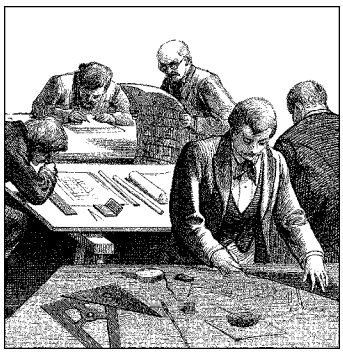
CHAPTER TEN

JUDICIOUS USE OF DATA TYPES



Before we go on to more advanced topics, we need to stop for a quick note on portability issues. Modern versions of the Linux kernel are highly portable, running on several very different architectures. Given the multiplatform nature of Linux, drivers intended for serious use should be portable as well.

But a core issue with kernel code is being able both to access data items of known length (for example, filesystem data structures or registers on device boards) and to exploit the capabilities of different processors (32-bit and 64-bit architectures, and possibly 16 bit as well).

Several of the problems encountered by kernel developers while porting x86 code to new architectures have been related to incorrect data typing. Adherence to strict data typing and compiling with the *-Wall -Wstrict-prototypes* flags can prevent most bugs.

Data types used by kernel data are divided into three main classes: standard C types such as `int`, explicitly sized types such as `u32`, and types used for specific kernel objects, such as `pid_t`. We are going to see when and how each of the three typing classes should be used. The final sections of the chapter talk about some other typical problems you might run into when porting driver code from the x86 to other platforms, and introduce the generalized support for linked lists exported by recent kernel headers.

If you follow the guidelines we provide, your driver should compile and run even on platforms on which you are unable to test it.

Use of Standard C Types

Although most programmers are accustomed to freely using standard types like `int` and `long`, writing device drivers requires some care to avoid typing conflicts and obscure bugs.

The problem is that you can't use the standard types when you need "a two-byte filler" or "something representing a four-byte string" because the normal C data types are not the same size on all architectures. To show the data size of the various C types, the *datasize* program has been included in the sample files provided on the O'Reilly FTP site, in the directory *misc-progs*. This is a sample run of the program on a PC (the last four types shown are introduced in the next section):

```
morgana% misc-progs/datasize
arch   Size:  char  shor   int  long   ptr long-long  u8 u16 u32 u64
i686            1     2     4     4     4     8        1   2   4   8
```

The program can be used to show that `long` integers and pointers feature a different size on 64-bit platforms, as demonstrated by running the program on different Linux computers:

```
arch   Size:  char  shor   int  long   ptr long-long  u8 u16 u32 u64
i386            1     2     4     4     4     8        1   2   4   8
alpha           1     2     4     8     8     8        1   2   4   8
armv4l          1     2     4     4     4     8        1   2   4   8
ia64            1     2     4     8     8     8        1   2   4   8
m68k            1     2     4     4     4     8        1   2   4   8
mips            1     2     4     4     4     8        1   2   4   8
ppc             1     2     4     4     4     8        1   2   4   8
sparc           1     2     4     4     4     8        1   2   4   8
sparc64         1     2     4     4     4     8        1   2   4   8
```

It's interesting to note that the user space of *Linux-sparc64* runs 32-bit code, so pointers are 32 bits wide in user space, even though they are 64 bits wide in kernel space. This can be verified by loading the *kdatasize* module (available in the directory *misc-modules* within the sample files). The module reports size information at load time using *printk* and returns an error (so there's no need to unload it):

```
kernel: arch   Size:  char short int long  ptr long-long u8 u16 u32 u64
kernel: sparc64         1    2    4    8    8     8       1   2   4   8
```

Although you must be careful when mixing different data types, sometimes there are good reasons to do so. One such situation is for memory addresses, which are special as far as the kernel is concerned. Although conceptually addresses are pointers, memory administration is better accomplished by using an unsigned integer type; the kernel treats physical memory like a huge array, and a memory address is just an index into the array. Furthermore, a pointer is easily dereferenced; when dealing directly with memory addresses you almost never want to dereference them in this manner. Using an integer type prevents this dereferencing, thus avoiding bugs. Therefore, addresses in the kernel are `unsigned long`, exploiting the fact that pointers and `long` integers are always the same size, at least on all the platforms currently supported by Linux.

The C99 standard defines the `intptr_t` and `uintptr_t` types for an integer variable which can hold a pointer value. These types are almost unused in the 2.4 kernel, but it would not be surprising to see them show up more often as a result of future development work.

Assigning an Explicit Size to Data Items

Sometimes kernel code requires data items of a specific size, either to match predefined binary structures* or to align data within structures by inserting "filler" fields (but please refer to "Data Alignment" later in this chapter for information about alignment issues).

The kernel offers the following data types to use whenever you need to know the size of your data. All the types are declared in `<asm/types.h>`, which in turn is included by `<linux/types.h>`:

```
u8;   /* unsigned byte (8 bits) */
u16;  /* unsigned word (16 bits) */
u32;  /* unsigned 32-bit value */
u64;  /* unsigned 64-bit value */
```

These data types are accessible only from kernel code (i.e., `__KERNEL__` must be defined before including `<linux/types.h>`). The corresponding signed types exist, but are rarely needed; just replace `u` with `s` in the name if you need them.

If a user-space program needs to use these types, it can prefix the names with a double underscore: `__u8` and the other types are defined independent of `__KERNEL__`. If, for example, a driver needs to exchange binary structures with a program running in user space by means of *ioctl*, the header files should declare 32-bit fields in the structures as `__u32`.

It's important to remember that these types are Linux specific, and using them hinders porting software to other Unix flavors. Systems with recent compilers will support the C99-standard types, such as `uint8_t` and `uint32_t`; when possible, those types should be used in favor of the Linux-specific variety. If your code must work with 2.0 kernels, however, use of these types will not be possible (since only older compilers work with 2.0).

You might also note that sometimes the kernel uses conventional types, such as `unsigned int`, for items whose dimension is architecture independent. This is usually done for backward compatibility. When `u32` and friends were introduced in version 1.1.67, the developers couldn't change existing data structures to the

* This happens when reading partition tables, when executing a binary file, or when decoding a network packet.

new types because the compiler issues a warning when there is a type mismatch between the structure field and the value being assigned to it.* Linus didn't expect the OS he wrote for his own use to become multiplatform; as a result, old structures are sometimes loosely typed.

Interface-Specific Types

Most of the commonly used data types in the kernel have their own `typedef` statements, thus preventing any portability problems. For example, a process identifier (pid) is usually `pid_t` instead of `int`. Using `pid_t` masks any possible difference in the actual data typing. We use the expression *interface-specific* to refer to a type defined by a library in order to provide an interface to a specific data structure.

Even when no interface-specific type is defined, it's always important to use the proper data type in a way consistent with the rest of the kernel. A jiffy count, for instance, is always `unsigned long`, independent of its actual size, so the `unsigned long` type should always be used when working with jiffies. In this section we concentrate on use of "`_t`" types.

The complete list of `_t` types appears in `<linux/types.h>`, but the list is rarely useful. When you need a specific type, you'll find it in the prototype of the functions you need to call or in the data structures you use.

Whenever your driver uses functions that require such "custom" types and you don't follow the convention, the compiler issues a warning; if you use the *-Wall* compiler flag and are careful to remove all the warnings, you can feel confident that your code is portable.

The main problem with `_t` data items is that when you need to print them, it's not always easy to choose the right *printk* or *printf* format, and warnings you resolve on one architecture reappear on another. For example, how would you print a `size_t`, which is `unsigned long` on some platforms and `unsigned int` on some others?

Whenever you need to print some interface-specific data, the best way to do it is by casting the value to the biggest possible type (usually `long` or `unsigned long`) and then printing it through the corresponding format. This kind of tweaking won't generate errors or warnings because the format matches the type, and you won't lose data bits because the cast is either a null operation or an extension of the item to a bigger data type.

In practice, the data items we're talking about aren't usually meant to be printed, so the issue applies only to debugging messages. Most often, the code needs only

* As a matter of fact, the compiler signals type inconsistencies even if the two types are just different names for the same object, like `unsigned long` and `u32` on the PC.

to store and compare the interface-specific types, in addition to passing them as arguments to library or kernel functions.

Although `_t` types are the correct solution for most situations, sometimes the right type doesn't exist. This happens for some old interfaces that haven't yet been cleaned up.

The one ambiguous point we've found in the kernel headers is data typing for I/O functions, which is loosely defined (see the section "Platform Dependencies" in Chapter 8). The loose typing is mainly there for historical reasons, but it can create problems when writing code. For example, one can get into trouble by swapping the arguments to functions like *outb*; if there were a `port_t` type, the compiler would find this type of error.

Other Portability Issues

In addition to data typing, there are a few other software issues to keep in mind when writing a driver if you want it to be portable across Linux platforms.

A general rule is to be suspicious of explicit constant values. Usually the code has been parameterized using preprocessor macros. This section lists the most important portability problems. Whenever you encounter other values that have been parameterized, you'll be able to find hints in the header files and in the device drivers distributed with the official kernel.

Time Intervals

When dealing with time intervals, don't assume that there are 100 jiffies per second. Although this is currently true for Linux-x86, not every Linux platform runs at 100 Hz (as of 2.4 you find values ranging from 20 to 1200, although 20 is only used in the IA-64 simulator). The assumption can be false even for the x86 if you play with the `HZ` value (as some people do), and nobody knows what will happen in future kernels. Whenever you calculate time intervals using jiffies, scale your times using `HZ` (the number of timer interrupts per second). For example, to check against a timeout of half a second, compare the elapsed time against `HZ/2`. More generally, the number of jiffies corresponding to `msec` milliseconds is always `msec*HZ/1000`. This detail had to be fixed in many network drivers when porting them to the Alpha; some of them didn't work on that platform because they assumed `HZ` to be 100.

Page Size

When playing games with memory, remember that a memory page is `PAGE_SIZE` bytes, not 4 KB. Assuming that the page size is 4 KB and hard-coding the value is a common error among PC programmers—instead, supported platforms show page sizes from 4 KB to 64 KB, and sometimes they differ between different

implementations of the same platform. The relevant macros are `PAGE_SIZE` and `PAGE_SHIFT`. The latter contains the number of bits to shift an address to get its page number. The number currently is 12 or greater, for 4 KB and bigger pages. The macros are defined in `<asm/page.h>`; user-space programs can use *getpagesize* if they ever need the information.

Let's look at a nontrivial situation. If a driver needs 16 KB for temporary data, it shouldn't specify an `order` of `2` to *get_free_pages*. You need a portable solution. Using an array of `#ifdef` conditionals may work, but it only accounts for platforms you care to list and would break on other architectures, such as one that might be supported in the future. We suggest that you use this code instead:

```
int order = (14 - PAGE_SHIFT > 0) ? 14 - PAGE_SHIFT : 0;
buf = get_free_pages(GFP_KERNEL, order);
```

The solution depends on the knowledge that 16 KB is `1<<14`. The quotient of two numbers is the difference of their logarithms (orders), and both `14` and `PAGE_SHIFT` are orders. The value of `order` is calculated at compile time, and the implementation shown is a safe way to allocate memory for any power of two, independent of `PAGE_SIZE`.

Byte Order

Be careful not to make assumptions about byte ordering. Whereas the PC stores multibyte values low-byte first (little end first, thus little-endian), most high-level platforms work the other way (big-endian). Modern processors can operate in either mode, but most of them prefer to work in big-endian mode; support for little-endian memory access has been added to interoperate with PC data and Linux usually prefers to run in the native processor mode. Whenever possible, your code should be written such that it does not care about byte ordering in the data it manipulates. However, sometimes a driver needs to build an integer number out of single bytes or do the opposite.

You'll need to deal with endianness when you fill in network packet headers, for example, or when you are dealing with a peripheral that operates in a specific byte ordering mode. In that case, the code should include `<asm/byteorder.h>` and should check whether `__BIG_ENDIAN` or `__LITTLE_ENDIAN` is defined by the header.

You could code a bunch of `#ifdef __LITTLE_ENDIAN` conditionals, but there is a better way. The Linux kernel defines a set of macros that handle conversions between the processor's byte ordering and that of the data you need to store or load in a specific byte order. For example:

```
u32 __cpu_to_le32 (u32);
u32 __le32_to_cpu (u32);
```

These two macros convert a value from whatever the CPU uses to an unsigned, little-endian, 32-bit quantity and back. They work whether your CPU is big-endian

or little-endian, and, for that matter, whether it is a 32-bit processor or not. They return their argument unchanged in cases where there is no work to be done. Use of these macros makes it easy to write portable code without having to use a lot of conditional compilation constructs.

There are dozens of similar routines; you can see the full list in `<linux/byteorder/big_endian.h>` and `<linux/byteorder/little_endian.h>`. After a while, the pattern is not hard to follow. *__be64_to_cpu* converts an unsigned, big-endian, 64-bit value to the internal CPU representation. *__le16_to_cpus*, instead, handles signed, little-endian, 16-bit quantities. When dealing with pointers, you can also use functions like *__cpu_to_le32p*, which take a pointer to the value to be converted rather than the value itself. See the include file for the rest.

Not all Linux versions defined all the macros that deal with byte ordering. In particular, the *linux/byteorder* directory appeared in version 2.1.72 to make order in the various `<asm/byteorder.h>` files and remove duplicate definitions. If you use our *sysdep.h*, you'll be able to use all of the macros available in Linux 2.4 when compiling code for 2.0 or 2.2.

Data Alignment

The last problem worth considering when writing portable code is how to access unaligned data—for example, how to read a four-byte value stored at an address that isn't a multiple of four bytes. PC users often access unaligned data items, but few architectures permit it. Most modern architectures generate an exception every time the program tries unaligned data transfers; data transfer is handled by the exception handler, with a great performance penalty. If you need to access unaligned data, you should use the following macros:

```
#include <asm/unaligned.h>
get_unaligned(ptr);
put_unaligned(val, ptr);
```

These macros are typeless and work for every data item, whether it's one, two, four, or eight bytes long. They are defined with any kernel version.

Another issue related to alignment is portability of data structures across platforms. The same data structure (as defined in the C-language source file) can be compiled differently on different platforms. The compiler arranges structure fields to be aligned according to conventions that differ from platform to platform. At least in theory, the compiler can even reorder structure fields in order to optimize memory usage.*

* Field reordering doesn't happen in currently supported architectures because it could break interoperability with existing code, but a new architecture may define field reordering rules for structures with holes due to alignment restrictions.

In order to write data structures for data items that can be moved across architectures, you should always enforce natural alignment of the data items in addition to standardizing on a specific endianness. *Natural alignment* means storing data items at an address that is a multiple of their size (for instance, 8-byte items go in an address multiple of 8). To enforce natural alignment while preventing the compiler from moving fields around, you should use filler fields that avoid leaving holes in the data structure.

To show how alignment is enforced by the compiler, the *dataalign* program is distributed in the *misc-progs* directory of the sample code, and an equivalent *kdataalign* module is part of *misc-modules*. This is the output of the program on several platforms and the output of the module on the SPARC64:

```
arch  Align:  char  short  int  long   ptr long-long  u8 u16 u32 u64
i386            1      2    4     4     4         4   1   2   4   4
i686            1      2    4     4     4         4   1   2   4   4
alpha           1      2    4     8     8         8   1   2   4   8
armv4l          1      2    4     4     4         4   1   2   4   4
ia64            1      2    4     8     8         8   1   2   4   8
mips            1      2    4     4     4         8   1   2   4   8
ppc             1      2    4     4     4         8   1   2   4   8
sparc           1      2    4     4     4         8   1   2   4   8
sparc64         1      2    4     4     4         8   1   2   4   8

kernel: arch  Align: char short int long  ptr long-long u8 u16 u32 u64
kernel: sparc64        1    2    4    8    8         8       1   2   4   8
```

It's interesting to note that not all platforms align 64-bit values on 64-bit boundaries, so you'll need filler fields to enforce alignment and ensure portability.

Linked Lists

Operating system kernels, like many other programs, often need to maintain lists of data structures. The Linux kernel has, at times, been host to several linked list implementations at the same time. To reduce the amount of duplicated code, the kernel developers have created a standard implementation of circular, doubly-linked lists; others needing to manipulate lists are encouraged to use this facility, introduced in version 2.1.45 of the kernel.

To use the list mechanism, your driver must include the file `<linux/list.h>`. This file defines a simple structure of type `list_head`:

```
struct list_head {
    struct list_head *next, *prev;
};
```

Linked lists used in real code are almost invariably made up of some type of structure, each one describing one entry in the list. To use the Linux list facility in your

code, you need only embed a `list_head` inside the structures that make up the list. If your driver maintains a list of things to do, say, its declaration would look something like this:

```
struct todo_struct {
    struct list_head list;
    int priority; /* driver specific */
    /* ... add other driver-specific fields */
};
```

The head of the list must be a standalone `list_head` structure. List heads must be initialized prior to use with the `INIT_LIST_HEAD` macro. A "things to do" list head could be declared and initialized with:

```
struct list_head todo_list;

INIT_LIST_HEAD(&todo_list);
```

Alternatively, lists can be initialized at compile time as follows:

```
LIST_HEAD(todo_list);
```

Several functions are defined in `<linux/list.h>` that work with lists:

`list_add(struct list_head *new, struct list_head *head);`
: This function adds the `new` entry immediately after the list head—normally at the beginning of the list. It can thus be used to build stacks. Note, however, that the `head` need not be the nominal head of the list; if you pass a `list_head` structure that happens to be in the middle of the list somewhere, the new entry will go immediately after it. Since Linux lists are circular, the head of the list is not generally different from any other entry.

`list_add_tail(struct list_head *new, struct list_head *head);`
: Add a new entry just before the given list head—at the end of the list, in other words. *list_add_tail* can thus be used to build first-in first-out queues.

`list_del(struct list_head *entry);`
: The given entry is removed from the list.

`list_empty(struct list_head *head);`
: Returns a nonzero value if the given list is empty.

`list_splice(struct list_head *list, struct list_head *head);`
: This function joins two lists by inserting `list` immediately after `head`.

The `list_head` structures are good for implementing a list of like structures, but the invoking program is usually more interested in the larger structures that make

up the list as a whole. A macro, *list_entry*, is provided that will map a `list_head` structure pointer back into a pointer to the structure that contains it. It is invoked as follows:

```
list_entry(struct list_head *ptr, type_of_struct, field_name);
```

where `ptr` is a pointer to the `struct list_head` being used, `type_of_struct` is the type of the structure containing the `ptr`, and `field_name` is the name of the list field within the structure. In our `todo_struct` structure from before, the list field is called simply `list`. Thus, we would turn a list entry into its containing structure with a line like this:

```
struct todo_struct *todo_ptr =
    list_entry(listptr, struct todo_struct, list);
```

The *list_entry* macro takes a little getting used to, but is not that hard to use.

The traversal of linked lists is easy: one need only follow the `prev` and `next` pointers. As an example, suppose we want to keep the list of `todo_struct` items sorted in descending priority order. A function to add a new entry would look something like this:

```
void todo_add_entry(struct todo_struct *new)
{
    struct list_head *ptr;
    struct todo_struct *entry;

    for (ptr = todo_list.next; ptr != &todo_list; ptr = ptr->next) {
        entry = list_entry(ptr, struct todo_struct, list);
        if (entry->priority < new->priority) {
            list_add_tail(&new->list, ptr);
            return;
        }
    }
    list_add_tail(&new->list, &todo_struct)
}
```

The `<linux/list.h>` file also defines a macro *list_for_each* that expands to the `for` loop used in this code. As you may suspect, you must be careful when modifying the list while traversing it.

Figure 10-1 shows how the simple `struct list_head` is used to maintain a list of data structures.

Although not all features exported by the *list.h* as it appears in Linux 2.4 are available with older kernels, our *sysdep.h* fills the gap by declaring all macros and functions for use in older kernels.

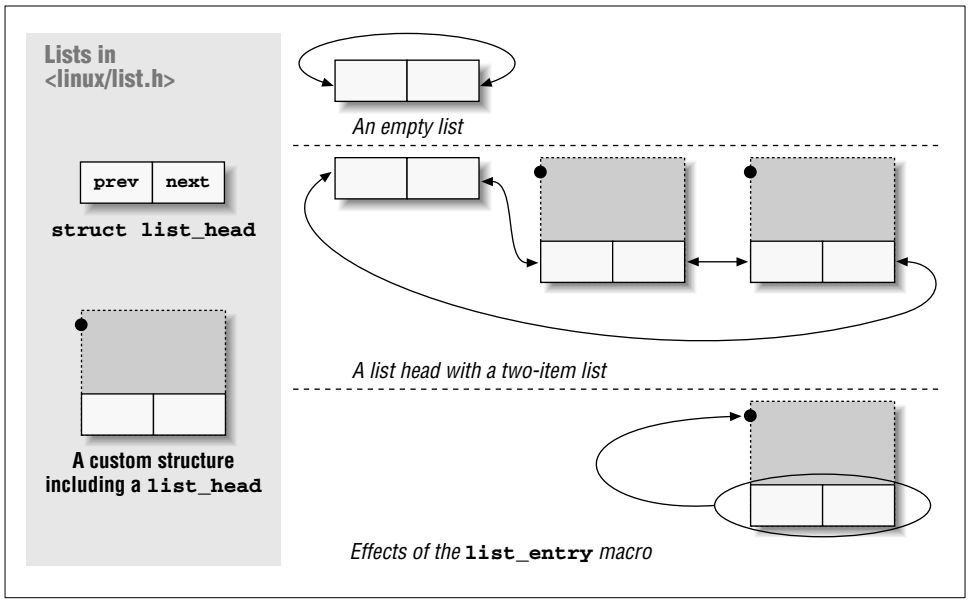


Figure 10-1. The list_head data structure

Quick Reference

The following symbols were introduced in this chapter.

```
#include <linux/types.h>
typedef u8;
typedef u16;
typedef u32;
typedef u64;
```

These types are guaranteed to be 8-, 16-, 32-, and 64-bit unsigned integer values. The equivalent signed types exist as well. In user space, you can refer to the types as `__u8`, `__u16`, and so forth.

```
#include <asm/page.h>
PAGE_SIZE
PAGE_SHIFT
```

These symbols define the number of bytes per page for the current architecture and the number of bits in the page offset (12 for 4-KB pages and 13 for 8-KB pages).

```
#include <asm/byteorder.h>
__LITTLE_ENDIAN
__BIG_ENDIAN
```

Only one of the two symbols is defined, depending on the architecture.

```
#include <asm/byteorder.h>
u32 __cpu_to_le32 (u32);
u32 __le32_to_cpu (u32);
```

Functions for converting between known byte orders and that of the processor. There are more than 60 such functions; see the various files in *include/linux/byteorder/* for a full list and the ways in which they are defined.

```
#include <asm/unaligned.h>
get_unaligned(ptr);
put_unaligned(val, ptr);
```

Some architectures need to protect unaligned data access using these macros. The macros expand to normal pointer dereferencing for architectures that permit you to access unaligned data.

```
#include <linux/list.h>
list_add(struct list_head *new, struct list_head *head);
list_add_tail(struct list_head *new, struct list_head
      *head);
list_del(struct list_head *entry);
list_empty(struct list_head *head);
list_entry(entry, type, member);
list_splice(struct list_head *list, struct list_head *head);
```

Functions for manipulating circular, doubly linked lists.