11 How To Make Pre 11.0 Drivers 64-Bit Safe
How To Make Pre 11.0 Drivers 64-Bit Safe

We recommend that you modify existing 32-bit drivers to be 64-bit clean so that they can be compiled to run in either type of kernel.

NOTE
For information about modification of Pre-Release 10.0 drivers, see the previous (HP-UX 10.20) driver Development Guide, HP Part No. B2355-90066)

NOTE
For information about specific driver entry points see the information for that entry point in a previous chapter; for example, driver_ioctl(), in “Writing a driver_ioctl() Routine” on page 131, Chapter 5.
Introduction

HP-UX 11.0 has two versions of the operating system kernel: a 32-bit kernel and a 64-bit kernel. The 64-bit kernel extends the capabilities of the 32-bit kernel in several ways. Among the extensions are the ability to address larger than 4 gigabytes of physical memory, map up to 16 terrabytes of virtual address space for 64-bit application programs and allow 32-bit and 64-bit applications to coexist on the same system.

To integrate your I/O driver with the 64-bit kernel requires an understanding of the differences between the 32-bit and 64-bit data models. It also requires that you make your I/O driver source “64-bit clean” and robust enough to deliver predictable results when executing in the 64-bit kernel environment.

Important Terms and Concepts

This section begins by defining important terms and concepts essential for migration of drivers to the 64-bit environment. Then it describes the 64-bit data model. Next, it discusses general guidelines, and finally discusses the I/O address space.

32-bit program  A program compiled to run in 32-bit mode. For example, programs compiled for the PA-RISC 1.x processors.

64-bit program  A program compiled to run in 64-bit mode. For example, programs compiled for the PA-RISC 2.0 processor in wide mode.

ILP32  C language data model where int, long, and pointer data types are 32 bits in size.

LP64  C language data model where the int data type is 32 bits wide, but long and pointer data types are 64 bits wide.

SIO  Server I/O: I/O environment for port-server drivers with origins in S/800 systems.

WSIO  Workstation and Server I/O: I/O environment for reentrant drivers with origins in S/700 systems and converged with S/800 10.x systems.
The 64-bit Data Model

ILP32 is a C language data model where int, long, and pointer data types are 32 bits wide. This is the data model commonly used by 32-bit operating systems, including the 32-bit HP-UX.

LP64 is another C language data model where the int data type remains 32 bits wide, but long and pointer data types are scaled up to 64 bits. This is the data model that has been adopted by HP for its 64-bit HP-UX implementation.

Table 11-1, “Summary of LP64 and ILP32 Data Models,” summarizes the two data models.

<table>
<thead>
<tr>
<th>Data type</th>
<th>LP64 bit size</th>
<th>ILP32 bit size</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>long long</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>pointer</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>long double</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>struct</td>
<td>depends on members</td>
<td>depends on members</td>
</tr>
<tr>
<td>enum</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>
LP64 Considerations

Size differences between the ILP32 and LP64 data models occur with long and pointer data types. As a consequence, the default integral data type int can differ in size from long. This subtle difference can cause results unintended by the programmer.

Consider the following example:

```c
main() {
    long L = -1;
    unsigned int i = 1;
    if (L > i)
        printf ("L greater than i\n");
    else
        printf("L not greater than i\n");
}
```

Under ANSI C integral promotion rules, if `sizeof(int)` equals `sizeof(long)` this will print

"L greater than i"

However, if `sizeof(int)` is less than `sizeof(long)` this will print

"L not greater than i"

Both results are ANSI conforming, correct, and consequences of the value preserving integral promotion rules of ANSI. If the same code is compiled in K&R mode, the unsigned preserving integral promotion rules of K&R will result in the program printing "L greater than i" with ILP32 and LP64.

Suppose we modify this example and change L to int. Under ANSI and K&R integral promotion rules, this will print "L greater than i" for both ILP32 and LP64.

This example illustrates that caution needs to be exercised when a data type is changed to or from long. The HP C compiler provides the option +M to identify code where ANSI and K&R may differ.

Another consideration is the alignment of data objects. With LP64, structures that contain long or pointer data types will be aligned to a double word offset. As an example, consider the following:
How To Make Pre 11.0 Drivers 64-Bit Safe

The 64-bit Data Model

```c
struct vals {
    int intval;
    long longval;
    int endval;
};
```

The data member longval has the strictest alignment requirement of the data members in the structure. In LP64 mode, the entire structure and longval are double word aligned. A 32-bit gap will exist between intval and longval. The compiler may also pad the size of the structure to the next double word size.
General Guidelines

64-bit Clean Headers

Header files contain the data declarations, structures, constants, macros, function prototypes, and external data objects that are the interfaces to modules. The same header files are expected to be compiled with source code for 32-bit and 64-bit drivers, and possibly with 32-bit and 64-bit libraries and applications.

To make this possible, header files need to be examined for declarations and usage that may not be compatible between the ILP32 and LP64 data models. This is referred to as making header files 64-bit clean.

Here are the general guidelines to clean an I/O driver header file:

• Examine declarations where long (or a variant of long) is specified. Where appropriate, fix the size of the declaration by replacing long or a variant of long by int (or a variant of int). Cases where this may be appropriate include application visible data structures that must be sized the same for both ILP32 and LP64, driver data structures that ought to be kept from growing unnecessarily large, and hardware data structures that must be fixed in size with 32-bit data types.

• Examine declarations where int (or a variant of int) is specified. Where appropriate, scale the size of the declaration by replacing int (or a variant of int) by long (or a variant of long). Cases where this may be appropriate include offsets that must displace greater than 4 GBytes in a 64-bit kernel, storage that is overloaded to store a pointer value, and storage for machine register values.

• Examine declarations where pointer data types are specified. If the pointer is a pointer to a function, specify the full ANSI function prototype with all arguments declared. This enables ANSI code to be type checked against the function prototype and expose incompatibilities at compile-time. If the pointer is in an application visible data structure, things get complicated because the application may be a 32-bit application (compiled with ILP32 data types) and your driver may be executing in the 64-bit kernel (compiled with LP64 data types). This situation is discussed in “Writing a driver_ioctl() Routine” on page 131, Chapter 5.
Useful Data Types

int32_t and uint32_t
The data types int32_t and uint32_t represent storage that must be fixed in size to 32 bits. They are declared in the header file _inttypes.h as:

    typedef int int32_t;
    typedef unsigned int uint32_t;

Consider the following data structure from the 10.20 header file diskio.h:

    typedef struct {
        long lba; /* capacity in DEV_BSIZE blocks */
    } capacity_type;

Contained in the structure is the data member lba declared as long. This is an application visible data structure that should be fixed in size to avoid compatibility problems between a 32-bit application and a 64-bit driver. For example,

    typedef struct {
        int32_t lba; /* capacity in DEV_BSIZE blocks */
    } capacity_type;

Since lba is intended to store values greater than or equal to zero and negative values are never stored, the correct declaration for lba is an unsigned type. The following is the 64-bit clean version of capacity_type in release 11.0:

    typedef struct {
        uint32_t lba; /* capacity in DEV_BSIZE blocks */
    } capacity_type;

intptr_t and uintptr_t
The data types intptr_t and uintptr_t represent storage that must scale in size with a pointer data type. They are declared in the header file _inttypes.h as:

    typedef long intptr_t; typedef unsigned long uintptr_t;

Consider the following macro from the 10.20 header file cpu.h:
How To Make Pre 11.0 Drivers 64-Bit Safe

General Guidelines

Chapter 11

#define ALIGN_IOBUF(P) ((char *)(((uintptr_t)(P) & ~((CPU_IOLINE-1))))

The pointer P is cast as (int), but the cast does not scale in size with the pointer. The cast is changed to (uintptr_t) in release 11.0 as shown below:
#define ALIGN_IOBUF(P) ((char *)((int)(P) + CPU_IOLINE-1) & ~(uintptr_t)CPU_IOLINE-1))

The cast in the second line explicitly promotes the value CPU_IOLINE-1 to uintptr_t before complementing the value.

ptr32_t

The data type ptr32_t represents storage for a 32-bit pointer and is declared in _inttypes.h as:
typedef uint32_t ptr32_t;

For an example see “Writing a driver_ioctl() Routine” on page 131, Chapter 5.

int64_t and uint64_t

The data types int64_t and uint64_t represent storage that must be fixed in size to 64 bits. They are declared in the header file _inttypes.h as:
typedef long long int64_t;
typedef unsigned long long uint64_t

Be aware that application code compiled with strict ANSI (compiler option -Aa) will not recognize the data type long long. However, long long is accepted in extended ANSI (compiler option -Ae ) and K&R (compiler option -Ac ) modes.

__LP64__

__LP64__ is defined when source code is compiled with LP64 data types. #ifdef __LP64__ may be useful where there are special LP64 considerations. For example, consider the following data structure in the 11.0 header file dma_A.h:
typedef struct {
    lock_t *lock;
    dma_A_chain_type *chain_list;
}
How To Make Pre 11.0 Drivers 64-Bit Safe

General Guidelines

```c
#ifdef __LP64__
    uint32_t pad[12]; /* align it to 64 bytes */
#else /* !__LP64__ */
    uint32_t pad[14]; /* align it to 64 bytes */
#endif /* !__LP64__ */
} dma_A_pool_t;
```

#ifdef __LP64__ is used to pad the data structure so that the size is 64 bytes in both ILP32 and LP64 modes.

Hardware Considerations

I/O drivers often map data structures into I/O hardware registers and memory areas that are accessed by I/O hardware. Care must be exercised to ensure that these structures are fixed in size when compiled with LP64 data types.

Consider the data structure compl_head in the 10.20 header file llio.h:

```c
struct compl_head {
    int sema;
    struct compl_entry *link;
    int filler[2];
};
```

A compl_head structure is updated by hardware when an I/O request completes. The link data member is viewed by hardware as a 32-bit value. With LP64, however, link will be scaled to 64 bits.

The first step is to fix the size of the structure. In release 11.0, the structure is declared as:

```c
struct compl_head {
    uint32_t sema;
    uint32_t link;
    uint32_t filler[2];
};
```

The next step is to examine code that accesses the compl_head and make corrections as needed.
Software Considerations

While data structures that map onto hardware and must be fixed in size are easily identified, data structures with software considerations can be much more subtle. Rather than elaborating on the many possible programming pitfalls, we present an example. Consider messages sent and received by I/O drivers in the SIO driver environment. In release 10.20, the message header is declared as:

typedef struct {
    shortint   msg_descriptor;
    shortint   message_id;
    int        transaction_num;
    port_num_type   from_port
} llio_std_header_type;

Notice that no data member in the structure is larger than 32 bits. As such, the size of llio_std_header_type is 12 bytes. In iLP32 and LP64 modes, sizeof(llio_std_head_type) returns the value 12.

The programming assumption that causes a problem is that the message body immediately follows the header. This assumption holds with ILP32 data types; but with LP64 data types, the message body happens to contain 64-bit data members that cause the compiler to align the body to a 64-bit boundary. The header is also aligned to a 64-bit boundary and this creates a 32-bit hole between the header and body of the message. The message is declared as:

typedef struct {
    llio_std_header_type msg_header;
    union {
        creation_info_type   creation_info;
        do_bind_req_type    do_bind_req;
    } union_name;
} io_message_type;

To send a message, SIO drivers add the sizeof(llio_std_header_type) to the size of the specific message.
General Guidelines

body type being sent. In LP64 mode, the calculated size of messages are 4 bytes less than the programmer intended.

A possible solution is to explicitly pad the message header as follows:

typedef struct {
    shortint   msg_descriptor;
    shortint   message_id;
    int         transaction_num;
    port_num_type from_port;

    #ifdef __LP64__
    int filler; /* pad to 64-bit boundary */
    #endif /* __LP64__ */
} llio_std_header_type;

ANSI Function Prototypes

With LP64, programmers can no longer assume that int, long, and pointer data types are the same size, and that these data types can be mixed across function call arguments and function return types. Many kernel functions return pointer or long data types. Without the appropriate function prototypes in scope, the compiler incorrectly truncates the return type to int. Programmers need to provide appropriate ANSI function prototypes (or K&R forward declarations) to their I/O drivers.

Compile time errors will result, however, if ANSI function prototypes are visible to the K&R compiler. To prevent such errors, kernel header files use the __( ) macro to declare prototypes. The macro is defined in the header file stdsyms.h as follows:

#if defined(_PROTOTYPES)
#define __(arg) arg
#else
#define __(arg) ()
#endif

To illustrate the __( ) macro, consider the following function prototype:

void *fl __(long arg1, int arg2);
Note the double parenthesis surrounding the arguments. This notation is equivalent to the cumbersome form it replaces:

```c
#if defined (_PROTOTYPES)
void *fl(long arg1, int arg2);
#else
void *fl();
#endif
```

Many kernel header files have incorporated ANSI function prototypes in release 11.0. Programmers should make sure that function prototypes are in scope for all kernel interfaces called by their drivers by including the appropriate header files.

To take full advantage of function prototypes, set the compiler option to `-Ae` to compile in extended ANSI mode. If your driver is compiled K&R, now is the time to convert to ANSI.

**kern_svcs.h header file**

A new header file, `kern_svcs.h`, has been added to `/usr/include/sys` and `/usr/conf`. This header file declares ANSI function prototypes of common kernel services.

Driver writers do not need to explicitly include `kern_svcs.h`. WSIO drivers include `kern_svcs.h` when `wsio.h` is included.

When porting a driver you should always check your routine declarations and parameters against `kern_svcs.h`.

**conf.h header file**

Among the changes made to `conf.h` are the inclusion of ANSI function prototypes for the function pointers declared in the `cdevsw` and `bdevsw` structures. Driver entry points are expected to match the function prototypes declared therein.

**dma.h header file**

Several changes have been made to the `dma.h` header file. WSIO specific macro definitions have been moved to `wsio.h`. Additionally, function prototypes have been added to the function pointer declarations in the `io_map_cntrl` structure. These prototypes enable type checking of arguments that are passed to the WSIO and SIO mapping services.
Consider the `wsio_dma_alloc` macro:

```c
#define wsio_dma_alloc(Isc, Iova) \
     (*Isc->map_funcs->dma_alloc)(Isc->map_funcs->arg, Iova)
```

The macro calls the function pointed to by `dma_alloc` in the `io_map_cntl` structure. The second argument to the macro is `Iova` which is prototyped by `dma_alloc` as `caddr_t *`. Suppose the driver calls this macro with the following code:

```c
uint32_t iova; /* only 32 bits of storage */
caddr_t pva = wsio_dma_alloc(isc, &iova);
```

This is clearly a type mismatch (and a programming error with LP64) for which the ANSI C compiler will generate a warning message. For DMA hardware that uses 32-bit `iova` values, the driver can do the following:

```c
uint32_t iova; /* only 32 bits of storage */
caddr_t tmp_iova;
caddr_t pva = wsio_dma_alloc(isc, &tmp_iova);
iova = (uint32_t)tmp_iova;
```

### uio.h header file

The `uio.h` header file is the declaration of the `iovec` structure. A pointer to the `iovec` structure is used not only in the `uio` structure, but is also specified as an argument type to the DMA mapping services. The `iovec` structure is declared as:

```c
struct iovec {
    caddr_t iov_base;
    size_t iov_len;
};
```

With LP 64 data types, the `iov_base` and `iov_len` fields will each require 64 bits of storage and must be aligned to a double word (64bit) address. I/O drivers that declare alternate data structures that map onto the `iovec` structure may require changes. For example, consider a driver that declares a data structure that represents an address/count pair as:

```c
struct ac_pair {
    uint32_t iov_base;
    uint32_t iov_len
};
```
The driver must not pass a pointer to an `ac_pair` as an argument to the DMA mapping services where a pointer to an `iovec` is expected.
How To Make Pre 11.0 Drivers 64-Bit Safe

General Guidelines