**Linux device driver**

In Linux, a device driver is code that implements a userspace or kernelspace abstraction of a physical device. Examples of device drivers include code that allows user applications to stream data through a 16550 UART, code that configures an Epson S1D13xxx LCD controller chip, and code manages the AT91RM9200's built-in Ethernet controller.

Device drivers come in several different types, depending on what abstraction they provide. Serial port drivers, for example, often implement the character device type. The *framebuffer* driver type is normally used to enable userspace applications to write to an LCD or CRT display. *Ethernet* drivers allow the Linux kernel's TCP/IP protocol stack to send packets over an Ethernet network.

A device driver abstraction is purely that, an abstraction. The underlying hardware associated with an Ethernet device driver may not actually be an Ethernet controller, for example; there may not even be any physical hardware at all! Such is the case for the usbnet driver, which allows a USB Device to communicate with a USB Host as though the two were connected via Ethernet. (The usbnet driver packages Ethernet packets and forwards them to the USB host/device drivers for transmission).

There may be more than one device driver abstraction associated with a particular piece of hardware. A multifunction chip, like the Silicon Motion SM501 Multimedia Controller, will usually have one or more associated framebuffer drivers for its video controllers, a USB Host bus controller driver for its OHCI Host port, an AC97 codec driver for its AC97-Link interface, and a character driver for its serial ports.

**Types**

*Character device driver*

A character device is accessed as a stream of sequential data (like a file). Examples are keyboards, mice

and the serial port. A driver for a character device usually implements at least the open, close, read and write system calls.

*Block device driver*

If a device is not accessed sequentially but rather randomly, it is called a block device. The most common block devices are hard disks, floppies, CD-ROMs and flash memories. Internally, a block device driver works with blocks of data, usually having a size of 4096 bytes. But in contrast to most Unix systems, Linux allows user software to perform I/O operations on an arbitrary number of bytes. Hence, block and character devices differ only in the way

their data is managed internally in the kernel.

*Network device driver*

Network device drivers implement the functionality of sending and receiving data packets of network interfaces. The most important difference between the previous device drivers and network drivers is that network drivers receive packets asynchronously from the outside, whereas the others operate only on requests from the kernel.

**Some doubts**

**USB is which type of driver**

**Audio**

**SCSI**

**PCI**

**Difference between Character and Block device**

A block device would read/write bytes in fixed size blocks, as in disk sectors. Character devices read/write 0 or more bytes, in a stream, such as a TTY or a [keyboard.](http://www.linuxquestions.org/questions/" \t "undefined)

[http://kona.kontera.com/javascript/lib/imgs/grey_loader.gif](http://www.linuxquestions.org/questions/" \t "undefined)

The nature of the device generally dictates how the [device driver](http://www.linuxquestions.org/questions/) is written for it, and you access the device accordingly. Mostly, when accessing devices, you treat then as if they are [files](http://www.linuxquestions.org/questions/): open-read-write-seek-flush-stat-close. You also have ioctl calls to access the device itself (as opposed to the data that it conveys).

According to kernel structures

* The **block\_device\_operations** structure, where the classic character driver uses the **file\_operations** structure
* The **register\_blkdev()** and **unregister\_blkdev()** functions, where the classic character driver uses the

**register\_chrdev()** and **unregister\_chrdev()** functions.

**read/write access to the block device**

* If a user program makes read/write library calls to access a character device, the VFS passes these requests on to the low level read/write functions in the driver.
* If a user program makes read/write library calls to access a block device, the VFS does not pass these requests on to low level read/write functions in the driver. Instead, the block\_read() and block\_write() functions ( see /usr/src/linux/fs/block\_dev.c) are used so that the user interaction is with the buffer not with the driver.

Clearly, if a block device driver does not have a direct read/write interaction with the user program because of the

intervening buffer, then it must provide functions to keep the buffer appropriately up to date. In short, the device

driver must provide low level read/write functionality to interact with the buffer as needed. These low level

functions are not entry points triggered directly by user programs and hence do not belong in the block\_device\_operations struct. Nevertheless, the kernel must know about the block driver's low level read/write

functionality.

If the block\_read() or block\_write() function finds that the buffer is not up to date, it will call the function ll\_rw\_block() which interfaces to the device driver's low level read/write functionality. The ll\_rw\_block() function

can be found in a directory with a somewhat similar name, i.e. /usr.src/linux/drivers/block/ll\_rw\_blk.c. The low

level read/write request may be deferred ('plugged') to allow the system to merge requests which may be adjacent on

the hard disk. Since disk seek and rotational latency are very slow (by the CPU's standards), any merging and sorting possible yield much better efficiency. A given device will have a its own request queues (typically just one), so that the merging and sorting of requests are possibilities that makes sense.

**Major/Minor Number**

Usually the major number identifies the driver and the minor number identifies a device managed by this driver.

We can create major/minor numbers using mknod i.e

**mknod /dev/ttyX c 253 0**

**Memory Allocation in drivers**

**kmalloc**

#include <linux/slab.h>

void \*kmalloc(size\_t size, int flags);

kfree()

The first argument to *kmalloc* is the size of the block to be allocated. The second argument, the allocation flags, is much more interesting, because it controls the behavior of *kmalloc* in a number of ways.

*Flags*

GFP\_KERNEL

The most commonly used flag, GFP\_KERNEL, means that the allocation (internally performed by calling, eventually, *\_\_get\_free\_pages*, which is the source of the GFP\_ prefix) is performed on behalf of a process running in kernel space. In other words, this means that the calling function is executing a system call on behalf of a process.

Using GFP\_KERNEL means that *kmalloc* can put the current process to sleep waiting for a page when called in low-memory situations. A function that allocates memory using GFP\_KERNEL must, therefore, be reentrant and cannot be running in atomic context. While the current process sleeps, the kernel takes proper action to locate some

free memory, either by flushing buffers to disk or by swapping out memory from a user process.

GFP\_ATOMIC

The allocation is high-priority and does not sleep. This is the flag to use in interrupt handlers, bottom halves and other situations where you cannot sleep. When GFP\_ATOMIC is used, *kmalloc* can use even the last free page. If that last page does not exist, however, the allocation fails.

A lot other flags are available like

GFP\_USER- Used to allocate memory for user-space pages; it may sleep.

GFP\_DMA- This flag requests allocation to happen in the DMA-capable memory zone

GFP\_HIGHMEM- This flag indicates that the allocated memory may be located in high memory

Etc..

**vmalloc**

#include <linux/vmalloc.h>

void \*vmalloc(unsigned long size);

void vfree(void \* addr);

Usage: An example of a function in the kernel that uses *vmalloc* is the *create\_module* system call, which uses *vmalloc* to get space for the module being created. Code and data of the module are later copied to the allocated space using *copy\_from\_user*. In this way, the module appears to be loaded into contiguous memory. You can verify, by looking in */proc/kallsyms*, that kernel symbols exported by modules lie in a different memory range from symbols exported by the kernel proper. Memory allocated with *vmalloc* is released by *vfree*, in the same way that *kfree* releases memory allocated by *kmalloc*.

**Difference between kmalloc/vmalloc**

kmalloc allocates physically contiguous memory, memory which pages are laid consecutively in physical RAM. vmalloc allocates memory which is contiguous in kernel virtual memory space (that means pages allocated that way are not contiguous in RAM, but the kernel sees them as one block).

kmalloc is the preffered way, as long as you don't need very big areas. The trouble is, if you want to do DMA from/to some hardware device, you'll need to use kmalloc, and you'll probably need bigger chunk. The solution is to allocate memory as soon as possible, before memory gets fragmented.

vmalloc is often slower than kmalloc, because it may have to remap the buffer space into a virtually contiguous range. kmalloc never remaps, though if not called with GFP\_ATOMIC kmalloc can block.kmalloc is limited in the size of buffer it can provide: 128 KBytes. If you need a really big buffer, you have to use vmalloc or some other mechanism like reserving high memory at boot.

Maximum size of memory that can be allocated by kmalloc is 128k. You want to check in the slab allocator to   
see what size it goes up to.

**Kzalloc**

void \*kzalloc(size\_t size, unsigned int gfp\_flags);

kmalloc returns garbage value after successful allocation so we have to do memset with zero but with kzalloc it will allocate zeros after allocation.

**Insmod/Modprobe**

**insmod** installs a loadable module in the running kernel. **insmod** tries to link a module into the running kernel by resolving all symbols from the kernel's exported symbol table.

**insmod** makes an init\_module system call to load the module into kernel memory. Loading it is the easy part, though. How does the kernel know to use it? The answer is that the init\_module system call invokes the module’s initialization routine right after it loads the module. **insmod** passes to init\_module the address of the subroutine in the module named init\_module as its initialization routine. The module set up an init\_module to call a kernel function that registers the subroutines that the module contains. For example, a character device driver's init\_module subroutine might call the kernel's register\_chrdev subroutine, passing the major and minor number of the device it intends to drive and the address of its own "open" routine among the arguments. register\_chrdev records in base kernel tables that when the kernel wants to open that particular device, it should call the open routine in our module.

In 2.4, **insmod** functions as a relocating linker/loader. The module object file contains an external reference to the symbol register\_chrdev. **insmod** does a query\_module system call to find out the addresses of various symbols that the existing kernel exports. register\_chrdev is among these. query\_module returns the address for which register\_chrdev stands and **insmod** patches that into the module where the module refers to register\_chrdev.

**Difference between modprobe**

Insmod will try to load only the module you specify. If it depends on other modules, the insmod will fail. Modprobe, on the other hand, checks the module's dependancies and loads any modules that it needs before it

tries to insert the module you specified.

insmod doesn't have the capability to resolve dependency issues, that is if the installation of a module requires some pre-requisites insmod is not able to resolve that. But modprobe can resolve dependency issues. While installing a module using modprobe, it first determines whether there exists any dependencies by checking the file /lib/modules/<kernel-version>/modules.dep. This file contains entries for a module and its corresponding dependencies. modules.dep files is generated by the command depmod.

*Note*: **insmod** uses the .modinfo section to read the kernel release number for which the module was built. It describes the form of the module's parameters. **insmod** uses this information to format the parameters you supply on the **insmod** command line into data structure initial values, which **insmod** inserts into the module as it loads it.

**Module\_init/Module\_exit**

Module\_init/module\_exit are defined as macros in include/linux/init.h as

#define module\_init(x) \_\_initcall(x);

#define module\_exit(x) \_\_exitcall(x);

which through linker magic ensures that the function is called on boot.

static int \_\_init test\_init(void)

{

return platform\_driver\_register(&test\_driver);

}

module\_init(test\_init);

The test\_init function can be considered as the entry point into the driver – of particular interest is the \_\_init macro and the static declaration. The \_\_init macro is used to describe the function as only being required during initialisation time. Once initialisation is performed the kernel will remove this function and release its memory. The module\_init macro is used to tell the kernel where the initialisation entry point to the module lives, i.e. what function to call at ‘start of day’. In a typical driver you will often see many initialisation functions marked with the \_\_init macro which are used for initialisation, and a single module\_init declaration.

With the above code in place, at an appropriate time during start-up, the kernel will call the test\_init function and once it has been executed it’s memory will be released. You can see this during the output from kernel boot (e.g. dmesg), for example an x86 machine may print the following:

*Freeing unused kernel memory: 386k freed*

Which means that 386k of memory that previously contained initialisation code and data has now been freed?

#define module\_init(x) \_\_initcall(x);

#define \_\_initcall(fn) device\_initcall(fn)

#define device\_initcall \_\_define\_initcall("6", fn, 6)

#define \_\_define\_initcall(level, fn, id) \

static initcall\_t \_\_initcall\_##fn##id \_\_used \

\_\_attribute\_\_ ((\_\_section\_\_(".initcall" level ".init"))) = fn

So another load of macros that result in yet another GCC attribute!

#define module\_init(x) static initcall\_t \_\_initcall\_x6 \_\_used \

\_\_attribute\_\_ ((\_\_section(".initcall6.init"))) = x;

So module\_init in the context of a built-in driver results in declaring a function pointer with a unique name to our point of entry. In addition the macro ensures the function pointer is located in a special section of the ELF – we’ll see why shortly.So at present we have ensured all our initialisation code and data are stored in the .init.text section, and that each module has a function pointer for it’s point of entry – which has a unique name and is also stored in a special section of the resulting ELF. In addition during link time the include/asm-generic/vmlinux.lds.h and arch/\*/kernel/vmlinux.lds.S scripts ensure that some labels/symbols surround the start and end of these sections. I.e. \_\_early\_initcall\_end and \_\_initcall\_end mark the start and end of the function pointers and \_\_init\_begin and \_\_init\_end mark the start and end of the .init.text section.

Finally we are in place to see how these functions get called and how they are eventually freed. During kernel start up a function called do\_initcalls in init/main.c is called, this is shown below.

static void \_\_init do\_initcalls(void)

{

initcall\_t \*call;

for (call = \_\_early\_initcall\_end; call < \_\_initcall\_end; call++)

do\_one\_initcall(\*call);

The purpose of this loop is to execute each of the init functions as set up by the module\_init macros. This is achieved with a simple for loop and a function pointer. Initially the function pointer is pointed to the label at the start of our function pointers ELF section, and is incremented (by the size of a function pointer (sizeof(initcall\_t \*)) until the end of the ELF section is reached. For each step the pointer is invoked and the init function is thus executed.

Once initialisation is complete, a function found in the architecture specific code named free\_initmem is used to release the memory pages taken up by the initialisation functions and data. The exact nature of the function depends on the architecture.

So in a nutshell the kernel makes clever use of [GCC attributes](http://gcc.gnu.org/onlinedocs/gcc/Function-Attributes.html) to ensure that initialisation functions and pointers to them are stored in unique sections of the ELF. Initialisation code at kernel start up then iterates through these function pointers and executes them in turn. Finally once all init code has been executed the entire ELF section (.init.text) is freed for use!

**Accessing user space memory from kernel**

Sometimes the kernel has to access userspace memory. This memory may not be directly accessible even by the kernel, so there are some specialized functions to access that part of memory.

*access\_ok(type, addr, size)*

Quick test to check for invalid addresses. *type* may be VERIFY\_READ or VERIFY\_WRITE.

*get\_user(var, addr)*

read userspace memory at *addr* and store into variable *var*.

*put\_user(var, addr)*

write variable *var* into userspace memory at *addr*

*copy\_from\_user(to, from, n)*

copy *n* bytes from userspace address *from* to kernel address *to*.

*copy\_to\_user(to, from, n)*

copy *n* bytes from kernel address *from* to userspace address *to*.

Be careful, the return values have different meanings for historical reasons: copy\_to/from\_user returns the number of bytes that could **not** be copied; the calling function is expected to return -EFAULT in this case. get/put\_user already return -EFAULT or zero and access\_ok returns 0 if the given address is invalid. You have to check the return value of get/put\_user and copy\_to/from\_user even when access\_ok saysthat the address is ok. The actual access to the memory could fail nevertheless.

**Note:** To read a simple variable from user space, you use the get\_user function. This function is used for simple types such as char and int, but larger data types like structures must use the copy\_from\_user function, instead.Same thing is applicable to put\_user/copy\_to\_user

Access\_ok: You use the access\_ok function to check the validity of the pointer in user space that you intend to access. The caller provides the pointer, which refers to the start of the data block, the size of the block, and the type of access (whether the area is intended to be read or written).

**Usage**

**If one GPIO pin has to be toggled the definitely we are going to use get\_user/put\_user function in kernel through ioctls i.e**

**In application**

**fd = open(“/dev/X”,O\_RDWR);**

**int value = 1;**

**ioctl(fd,X\_CMD,value);**

**ioctl(fd,X\_CMD,value);**

**In kernel,in the ioctl fops and wrt to the command being sent in ioctl parameter,i.e**

**static int X\_ioctl(struct inode \*inode, struct file \*file, unsigned int cmd, unsigned long arg){**

**void \_\_user \*argp = (void \_\_user \*)arg;**

**int \_\_user \*p = argp;**

**int value;**

**switch(cmd){**

**case X\_CMD:**

**if (get\_user(value, p))**

**return -EFAULT;**

**at91\_set\_gpio\_value(AT91\_PIN\_PB22, value);**

**}**

**}**

**For put\_user for e.g in watchdog timer you want to know the maximum time then again in watchdog driver ioctls,lets assume time = 16,then you call the ioctl and get this time from application**

**case WDIOC\_GETTIMEOUT:**

**return put\_user(timeo, p);//p is same above the argument**

**In application,**

**ioctl(watchdog\_fd,WDIOC\_GETTIMEOUT,empty\_variable);**

The copy\_from\_user and copy\_to\_user are basically used in read and write fops of drivers to get the buffer data from user and put the data in user buffer respectively.

**IOCTL**

In computing, ioctl ( or "i-o-control") is a system call for device-specific operations and other operations which cannot be expressed by regular system calls. It takes a parameter specifying a request code; the effect of a call depends completely on the request code. ...

Device files are supposed to represent physical devices. Most physical devices are used for output as well as input, so there has to be some mechanism for device drivers in the kernel to get the output to send to the device from processes. This is done by opening the device file for output and writing to it, just like writing to a file. In the following example, this is implemented by device\_write.

This is not always enough. Imagine you had a serial port connected to a modem (even if you have an internal modem, it is still implemented from the CPU's perspective as a serial port connected to a modem, so you don't have to tax your imagination too hard). The natural thing to do would be to use the device file to write things to the modem (either modem commands or data to be sent through the phone line) and read things from the modem (either responses for commands or the data received through the phone line). However, this leaves open the question of what to do when you need to talk to the serial port itself, for example to send the rate at which data is sent and received.

The answer in Unix is to use a special function called ioctl (short for Input Output ConTroL). Every device can have its own ioctl commands, which can be read ioctl's (to send information from a process to the kernel), write ioctl's (to return information to a process),both or neither. The ioctl function is called with three parameters: the file descriptor of the appropriate device file, the ioctl number, and a parameter, which is of type long so you can use a cast to use it to pass anything.

The ioctl number encodes the major device number, the type of the ioctl, the command, and the type of the parameter. This ioctl number is usually created by a macro call (\_IO, \_IOR, \_IOW or \_IOWR --- depending on the type) in a header file. This header file should then be included both by the programs which will use ioctl (so they can generate the appropriate ioctl's) and by the kernel module.

You can refer the above example source.

If you are adding new ioctl's to the kernel, you should use the \_IO macros defined in <linux/ioctl.h>:

\_IO an ioctl with no parameters

\_IOW an ioctl with write parameters (copy\_from\_user)

\_IOR an ioctl with read parameters (copy\_to\_user)

\_IOWR an ioctl with both write and read parameters.

**What is magic number in ioctl:** To help programmers create unique ioctl command codes, these codes have been split up into several bitfields. The first versions of Linux used 16-bit numbers: the top eight were the "magic" numbers associated with the device, and the bottom eight were a sequential number, unique within the device.

For e.g #define FBIOGET\_VSCREENINFO 0x4600(0x46 is ‘F’ which is unique to the device)

**Locking Mechanisms**

**Race condition**

A **race condition** or **race** [**hazard**](http://en.wikipedia.org/wiki/Hazard_(logic)) is a flaw in an electronic [system](http://en.wikipedia.org/wiki/System) or process whereby the output and/or result of the process is unexpectedly and critically dependent on the sequence or [timing](http://en.wikipedia.org/wiki/Timing) of other events. The term originates with the idea of two [signals](http://en.wikipedia.org/wiki/Signal_(information_theory)) racing each other to influence the [output](http://en.wikipedia.org/wiki/Output) first.In the other way,if two process tries to access the same resource or memory allocation for the same variable or trying the access the same memory,it leads to Race condition.

**Locking mechanisms**

1. Mutex
2. Semaphore
3. spin locks

**Semaphore**

**#include <asm/semaphore.h>**

Semaphores can be used to synchronize processes. A Semaphore is an integer variable which can be

increased and decreased. When a process tries to decrese the value of the semaphore below zero, it is

blocked until it is possible to decrease the semaphore without making it negative (i.e. until some other

process increases the semaphore).

Unlike spinlocks, the blocked process is put to sleep instead of trying over and over again. As Interrupt

handlers are not allowed to sleep it is not possible to use semaphores there.

sema\_init(struct semaphore \*sem, value)

initialize semaphore.

up(struct semaphore \*sem)

Increase semaphore (*V(sem)*)

down(struct semaphore \*sem)

decrease semaphore, wait if it would become negative (*P(sem)*)

int down\_interruptible(struct semaphore \*sem);

*down* decrements the value of the semaphore and waits as long as need be. *down\_*

*interruptible* does the same, but the operation is interruptible. The interruptible version

is almost always the one you will want; it allows a user-space process that is

waiting on a semaphore to be interrupted by the user.

A critical section can be implemented by initializing the semaphore to 1 and surrounding the critical

section with down() and up() calls. It is advisable to use the interruptible version if the process can block

for a long time.

ret = down\_interruptible(&sem);

if (ret) goto out;

/\* critical section \*/

up(&sem);

ret = 0;

out:gone

**Mutex**

Mutex is same as semaphore. however, semaphores are used in a mutex mode. To make this common

case a little easier, the kernel has provided a set of helper functions and macros.

Thus, a mutex can be declared and initialized with one of the following:

DECLARE\_MUTEX(name);

DECLARE\_MUTEX\_LOCKED(name);

Here, the result is a semaphore variable (called name) that is initialized to 1 (with DECLARE\_MUTEX) or 0 (with DECLARE\_MUTEX\_LOCKED). In the latter case, the mutex starts out in a locked state; it will have to be explicitly unlocked before any thread will beallowed access.

If the mutex must be initialized at runtime (which is the case if it is allocated dynamically, for example), use one of the following:

void init\_MUTEX(struct semaphore \*sem);

void init\_MUTEX\_LOCKED(struct semaphore \*sem);

In the Linux world, the *P* function is called *down*—or some variation of that name. Here, “down” refers to the fact that the function decrements the value of the semaphore and, perhaps after putting the caller to sleep for a while to wait for the semaphore to become available, grants access to the protected resources.

**Spin locks**

Semaphores are a useful tool for mutual exclusion, but they are not the only such tool provided by the kernel. Instead, most locking is implemented with a mechanism called a *spinlock*. Unlike semaphores, spinlocks may be used in code that cannot sleep, such as interrupt handlers. When properly used, spinlocks offer higher performance than semaphores in general. They do, however, bring a different set of constraints on their use. Spinlocks are simple in concept. A spinlock is a mutual exclusion device that can have only two values: “locked” and “unlocked.” It is usually implemented as a single bit in an integer value. Code wishing to take out a particular lock tests the relevant bit. If the lock is available, the “locked” bit is set and the code continues into the critical section. If, instead, the lock has been taken by somebody else, the code goes into a tight loop where it repeatedly checks the lock until it becomes available. This loop is the “spin” part of a spinlock.

The required include file for the spinlock primitives is *<linux/spinlock.h>*

This initialization may be done at compile time as follows:

spinlock\_t my\_lock = SPIN\_LOCK\_UNLOCKED;

or at runtime with:

void spin\_lock\_init(spinlock\_t \*lock);

Before entering a critical section, your code must obtain the requisite lock with:

void spin\_lock(spinlock\_t \*lock);

Once you call *spin\_lock*, you will spin until the lock becomes available. To release a lock that you have obtained, pass it to:

void spin\_unlock(spinlock\_t \*lock);

Imagine for a moment that your driver acquires a spinlockand goes about its business within its critical section. Somewhere in the middle, your driver loses the processor. Perhaps it has called a function (*copy\_from\_user*, say) that puts the process to sleep. Or, perhaps, kernel preemption kicks in, and a higher-priority process pushes

your code aside. Your code is now holding a lockthat it will not release any time in the foreseeable future. If some other thread tries to obtain the same lock, it will, in the best case, wait (spinning in the processor) for a very long time. In the worst case, the system could deadlock entirely.

There are actually four functions that can lock a spinlock:

void spin\_lock(spinlock\_t \*lock);

void spin\_lock\_irqsave(spinlock\_t \*lock, unsigned long flags);

void spin\_lock\_irq(spinlock\_t \*lock);

void spin\_lock\_bh(spinlock\_t \*lock)

*spin\_lock\_irqsave* disables interrupts (on the local processor only) before taking the spinlock; the previous interrupt state is stored in flags. If you are absolutely sure nothing else might have already disabled interrupts on your processor (or, in other words, you are sure that you should enable interrupts when you release your spinlock), you can use *spin\_lock\_irq* instead and not have to keep track of the flags. Finally, *spin\_lock\_bh* disables software interrupts before taking the lock, but leaves hardware interrupts enabled.

There are also four ways to release a spinlock; the one you use must correspond to the function you used to take the lock:

void spin\_unlock(spinlock\_t \*lock);

void spin\_unlock\_irqrestore(spinlock\_t \*lock, unsigned long flags);

void spin\_unlock\_irq(spinlock\_t \*lock);

void spin\_unlock\_bh(spinlock\_t \*lock);

Each *spin\_unlock* variant undoes the workperformed by the corresponding *spin\_lock* function. The flags argument passed to *spin\_unlock\_irqrestore* must be the samevariable passed to *spin\_lock\_irqsave*. You must also call *spin\_lock\_irqsave* and *spin\_ unlock\_irqrestore* in the same function; otherwise, your code may breakon somearchitectures.