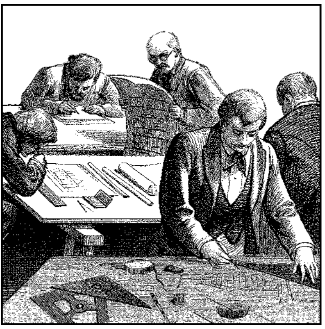
CHAPTER 11

Data Types in the Kernel



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 numerous 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 2-byte filler" or "something representing a 4-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 O'Reilly's FTP site in the directory *misc-progs*. This is a sample run of the program on an i386 system (the last four types shown are introduced in the next section):

```
morgana% misc-progs/datasize
arch   Size:  char  short  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  short  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
x86_64          1     2     4     8     8     8        1   2   4   8
```

It's interesting to note that the SPARC 64 architecture runs with a 32-bit user space, so pointers are 32 bits wide there, 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 often 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, generic memory addresses in the kernel are usually `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.

For what it's worth, the C99 standard defines the `intptr_t` and `uintptr_t` types for an integer variable that can hold a pointer value. These types are almost unused in the 2.6 kernel, however.

Assigning an Explicit Size to Data Items

Sometimes kernel code requires data items of a specific size, perhaps to match predefined binary structures,[*] to communicate with user space, or to align data within structures by inserting "padding" fields (but refer to the section "Data Alignment" 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 */
```

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 support the C99-standard types, such as `uint8_t` and `uint32_t`; if portability is a concern, those types can be used in favor of the Linux-specific variety.

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 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 operating system (OS) he wrote for his own use to become multiplatform; as a result, old structures are sometimes loosely typed.

---

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

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

Interface-Specific Types

Some 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.

Note that, in recent times, relatively few new interface-specific types have been defined. Use of the `typedef` statement has gone out of favor among many kernel developers, who would rather see the real type information used directly in the code, rather than hidden behind a user-defined type. Many older interface-specific types remain in the kernel, however, and they will not be going away anytime soon.

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.

Many `_t` types are defined 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`, that 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 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 9). 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 can 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 1000 jiffies per second. Although this is currently true for the i386 architecture, not every Linux platform runs at this speed. 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`.

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 hardcoding 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 pages that are 4 KB and larger. The macros are defined in *<asm/page.h>*; user-space programs can use the *getpagesize* library function 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. Such a solution, fortunately, has been written by the kernel developers and is called *get_order*:

```
#include <asm/page.h>
int order = get_order(16*1024);
buf = get_free_pages(GFP_KERNEL, order);
```

Remember that the argument to *get_order* must be a power of two.

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), some high-level platforms work the other way (big-endian). 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, or it must communicate with a device that expects a specific order.

The include file *<asm/byteorder.h>* defines either `__BIG_ENDIAN` or `__LITTLE_ENDIAN`, depending on the processor's byte ordering. When dealing with byte ordering issues, 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.

Data Alignment

The last problem worth considering when writing portable code is how to access unaligned data—for example, how to read a 4-byte value stored at an address that isn't a multiple of 4 bytes. i386 users often access unaligned data items, but not all architectures permit it. Many 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.

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 to arrange the fields in unpredictable ways, 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
x86_64          1     2     4     8     8     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 need filler fields to enforce alignment and ensure portability.

Finally, be aware that the compiler may quietly insert padding into structures itself to ensure that every field is aligned for good performance on the target processor. If you are defining a structure that is intended to match a structure expected by a device, this automatic padding may thwart your attempt. The way around this problem is to tell the compiler that the structure must be "packed," with no fillers added. For example, the kernel header file *<linux/edd.h>* defines several data structures used in interfacing with the x86 BIOS, and it includes the following definition:

```
struct {
        u16 id;
        u64 lun;
        u16 reserved1;
        u32 reserved2;
} __attribute__ ((packed)) scsi;
```

Without the `__attribute__ ((packed))`, the `lun` field would be preceded by two filler bytes or six if we compile the structure on a 64-bit platform.

Pointers and Error Values

Many internal kernel functions return a pointer value to the caller. Many of those functions can also fail. In most cases, failure is indicated by returning a `NULL` pointer value. This technique works, but it is unable to communicate the exact nature of the problem. Some interfaces really need to return an actual error code so that the caller can make the right decision based on what actually went wrong.

A number of kernel interfaces return this information by encoding the error code in a pointer value. Such functions must be used with care, since their return value cannot simply be compared against `NULL`. To help in the creation and use of this sort of interface, a small set of functions has been made available (in *<linux/err.h>*).

A function returning a pointer type can return an error value with:

```
void *ERR_PTR(long error);
```

where `error` is the usual negative error code. The caller can use *IS_ERR* to test whether a returned pointer is an error code or not:

```
long IS_ERR(const void *ptr);
```

If you need the actual error code, it can be extracted with:

```
long PTR_ERR(const void *ptr);
```

You should use *PTR_ERR* only on a value for which *IS_ERR* returns a true value; any other value is a valid pointer.

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.

When working with the linked list interface, you should always bear in mind that the list functions perform no locking. If there is a possibility that your driver could attempt to perform concurrent operations on the same list, it is your responsibility to implement a locking scheme. The alternatives (corrupted list structures, data loss, kernel panics) tend to be difficult to diagnose.

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 is usually a standalone `list_head` structure. Figure 11-1 shows how the simple `struct list_head` is used to maintain a list of data structures.

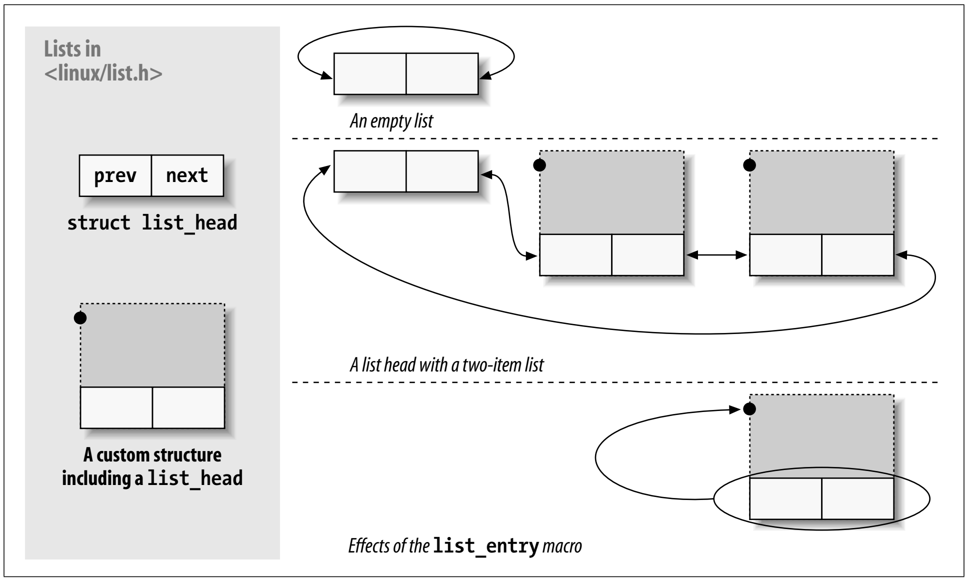


Figure 11-1. The list_head data structure

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:

```
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);`
: Adds the `new` entry immediately after the list head—normally at the beginning of the list. Therefore, it can 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 goes 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);`
: Adds 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);`
`list_del_init(struct list_head *entry);`
: The given entry is removed from the list. If the entry might ever be reinserted into another list, you should use *list_del_init*, which reinitializes the linked list pointers.

`list_move(struct list_head *entry, struct list_head *head);`
`list_move_tail(struct list_head *entry, struct list_head *head);`
: The given `entry` is removed from its current list and added to the beginning of `head`. To put the entry at the end of the new list, use *list_move_tail* instead.

`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);`
: 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 maps 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 such as:

```
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)
}
```

However, as a general rule, it is better to use one of a set of predefined macros for creating loops that iterate through lists. The previous loop, for example, could be coded as:

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

    list_for_each(ptr, &todo_list) {
        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)
}
```

Using the provided macros helps avoid simple programming errors; the developers of these macros have also put some effort into ensuring that they perform well. A few variants exist:

`list_for_each(struct list_head *cursor, struct list_head *list)`
: This macro creates a `for` loop that executes once with `cursor` pointing at each successive entry in the list. Be careful about changing the list while iterating through it.

`list_for_each_prev(struct list_head *cursor, struct list_head *list)`
: This version iterates backward through the list.

`list_for_each_safe(struct list_head *cursor, struct list_head *next, struct list_head *list)`
: If your loop may delete entries in the list, use this version. It simply stores the next entry in the list in `next` at the beginning of the loop, so it does not get confused if the entry pointed to by `cursor` is deleted.

`list_for_each_entry(type *cursor, struct list_head *list, member)`
`list_for_each_entry_safe(type *cursor, type *next, struct list_head *list, member)`
: These macros ease the process of dealing with a list containing a given `type` of structure. Here, `cursor` is a pointer to the containing structure type, and `member` is the name of the `list_head` structure within the containing structure. With these macros, there is no need to put `list_entry` calls inside the loop.

If you look inside *<linux/list.h>*, you see some additional declarations. The `hlist` type is a doubly linked list with a separate, single-pointer list head type; it is often used for creation of hash tables and similar structures. There are also macros for iterating through both types of lists that are intended to work with the read-copy-update mechanism (described in the section "Read-Copy-Update" in Chapter 5). These primitives are unlikely to be useful in device drivers; see the header file if you would like more information on how they work.

Quick Reference

The following symbols were introduced in this chapter:

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

Symbols that 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 that convert 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/err.h>`
`void *ERR_PTR(long error);`
`long PTR_ERR(const void *ptr);`
`long IS_ERR(const void *ptr);`

Functions allow error codes to be returned by functions that return a pointer value.

`#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_del_init(struct list_head *entry);`
`list_empty(struct list_head *head);`
`list_entry(entry, type, member);`
`list_move(struct list_head *entry, struct list_head *head);`
`list_move_tail(struct list_head *entry, struct list_head *head);`
`list_splice(struct list_head *list, struct list_head *head);`

Functions that manipulate circular, doubly linked lists.

```
list_for_each(struct list_head *cursor, struct list_head *list)
list_for_each_prev(struct list_head *cursor, struct list_head *list)
list_for_each_safe(struct list_head *cursor, struct list_head *next, struct
  list_head *list)
list_for_each_entry(type *cursor, struct list_head *list, member)
list_for_each_entry_safe(type *cursor, type *next struct list_head *list,
  member)
```

Convenience macros for iterating through linked lists.