` and `u` are also valid) to indicate integer constants are `long` and `unsigned` respectively. These suffixes can be used in combination to indicate an `unsigned long` constant.

2.3 Poorly Sized Malloc Calls

Using standard C library routines is also prone to errors arising from invalid assumptions. The `malloc` memory allocation routine is a good example. While a bad practice, the value passed to `malloc`'s allocation size parameter may be based on implicit type size assumptions. For example:

```
#include <stdlib.h>
long *list20;
list20 = (long*) malloc(20*4);
```

instead of:

```
list20 =
  (long*) malloc(20 * sizeof(long));
```

Compilers are unlikely to report such problems. You can find these hard-to-find problems by inspection of the code and the use of a debugging `malloc` package during testing. Other routines requiring byte extent lengths like `bcopy`, `bzero`, `memset`, `memcpy`, and `memmove` are subject to the same problem.

2.4 Careful Using Variable Argument Lists

Routines that take variable argument lists can often hide 64-bit portability problems. The most common examples of such routines are standard routines like `print`, `scanf`, `fprintf`, `syslog`, `vprintf`, and their relatives. In the case of `printf`-like routines, the variable number of arguments of various types are interpreted at run-time by a format string parameter. Routines using variable argument lists tend to obscure 64-bit portability problems for two related reasons:

1. The compiler cannot warn you of argument passing type mismatches.

2. The compiler may not promote parameters to expected types.

Here is a non-portable program demonstrating the first problem:

```
#include <stdio.h>
main(int argc, char **argv)
{
   char *thirty = "30";
   long value;
   sscanf(thirty, "%d", &value);
   printf("result = %d\n", value);
}
```

On a 32-bit system, the program outputs 30 as expected. On a 64-bit **LP64** system, the program outputs an undefined value. The reason is because `value` is declared as a `long` but is printed out by the `%d` format specifier intended for `int` values. The program can be fixed by either changing the type of `value` to `int` or by changing the `sscanf` format string to be `%ld` to scan into a `long` value.

More 64-bit `printf`-related advice is to use the `%p` format specifier (defined in the ANSI C library specification) when printing pointers values (generally, this is limited to debugging purposes). Using the more traditional `%x` specifier to print pointers as hexadecimal integers will output a truncated representation of 64-bit pointers. Bogus debugging output can only make debugging your 64-bit code harder.

To demonstrate the second problem, consider a routine to print a null terminated list of character strings. Remember that a null pointer is represented by the integral value 0. The routine is implemented portably as:

```
#include <stdarg.h>
void print_list(char *word, ...)
{
   va_list ap;

   va_start(ap, word);
   while(word) {
      puts(word);
      word = va_arg(ap, char *);
   }
   va_end(ap);
}
```

But, problems can result when you call the routine in an unportable manner. For example:

```
  print_list("hi", "there", 0);
```

Trailing variable arguments beyond the explictly typed arguments of `print_list` suffer the default argument promotion rules. It is ambiguous whether the trailing zero should be passed as a pointer or an integer. If treated as an integer, default argument promotion rules would pass the 0 as an `int` which is not the size of a 64-bit **LP64** pointer. The call to `print_list` could be portably rewritten:

```
  print_list("hi", "there", (void*) 0);
```

Using the standard `NULL` symbolic constant found in `stdio.h` or `stdlib.h` should also ensure portability.

Type name	Meaning
int8_t	exactly 8-bit signed integer
uint8_t	exactly 8-bit unsigned integer
int16_t	exactly 16-bit signed integer
uint16_t	exactly 16-bit unsigned integer
int32_t	exactly 32-bit signed integer
uint32_t	exactly 32-bit unsigned integer
int64_t	exactly 64-bit signed integer
uint64_t	exactly 64-bit unsigned integer
intmax_t	largest signed integer supported by the implementation
uintmax_t	largest unsigned integer supported by the implementation
intptr_t	signed integer guaranteed to be exactly the size of a pointer
uintptr_t	unsigned integer guaranteed to be exactly the size of a pointer

Table 2: Explanation of `typedefs` in `inttypes.h`.

2.5 Using inttypes.h

For most of the problems above, safe coding practices avoid the problems, but there are cases where code needs to rely on the physical sizes of data types. For example, the parser for a binary file format that has precisely sized fields may want a data structure containing fields sized to that of the binary file format. But is there a portable way to request a type with exactly 32-bits?

The straightforward way to solve the problem is to develop a set of exactly sized `typedefs`. For example, you could use `int32` when a true 32-bit integer is desired. Then, make sure the `typedef` for `int32` is established to be the correctly sized type in a system-specific header file. While this isolates the portability issue to a single header file, this means this system-specific header file must be updated for each specific system you plan to compile for.

To avoid foisting this general problem on every software developer, 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 type declarations to address exactly this problem. Using `inttypes.h` helps you write code that is portable to any of the possible 32-bit and 64-bit C data type models.

The types defined in `inttypes.h` are described in Table 2. Along with the types, the `inttypes.h` header file defines:

- Maximum and minimum constants for each new integer type. Examples: `INT8_MIN`, `UINT32_MAX`, and `INTMAX_MIN`.
- Safe macros for using exactly sized constants. Example: `INT32_C(0xffffffff)`.
- Extended versions of `printf` and `scanf` family of routines that accept an extended format syntax allowing formatted I/O with the new types. Example:

```
uint16_t u16;
int32_t  s32;
uint64_t u64;
i_printf("int16 is %w16u\n", u16);
i_printf("int32 is %#8w32x\n", s32);
i_scanf("%w16o%w64x", &u16, &u64 );
```

- Versions for each new data type of the ANSI C library `strtol` routine with analogous semantics. Example routine names: `strtoi8` (returning an `int8_t`) and `strtou16` (returning a `uint16_t`).
- Versions for a few of the new data types of the ANSI C library `abs` and `div` routines with analogous semantics. Examples: `abs_32` (taking and returning a `int32_t`) and `div_64` (taking two `int64_ts` and returning a `div64_t`).

In the previous examples showing portability problems that arise due to the changing size of the `long` type in the **32BIT** versus the **LP64** type models, using exactly sized types from `inttypes.h` would avoid the problems. For example, the `struct` and `union` declarations from the previous examples are portably rewritten as:

```
#include <inttypes.h>

union {
  uint8_t c[4];
  uint32_t l;
} combo;

struct header {
  int32_t tag;
  uint32_t length;
};
```

Be careful using `inttypes.h` since older development environments may not support this header file. Also, it is conceivable a future ANSI specification may include a builtin facility for handling exactly sized types.

2.6 Changed Page Sizes

A vendor's 64-bit operating systems may have a different page size from the vendor's 32-bit operating system. For example, IRIX 5.3 has a 4 kilobyte page size, while IRIX 6.0.1 has a 16 kilobyte page size. If your code is tuned to a particular page size or depends on proper page alignment of data structures, you may need to adjust to a new page size when you port to a 64-bit operating system. Most Unix operating systems implement a `getpagesize` routine to return the number of bytes in a page so that you can portably deal with different page sizes.

3 The Effects of 64-bit on X

So far, the discussion has been limited to the generic porting issues for moving code from 32-bit to 64-bit systems. All these generic issues apply to X programs, but there are additional points specific to how the X Window System operates on a 64-bit workstation that you should be aware of.

3.1 The X11 Protocol Remains

The X11 protocol that underlies the X Window System is largely a 32-bit aligned protocol. Because it is a standard, the X11 protocol is fixed and does not change when machines are upgraded to 64-bit operating systems. There is not any justifiable need for a 64-bit version of the X11 protocol. Window system operations work perfectly well manipulating 32-bit values.[2]

It is worth considering if the X server needs to run as a 64-bit process on a 64-bit system. On systems like Digital's Alpha where 64-bit is the only option, the X server necessarily runs as a 64-bit process. Excepting the case where only 64-bit is supported, the X server does not need to run as a 64-bit process. On systems that also support 32-bit processes like SGI's IRIX 6.0.1, the X server does operate as a 32-bit process. Indeed, by running as a 32-bit process, the memory footprint of the X server is smaller than if it ran as a 64-bit process.

3.2 The 64-bit Xlib Interface

What needs to change on a 64-bit system is the implementation of Xlib that provides the C language interface to the X11 protocol. While the X11 protocol does not change, there must be changes in Xlib's implementation for 64-bit systems to reflect that fundamental C data types have enlarged, while the X11 protocol remains fixed.

When Xlib was originally conceived, most machines were 32-bit or 16-bit architectures. Because the X11 protocol mandated that certain values must be treated as 32-bit quantities (like pixels, XIDs, window event masks, and 32-bit window property values), Xlib's designers used the `long` type for such values. C implementations for both 16-bit and 32-bit architectures generally implemented the `long` type as a 32-bit value. While X was not particularly appropriate for 16-bit computers, the Xlib API was careful to not preclude 16-bit implementations.

The result of this decision to make necessarily 32-bit values in the Xlib API be `typedef`ed or declared of type `long` is that when the API is implemented on a 64-bit system using the **LP64** data type model, these `long` values are now 64-bit values. At the API level, because 32-bit values fit in 64-bit `long` types with plenty of room to spare, if you follow all the typing rules of the Xlib API correctly, your X code can recompile and be expected to run properly in a 64-bit environment. But within Xlib's implementation, when data passed through the API is encoded into or decoded from X11 protocol, care must be taken converting between the API and protocol representations of data. Specifically, data declared to be of type `long` in the API must be resized when moving between the 64-bit API and 32-bit protocol representation of the data. Fortunately, this work is hidden within Xlib's implementation.

Xlib's 64-bit support was completed with the X Consortium's X11R6 release.[3] The 64-bit prototypes for the Xlib API are the same as the 32-bit prototypes.

3.3 X API Types That Change Size

There are a reasonably significant number of type declarations in the various standard X APIs that are based on the size of `long`, and these types will increase in size relative to their **32BIT** sizes when compiling with a 64-bit **LP64** data type model. Be careful in your code not to assume the `sizeof` these types is the same as the `sizeof(int)`. Table 3 lists the types to be wary of.

3.4 Pitfalls of 32-bit Window Properties

There are a few obscure pitfalls in X client code. One area is dealing with X window properties. X window properties are named and typed arrays of data that can be associated with a window. Properties are useful for inter-client communication, particularly with the window manger.

Properties come in 8-bit, 16-bit, and 32-bit data formats. Be careful when setting or querying a property with 32-bit data. When specifying a 32-bit property data format using `XChangeProperty` or `XGetWindowProperty`, the property data is laid out in an array of `long` values. This is not obvious from the prototypes of these routines that deceptively claim the data is passed to the routine as an `unsigned char*`. In truth, 32-bit property data is specified to be laid out in an array of `long` sized data. While in the 32-bit world, an array of `long`s would mean the 32-bit values were arranged in a contiguous list, in the 64-bit **LP64 model**, the array will be of 64-bit `long` values, each logically holding a 32-bit value.

[2] The sufficiency of 32-bit sizes for the X11 protocol is contrasted with the need to upgrade the Network File System (NFS) protocol to support 64-bit file systems generally associated with 64-bit operating systems. NFS version 3 provides support for huge files in excess of two gigabytes (the limit for NFS version 2). While huge file sizes can surpass 32-bit limits justifying 64-bit protocols like NFS version 3, window system operations are unlikely to ever exceed the restrictions of 32-bit data value ranges.

[3] X11R5 included code for Cray's particular 64-bit architecture, but the support for the **LP64** data type model was not complete until X11R6.

Xlib	Xmu	Xt	Xaw	Xm
Atom	Pixel	EventMask	XawTextPosition	XmOffset
Colormap	XctFlags	Pixel		XmTextPosition
Cursor		XtBlockHookId		
Drawable		XtGCMask		
Font		XtInputId		
GContext		XtInputMask		
KeySym		XtIntervalId		
Mask		XtSignalId		
Pixmap		XtValueMask		
Time		XtVersionType		
VisualID		XtWorkProcId		
Window				
XID				
XIMFeedback				
XIMHotKeyState				
XIMPreeditState				
XIMResetState				
XIMStringConversionFeedback				
XIMStyle				
XcmsColorFormat				

Table 3: Type declarations in various X APIs that are based on the size of `long`.

The following code would not be portable to a 64-bit **LP64** machine:

```
/* overlayInfo should be
   pointer to long, not int! */
int *overlayInfo;
Atom overlayVisualsAtom, actualType;
unsigned long sizeData, bytesLeft;
int actualFormat;

overlayVisualsAtom = XInternAtom(display,
  "SERVER_OVERLAY_VISUALS", True);
status = XGetWindowProperty(dpy, root,
  overlayVisualsAtom, 0L, 10000L, False,
  overlayVisualsAtom, &actualType,
  &actualFormat, &sizeData, &bytesLeft,
  (unsigned char**) &overlayInfo);
if(status != Success ||
   actualType != overlayVisualsAtom ||
   actualFormat != 32 ||
   sizeData < 4)
{
  /* bogus property */
} else {
  /* process info returned in array
     overlayInfo */
}
```

While the code compiles correctly, the subtle problem is that the `overlayInfo` variable (first line) is declared to be an array of `int` data, instead of `long` data as Xlib expects it to be if the property is to contain 32-bit data.[4] On a 64-bit **LP64** machine, the `overlayInfo` contains scattered pieces of the actual data for the property. Because the cast of `overlayInfo` to `(unsigned char**)` is necessary to match the prototype of `XGetWindowProperty`, this kind of subtle typing problem cannot be caught by compilers.

Code using 8-byte or 16-byte property data is likely to work without modification because the `char` and `short` data types do not change size.

3.5 Careful with X Images

Another area of Xlib programming that is prone to 64-bit portability problems is use of Xlib's image routines that manipulate `XImage*` data structures: `XCreateImage`, `XGetImage`, `XPutImage`, `XGetSubImage`, etc. If the routines used to manipulate pixels in `XImage*` objects are limited to the standard interfaces like `XGetPixel` and `XPutPixel`, your code is very likely to be portable.

But with sufficient understanding of the format of image data, application code can be rewritten without `XGetPixel` and `XPutPixel` to substantially speed up pixel operations on `XImage*` objects. The `XGetPixel` and `XPutPixel` routines have fairly high overhead due to their generality and tend to be called repeatedly so this is a good area for optimization.

[4] The `SERVER_OVERLAY_VISUALS` is used as a convention adopted by SGI, HP, and other X vendors to encode information about X server overlay and underlay visuals as explained in the July/August '93 issue of *The X Journal*.

Such optimizations are ripe for 64-bit portability bugs because the code involved is often dependent on the size of basic data types. Potentially, with more tuning, you can actually improve the performance of your image operations by using 64-bit operations to move data faster.

A good idea when optimizing image operations is to code an unoptimized, safe version of the operation using `XGetPixel` and `XPutPixel`. This makes it easy to fall back to the safe version of the image operation if the assumptions for the optimization are incorrect.

3.6 Xt Argument Lists

The X Toolkit has two programmatic interfaces to setting and getting widget resource values. Both interfaces circumvent the compiler's ability to catch type mismatches and are therefore prone to 64-bit portability problems, particularly when getting resources. The older interface sets and gets resource values through the construction of `ArgList` resource templates. These templates allow no opportunity for compiler type checking because resource values in the argument lists are considered having dynamic type.

The example below works fine on a 32-bit system and will compile without errors on a 64-bit system, but the 64-bit executable has a possibly fatal bug:

```
#include <Xm/Xm.h>
#include <Xm/Label.h>
main(int argc, char **argv)
{
  XtAppContext app_context;
  /* pixel should be of type long! */
  int padding_needed_to_demo_bug, pixel;
  Widget toplevel, hello;
  Arg arg;

  toplevel = XtVaAppInitialize(
    &app_context, "xhello", NULL, 0,
    &argc, argv, NULL, NULL);
  hello = XtVaCreateManagedWidget(
    "hello", xmLabelWidgetClass,
    toplevel, NULL);
  XtSetArg(arg, XmNbackground, &pixel);
  XtGetValues(hello, &arg, 1);
  printf("pixel = %ld\n", pixel);
  XtRealizeWidget(toplevel);
  XtAppMainLoop(app_context);
}
```

Compiled as a 64-bit executable on an IRIX 6.0.1 machine, the program incorrectly outputs zero and corrupts the word of memory following the `pixel` variable. One observed result is the corruption of the `toplevel` widget value, crashing the program. If the `pixel` variable is declared to be of type `Pixel`, the program works as expected. The unmodified program incorrectly assumes that the X Toolkit's `Pixel` type is the same size as an `int`.

The second, newer interface is potentially more prone to errors because it relies on variable argument lists. The program above would also fail in the same way if the `XtSetArg` and `XtGetValues` lines were replaced with:

```
XtVaGetValues(hello,
  XmNbackground, &pixel, NULL);
```

Widget resources of type `Pixel`, `XawTextPosition`, and `XmTextPosition` can be easily used incorrectly as in the above example. Be careful since the underlying type for each of these types is `long`, not `int`.

3.7 Extension Writers Beware

Client libraries for X extensions are particularly prone to 64-bit portability problems because X client library code performs the task of correctly converting to and from the API world of C types to the protocol world's fixed protocol fields. It is not surprising that some X extension client library code in the X11R6 distribution (the X Input extension library is an example) was not 64-bit clean.

If an X extension client library is compiled 64-bit but is not actually 64-bit clean, the likely result is X protocol errors, though unreported incorrect results or crashes are also possible.

For the X extension client library author, be sure to use the `Data32`, `Data16`, and `Data` macros for writing appropriately sized protocol data, and use the `_XRead32`, `_XRead16`, and `_XRead` routines for reading appropriately sized protocol data (`Data` and `_XRead` are for 8-bit data). Also, inspect the X11R6 `Xlibint.h` header file and see how the `LONG64` and `WORD64` compile flags are used. `LONG64` is defined by 64-bit environments using the **LP64** data type model, i.e., the `long` data type is 64 bits long. `WORD64` is defined by 64-bit environments using the **ILP64** data type model, i.e., the `int` data type is 64 bits long.

Conclusions

As 64-bit X workstations and servers become more commonplace, more X applications and libraries will be ported to or developed for 64-bit operating environments. While Digital and SGI are at the forefront of this trend, you can expect other workstation vendors to follow their lead. If you are well informed about the portability issues involved in moving to 64-bit systems, you and your X code will be ready for 64-bit systems and the process of migrating to 64-bit systems will be much easier.