CHAPTER SIX

FLOW OF TIME

At this point, we know the basics of how to write a full-featured char module. Real-world drivers, however, need to do more than implement the necessary operations; they have to deal with issues such as timing, memory management, hardware access, and more. Fortunately, the kernel makes a number of facilities available to ease the task of the driver writer. In the next few chapters we'll fill in information on some of the kernel resources that are available, starting with how timing issues are addressed. Dealing with time involves the following, in order of increasing complexity:

- Understanding kernel timing
- Knowing the current time
- Delaying operation for a specified amount of time
- Scheduling asynchronous functions to happen after a specified time lapse

Time Intervals in the Kernel

The first point we need to cover is the timer interrupt, which is the mechanism the kernel uses to keep track of time intervals. Interrupts are asynchronous events that are usually fired by external hardware; the CPU is interrupted in its current activity and executes special code (the Interrupt Service Routine, or ISR) to serve the interrupt. Interrupts and ISR implementation issues are covered in Chapter 9.

Timer interrupts are generated by the system's timing hardware at regular intervals; this interval is set by the kernel according to the value of `HZ`, which is an

architecture-dependent value defined in `<linux/param.h>`. Current Linux versions define `HZ` to be 100 for most platforms, but some platforms use 1024, and the IA-64 simulator uses 20. Despite what your preferred platform uses, no driver writer should count on any specific value of `HZ`.

Every time a timer interrupt occurs, the value of the variable `jiffies` is incremented. `jiffies` is initialized to 0 when the system boots, and is thus the number of clock ticks since the computer was turned on. It is declared in `<linux/sched.h>` as `unsigned long volatile`, and will possibly overflow after a long time of continuous system operation (but no platform features jiffy overflow in less than 16 months of uptime). Much effort has gone into ensuring that the kernel operates properly when `jiffies` overflows. Driver writers do not normally have to worry about `jiffies` overflows, but it is good to be aware of the possibility.

It is possible to change the value of `HZ` for those who want systems with a different clock interrupt frequency. Some people using Linux for hard real-time tasks have been known to raise the value of `HZ` to get better response times; they are willing to pay the overhead of the extra timer interrupts to achieve their goals. All in all, however, the best approach to the timer interrupt is to keep the default value for `HZ`, by virtue of our complete trust in the kernel developers, who have certainly chosen the best value.

Processor-Specific Registers

If you need to measure very short time intervals or you need extremely high precision in your figures, you can resort to platform-dependent resources, selecting precision over portability.

Most modern CPUs include a high-resolution counter that is incremented every clock cycle; this counter may be used to measure time intervals precisely. Given the inherent unpredictability of instruction timing on most systems (due to instruction scheduling, branch prediction, and cache memory), this clock counter is the only reliable way to carry out small-scale timekeeping tasks. In response to the extremely high speed of modern processors, the pressing demand for empirical performance figures, and the intrinsic unpredictability of instruction timing in CPU designs caused by the various levels of cache memories, CPU manufacturers introduced a way to count clock cycles as an easy and reliable way to measure time lapses. Most modern processors thus include a counter register that is steadily incremented once at each clock cycle.

The details differ from platform to platform: the register may or may not be readable from user space, it may or may not be writable, and it may be 64 or 32 bits wide—in the latter case you must be prepared to handle overflows. Whether or not the register can be zeroed, we strongly discourage resetting it, even when

hardware permits. Since you can always measure differences using unsigned variables, you can get the work done without claiming exclusive ownership of the register by modifying its current value.

The most renowned counter register is the TSC (timestamp counter), introduced in x86 processors with the Pentium and present in all CPU designs ever since. It is a 64-bit register that counts CPU clock cycles; it can be read from both kernel space and user space.

After including `<asm/msr.h>` (for "machine-specific registers"), you can use one of these macros:

```
rdtsc(low,high);
rdtscl(low);
```

The former atomically reads the 64-bit value into two 32-bit variables; the latter reads the low half of the register into a 32-bit variable and is sufficient in most cases. For example, a 500-MHz system will overflow a 32-bit counter once every 8.5 seconds; you won't need to access the whole register if the time lapse you are benchmarking reliably takes less time.

These lines, for example, measure the execution of the instruction itself:

```
unsigned long ini, end;
rdtscl(ini); rdtscl(end);
printk("time lapse: %li\n", end - ini);
```

Some of the other platforms offer similar functionalities, and kernel headers offer an architecture-independent function that you can use instead of *rdtsc*. It is called *get_cycles*, and was introduced during 2.1 development. Its prototype is

```
#include <linux/timex.h>
cycles_t get_cycles(void);
```

The function is defined for every platform, and it always returns 0 on the platforms that have no cycle-counter register. The `cycles_t` type is an appropriate unsigned type that can fit in a CPU register. The choice to fit the value in a single register means, for example, that only the lower 32 bits of the Pentium cycle counter are returned by *get_cycles*. The choice is a sensible one because it avoids the problems with multiregister operations while not preventing most common uses of the counter—namely, measuring short time lapses.

Despite the availability of an architecture-independent function, we'd like to take the chance to show an example of inline assembly code. To this aim, we'll implement a *rdtscl* function for MIPS processors that works in the same way as the x86 one.

We'll base the example on MIPS because most MIPS processors feature a 32-bit counter as register 9 of their internal "coprocessor 0." To access the register, only

readable from kernel space, you can define the following macro that executes a "move from coprocessor 0" assembly instruction:*

```
#define rdtscl(dest) \
   __asm__ __volatile__("mfc0 %0,$9; nop" : "=r" (dest))
```

With this macro in place, the MIPS processor can execute the same code shown earlier for the x86.

What's interesting with *gcc* inline assembly is that allocation of general-purpose registers is left to the compiler. The macro just shown uses `%0` as a placeholder for "argument 0," which is later specified as "any register (`r`) used as output (`=`)." The macro also states that the output register must correspond to the C expression `dest`. The syntax for inline assembly is very powerful but somewhat complex, especially for architectures that have constraints on what each register can do (namely, the x86 family). The complete syntax is described in the *gcc* documentation, usually available in the *info* documentation tree.

The short C-code fragment shown in this section has been run on a K7-class x86 processor and a MIPS VR4181 (using the macro just described). The former reported a time lapse of 11 clock ticks, and the latter just 2 clock ticks. The small figure was expected, since RISC processors usually execute one instruction per clock cycle.

Knowing the Current Time

Kernel code can always retrieve the current time by looking at the value of `jiffies`. Usually, the fact that the value represents only the time since the last boot is not relevant to the driver, because its life is limited to the system uptime. Drivers can use the current value of `jiffies` to calculate time intervals across events (for example, to tell double clicks from single clicks in input device drivers). In short, looking at `jiffies` is almost always sufficient when you need to measure time intervals, and if you need very sharp measures for short time lapses, processor-specific registers come to the rescue.

It's quite unlikely that a driver will ever need to know the wall-clock time, since this knowledge is usually needed only by user programs such as *cron* and *at*. If such a capability is needed, it will be a particular case of device usage, and the driver can be correctly instructed by a user program, which can easily do the con-

* The trailing *nop* instruction is required to prevent the compiler from accessing the target register in the instruction immediately following *mfc0*. This kind of interlock is typical of RISC processors, and the compiler can still schedule useful instructions in the delay slots. In this case we use *nop* because inline assembly is a black box for the compiler and no optimization can be performed.

version from wall-clock time to the system clock. Dealing directly with wall-clock time in a driver is often a sign that policy is being implemented, and should thus be looked at closely.

If your driver really needs the current time, the *do_gettimeofday* function comes to the rescue. This function doesn't tell the current day of the week or anything like that; rather, it fills a `struct timeval` pointer—the same as used in the *gettimeofday* system call—with the usual seconds and microseconds values. The prototype for *do_gettimeofday* is:

```
#include <linux/time.h>
void do_gettimeofday(struct timeval *tv);
```

The source states that *do_gettimeofday* has "near microsecond resolution" for many architectures. The precision does vary from one architecture to another, however, and can be less in older kernels. The current time is also available (though with less precision) from the `xtime` variable (a `struct timeval`); however, direct use of this variable is discouraged because you can't atomically access both the `timeval` fields `tv_sec` and `tv_usec` unless you disable interrupts. As of the 2.2 kernel, a quick and safe way of getting the time quickly, possibly with less precision, is to call *get_fast_time*:

```
void get_fast_time(struct timeval *tv);
```

Code for reading the current time is available within the *jit* ("Just In Time") module in the source files provided on the O'Reilly FTP site. *jit* creates a file called */proc/currentime*, which returns three things in ASCII when read:

- The current time as returned by *do_gettimeofday*
- The current time as found in `xtime`
- The current `jiffies` value

We chose to use a dynamic */proc* file because it requires less module code—it's not worth creating a whole device just to return three lines of text.

If you use *cat* to read the file multiple times in less than a timer tick, you'll see the difference between `xtime` and *do_gettimeofday*, reflecting the fact that `xtime` is updated less frequently:

```
morgana% cd /proc; cat currentime currentime currentime
gettime: 846157215.937221
xtime:   846157215.931188
jiffies: 1308094
gettime: 846157215.939950
xtime:   846157215.931188
jiffies: 1308094
gettime: 846157215.942465
xtime:   846157215.941188
jiffies: 1308095
```

Delaying Execution

Device drivers often need to delay the execution of a particular piece of code for a period of time—usually to allow the hardware to accomplish some task. In this section we cover a number of different techniques for achieving delays. The circumstances of each situation determine which technique is best to use; we'll go over them all and point out the advantages and disadvantages of each.

One important thing to consider is whether the length of the needed delay is longer than one clock tick. Longer delays can make use of the system clock; shorter delays typically must be implemented with software loops.

Long Delays

If you want to delay execution by a multiple of the clock tick or you don't require strict precision (for example, if you want to delay an integer number of seconds), the easiest implementation (and the most braindead) is the following, also known as *busy waiting*:

```
unsigned long j = jiffies + jit_delay * HZ;

while (jiffies < j)
    /* nothing */;
```

This kind of implementation should definitely be avoided. We show it here because on occasion you might want to run this code to understand better the internals of other code.

So let's look at how this code works. The loop is guaranteed to work because `jiffies` is declared as `volatile` by the kernel headers and therefore is reread any time some C code accesses it. Though "correct," this busy loop completely locks the processor for the duration of the delay; the scheduler never interrupts a process that is running in kernel space. Still worse, if interrupts happen to be disabled when you enter the loop, `jiffies` won't be updated, and the `while` condition remains true forever. You'll be forced to hit the big red button.

This implementation of delaying code is available, like the following ones, in the *jit* module. The */proc/jit** files created by the module delay a whole second every time they are read. If you want to test the busy wait code, you can read */proc/jitbusy*, which busy-loops for one second whenever its *read* method is called; a command such as *dd if=/proc/jitbusy bs=1* delays one second each time it reads a character.

As you may suspect, reading */proc/jitbusy* is terrible for system performance, because the computer can run other processes only once a second.

A better solution that allows other processes to run during the time interval is the following, although it can't be used in hard real-time tasks or other time-critical situations.

```
while (jiffies < j)
    schedule();
```

The variable `j` in this example and the following ones is the value of `jiffies` at the expiration of the delay and is always calculated as just shown for busy waiting.

This loop (which can be tested by reading */proc/jitsched*) still isn't optimal. The system can schedule other tasks; the current process does nothing but release the CPU, but it remains in the run queue. If it is the only runnable process, it will actually run (it calls the scheduler, which selects the same process, which calls the scheduler, which . . .). In other words, the load of the machine (the average number of running processes) will be at least one, and the idle task (process number 0, also called *swapper* for historical reasons) will never run. Though this issue may seem irrelevant, running the idle task when the computer is idle relieves the processor's workload, decreasing its temperature and increasing its lifetime, as well as the duration of the batteries if the computer happens to be your laptop. Moreover, since the process is actually executing during the delay, it will be accounted for all the time it consumes. You can see this by running *time cat /proc/jitsched*.

If, instead, the system is very busy, the driver could end up waiting rather longer than expected. Once a process releases the processor with *schedule*, there are no guarantees that it will get it back anytime soon. If there is an upper bound on the acceptable delay time, calling *schedule* in this manner is not a safe solution to the driver's needs.

Despite its drawbacks, the previous loop can provide a quick and dirty way to monitor the workings of a driver. If a bug in your module locks the system solid, adding a small delay after each debugging *printk* statement ensures that every message you print before the processor hits your nasty bug reaches the system log before the system locks. Without such delays, the messages are correctly printed to the memory buffer, but the system locks before *klogd* can do its job.

The best way to implement a delay, however, is to ask the kernel to do it for you. There are two ways of setting up short-term timeouts, depending on whether your driver is waiting for other events or not.

If your driver uses a wait queue to wait for some other event, but you also want to be sure it runs within a certain period of time, it can use the timeout versions of the sleep functions, as shown in "Going to Sleep and Awakening" in Chapter 5:

```
sleep_on_timeout(wait_queue_head_t *q, unsigned long timeout);
interruptible_sleep_on_timeout(wait_queue_head_t *q,
                               unsigned long timeout);
```

Both versions will sleep on the given wait queue, but will return within the timeout period (in jiffies) in any case. They thus implement a bounded sleep that will

not go on forever. Note that the timeout value represents the number of jiffies to wait, not an absolute time value. Delaying in this manner can be seen in the implementation of */proc/jitqueue*:

```
wait_queue_head_t wait;

init_waitqueue_head (&wait);
interruptible_sleep_on_timeout(&wait, jit_delay*HZ);
```

In a normal driver, execution could be resumed in either of two ways: somebody calls *wake_up* on the wait queue, or the timeout expires. In this particular implementation, nobody will ever call *wake_up* on the wait queue (after all, no other code even knows about it), so the process will always wake up when the timeout expires. That is a perfectly valid implementation, but, if there are no other events of interest to your driver, delays can be achieved in a more straightforward manner with *schedule_timeout*:

```
set_current_state(TASK_INTERRUPTIBLE);
schedule_timeout (jit_delay*HZ);
```

The previous line (for */proc/jitself*) causes the process to sleep until the given time has passed. *schedule_timeout*, too, expects a time offset, not an absolute number of jiffies. Once again, it is worth noting that an extra time interval could pass between the expiration of the timeout and when your process is actually scheduled to execute.

Short Delays

Sometimes a real driver needs to calculate very short delays in order to synchronize with the hardware. In this case, using the `jiffies` value is definitely not the solution.

The kernel functions *udelay* and *mdelay* serve this purpose.* Their prototypes are

```
#include <linux/delay.h>
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

The functions are compiled inline on most supported architectures. The former uses a software loop to delay execution for the required number of microseconds, and the latter is a loop around *udelay*, provided for the convenience of the programmer. The *udelay* function is where the BogoMips value is used: its loop is based on the integer value `loops_per_second`, which in turn is the result of the BogoMips calculation performed at boot time.

The *udelay* call should be called only for short time lapses because the precision of `loops_per_second` is only eight bits, and noticeable errors accumulate when

* The `u` in *udelay* represents the Greek letter mu and stands for *micro*.

calculating long delays. Even though the maximum allowable delay is nearly one second (since calculations overflow for longer delays), the suggested maximum value for *udelay* is 1000 microseconds (one millisecond). The function *mdelay* helps in cases where the delay must be longer than one millisecond.

It's also important to remember that *udelay* is a busy-waiting function (and thus *mdelay* is too); other tasks can't be run during the time lapse. You must therefore be very careful, especially with *mdelay*, and avoid using it unless there's no other way to meet your goal.

Currently, support for delays longer than a few microseconds and shorter than a timer tick is very inefficient. This is not usually an issue, because delays need to be just long enough to be noticed by humans or by the hardware. One hundredth of a second is a suitable precision for human-related time intervals, while one millisecond is a long enough delay for hardware activities.

Although *mdelay* is not available in Linux 2.0, *sysdep.h* fills the gap.

Task Queues

One feature many drivers need is the ability to schedule execution of some tasks at a later time without resorting to interrupts. Linux offers three different interfaces for this purpose: task queues, tasklets (as of kernel 2.3.43), and kernel timers. Task queues and tasklets provide a flexible utility for scheduling execution at a later time, with various meanings for "later"; they are most useful when writing interrupt handlers, and we'll see them again in "Tasklets and Bottom-Half Processing," in Chapter 9. Kernel timers are used to schedule a task to run at a specific time in the future and are dealt with in "Kernel Timers," later in this chapter.

A typical situation in which you might use task queues or tasklets is to manage hardware that cannot generate interrupts but still allows blocking read. You need to poll the device, while taking care not to burden the CPU with unnecessary operations. Waking the reading process at fixed time intervals (for example, using `current->timeout`) isn't a suitable approach, because each poll would require two context switches (one to run the polling code in the reading process, and one to return to a process that has real work to do), and often a suitable polling mechanism can be implemented only outside of a process's context.

A similar problem is giving timely input to a simple hardware device. For example, you might need to feed steps to a stepper motor that is directly connected to the parallel port—the motor needs to be moved by single steps on a timely basis. In this case, the controlling process talks to your device driver to dispatch a movement, but the actual movement should be performed step by step at regular intervals after returning from *write*.

The preferred way to perform such floating operations quickly is to register a task for later execution. The kernel supports *task queues*, where tasks accumulate to be "consumed" when the queue is run. You can declare your own task queue and trigger it at will, or you can register your tasks in predefined queues, which are run (triggered) by the kernel itself.

This section first describes task queues, then introduces predefined task queues, which provide a good start for some interesting tests (and hang the computer if something goes wrong), and finally introduces how to run your own task queues. Following that, we look at the new *tasklet* interface, which supersedes task queues in many situations in the 2.4 kernel.

The Nature of Task Queues

A task queue is a list of tasks, each task being represented by a function pointer and an argument. When a task is run, it receives a single `void *` argument and returns `void`. The pointer argument can be used to pass along a data structure to the routine, or it can be ignored. The queue itself is a list of structures (the tasks) that are owned by the kernel module declaring and queueing them. The module is completely responsible for allocating and deallocating the structures, and static structures are commonly used for this purpose.

A queue element is described by the following structure, copied directly from `<linux/tqueue.h>`:

```
struct tq_struct {
    struct tq_struct *next;         /* linked list of active bh's */
    int sync;                       /* must be initialized to zero */
    void (*routine)(void *);        /* function to call */
    void *data;                     /* argument to function */
};
```

The "bh" in the first comment means *bottom half*. A bottom half is "half of an interrupt handler"; we'll discuss this topic thoroughly when we deal with interrupts in "Tasklets and Bottom-Half Processing," in Chapter 9. For now, suffice it to say that a bottom half is a mechanism provided by a device driver to handle asynchronous tasks which, usually, are too large to be done while handling a hardware interrupt. This chapter should make sense without an understanding of bottom halves, but we will, by necessity, refer to them occasionally.

The most important fields in the data structure just shown are `routine` and `data`. To queue a task for later execution, you need to set both these fields before queueing the structure, while `next` and `sync` should be cleared. The `sync` flag in the structure is used by the kernel to prevent queueing the same task more than once, because this would corrupt the `next` pointer. Once the task has been queued, the structure is considered "owned" by the kernel and shouldn't be modified until the task is run.

The other data structure involved in task queues is `task_queue`, which is currently just a pointer to `struct tq_struct`; the decision to `typedef` this pointer to another symbol permits the extension of `task_queue` in the future, should the need arise. `task_queue` pointers should be initialized to `NULL` before use.

The following list summarizes the operations that can be performed on task queues and `struct tq_struct`s.

`DECLARE_TASK_QUEUE(name);`
: This macro declares a task queue with the given `name`, and initializes it to the empty state.

`int queue_task(struct tq_struct *task, task_queue *list);`
: As its name suggests, this function queues a task. The return value is 0 if the task was already present on the given queue, nonzero otherwise.

`void run_task_queue(task_queue *list);`
: This function is used to consume a queue of accumulated tasks. You won't need to call it yourself unless you declare and maintain your own queue.

Before getting into the details of using task queues, we need to pause for a moment to look at how they work inside the kernel.

How Task Queues Are Run

A task queue, as we have already seen, is in practice a linked list of functions to call. When *run_task_queue* is asked to run a given queue, each entry in the list is executed. When you are writing functions that work with task queues, you have to keep in mind when the kernel will call *run_task_queue*; the exact context imposes some constraints on what you can do. You should also not make any assumptions regarding the order in which enqueued tasks are run; each of them must do its task independently of the other ones.

And when are task queues run? If you are using one of the predefined task queues discussed in the next section, the answer is "when the kernel gets around to it." Different queues are run at different times, but they are always run when the kernel has no other pressing work to do.

Most important, they almost certainly are *not* run when the process that queued the task is executing. They are, instead, run asynchronously. Until now, everything we have done in our sample drivers has run in the context of a process executing system calls. When a task queue runs, however, that process could be asleep, executing on a different processor, or could conceivably have exited altogether.

This asynchronous execution resembles what happens when a hardware interrupt happens (which is discussed in detail in Chapter 9). In fact, task queues are often

run as the result of a "software interrupt." When running in *interrupt mode* (or *interrupt time*) in this way, your code is subject to a number of constraints. We will introduce these constraints now; they will be seen again in several places in this book. Repetition is called for in this case; the rules for interrupt mode must be followed or the system will find itself in deep trouble.

A number of actions require the context of a process in order to be executed. When you are outside of process context (i.e., in interrupt mode), you must observe the following rules:

- No access to user space is allowed. Because there is no process context, there is no path to the user space associated with any particular process.
- The `current` pointer is not valid in interrupt mode, and cannot be used.
- No sleeping or scheduling may be performed. Interrupt-mode code may not call *schedule* or *sleep_on*; it also may not call any other function that may sleep. For example, calling `kmalloc(..., GFP_KERNEL)` is against the rules. Semaphores also may not be used since they can sleep.

Kernel code can tell if it is running in interrupt mode by calling the function *in_interrupt( )*, which takes no parameters and returns nonzero if the processor is running in interrupt time.

One other feature of the current implementation of task queues is that a task can requeue itself in the same queue from which it was run. For instance, a task being run from the timer tick can reschedule itself to be run on the next tick by calling *queue_task* to put itself on the queue again. Rescheduling is possible because the head of the queue is replaced with a `NULL` pointer before consuming queued tasks; as a result, a new queue is built once the old one starts executing.

Although rescheduling the same task over and over might appear to be a pointless operation, it is sometimes useful. For example, consider a driver that moves a pair of stepper motors one step at a time by rescheduling itself on the timer queue until the target has been reached. Another example is the *jiq* module, where the printing function reschedules itself to produce its output—the result is several iterations through the timer queue.

Predefined Task Queues

The easiest way to perform deferred execution is to use the queues that are already maintained by the kernel. There are a few of these queues, but your driver can use only three of them, described in the following list. The queues are declared in `<linux/tqueue.h>`, which you should include in your source.

The scheduler queue
: The scheduler queue is unique among the predefined task queues in that it runs in process context, implying that the tasks it runs have a bit more freedom in what they can do. In Linux 2.4, this queue runs out of a dedicated

kernel thread called *keventd* and is accessed via a function called *schedule_task*. In older versions of the kernel, *keventd* was not used, and the queue (`tq_scheduler`) was manipulated directly.

`tq_timer`
: This queue is run by the timer tick. Because the tick (the function *do_timer*) runs at interrupt time, any task within this queue runs at interrupt time as well.

`tq_immediate`
: The immediate queue is run as soon as possible, either on return from a system call or when the scheduler is run, whichever comes first. The queue is consumed at interrupt time.

Other predefined task queues exist as well, but they are not generally of interest to driver writers.

The timeline of a driver using a task queue is represented in Figure 6-1. The figure shows a driver that queues a function in `tq_immediate` from an interrupt handler.

How the examples work

Examples of deferred computation are available in the *jiq* ("Just In Queue") module, from which the source in this section has been extracted. This module creates */proc* files that can be read using *dd* or other tools; this is similar to *jit*.

The process reading a *jiq* file is put to sleep until the buffer is full.* This sleeping is handled with a simple wait queue, declared as

```
DECLARE_WAIT_QUEUE_HEAD (jiq_wait);
```

The buffer is filled by successive runs of a task queue. Each pass through the queue appends a text string to the buffer being filled; each string reports the current time (in jiffies), the process that is `current` during this pass, and the return value of *in_interrupt*.

The code for filling the buffer is confined to the *jiq_print_tq* function, which executes at each run through the queue being used. The printing function is not interesting and is not worth showing here; instead, let's look at the initialization of the task to be inserted in a queue:

```
struct tq_struct jiq_task; /* global: initialized to zero */

    /* these lines are in jiq_init() */
    jiq_task.routine = jiq_print_tq;
    jiq_task.data = (void *)&jiq_data;
```

* The buffer of a */proc* file is a page of memory, 4 KB, or whatever is appropriate for the platform you use.

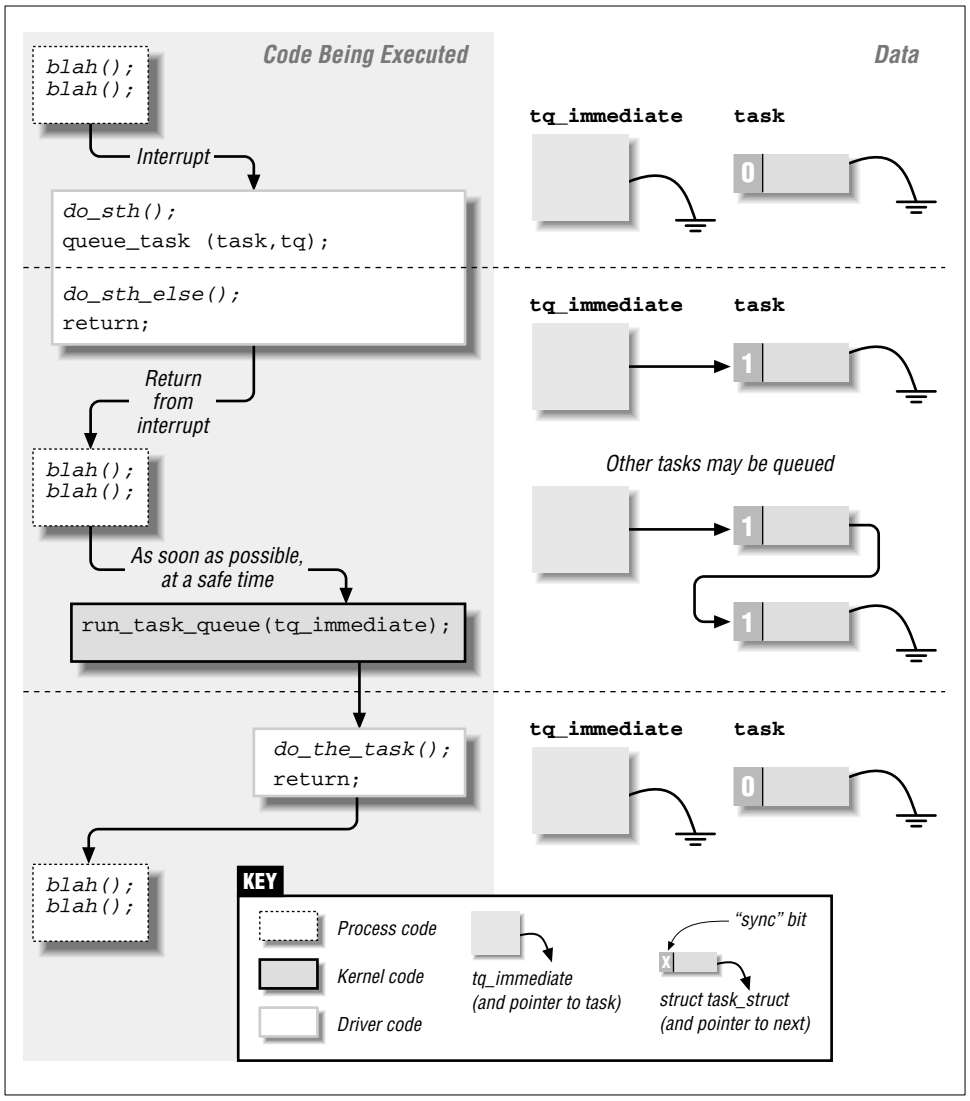

Figure 6-1. Timeline of task-queue usage

There's no need to clear the `sync` and `next` fields of `jiq_task` because static variables are initialized to 0 by the compiler.

The scheduler queue

The scheduler queue is, in some ways, the easiest to use. Because tasks executed

from this queue do not run in interrupt mode, they can do more things; in particular, they can sleep. Many parts of the kernel use this queue to accomplish a wide variety of tasks.

As of kernel 2.4.0-test11, the actual task queue implementing the scheduler queue is hidden from the rest of the kernel. Rather than use *queue_task* directly, code using this queue must call *schedule_task* to put a task on the queue:

```
int schedule_task(struct tq_struct *task);
```

`task`, of course, is the task to be scheduled. The return value is directly from *queue_task*: nonzero if the task was not already on the queue.

Again, as of 2.4.0-test11, the kernel runs a special process, called *keventd*, whose sole job is running tasks from the scheduler queue. *keventd* provides a predictable process context for the tasks it runs (unlike the previous implementation, which would run tasks under an essentially random process's context).

There are a couple of implications to the *keventd* implementation that are worth keeping in mind. The first is that tasks in this queue can sleep, and some kernel code takes advantage of that freedom. Well-behaved code, however, should take care to sleep only for very short periods of time, since no other tasks will be run from the scheduler queue while *keventd* is sleeping. It is also a good idea to keep in mind that your task shares the scheduler queue with others, which can also sleep. In normal situations, tasks placed in the scheduler queue will run very quickly (perhaps even before *schedule_task* returns). If some other task sleeps, though, the time that elapses before your tasks execute could be significant. Tasks that absolutely have to run within a narrow time window should use one of the other queues.

/proc/jiqsched is a sample file that uses the scheduler queue. The *read* function for the file dispatches everything to the task queue in the following way:

```
int jiq_read_sched(char *buf, char **start, off_t offset,
                   int len, int *eof, void *data)
{

    jiq_data.len = 0;               /* nothing printed, yet */
    jiq_data.buf = buf;             /* print in this place */
    jiq_data.jiffies = jiffies;     /* initial time */

    /* jiq_print will queue_task() again in jiq_data.queue */
    jiq_data.queue = SCHEDULER_QUEUE;

    schedule_task(&jiq_task);             /* ready to run */
    interruptible_sleep_on(&jiq_wait);    /* sleep till completion */

    *eof = 1;
    return jiq_data.len;
}
```

Reading */proc/jiqsched* produces output like the following:

```
 time  delta interrupt  pid cpu command
601687   0         0       2   1 keventd
601687   0         0       2   1 keventd
601687   0         0       2   1 keventd
601687   0         0       2   1 keventd
601687   0         0       2   1 keventd
601687   0         0       2   1 keventd
601687   0         0       2   1 keventd
601687   0         0       2   1 keventd
601687   0         0       2   1 keventd
```

In this output, the `time` field is the value of `jiffies` when the task is run, `delta` is the change in `jiffies` since the last time the task ran, `interrupt` is the output of the *in_interrupt* function, `pid` is the ID of the running process, `cpu` is the number of the CPU being used (always 0 on uniprocessor systems), and `command` is the command being run by the current process.

In this case, we see that the task is always running under the *keventd* process. It also runs very quickly—a task that resubmits itself to the scheduler queue can run hundreds or thousands of times within a single timer tick. Even on a very heavily loaded system, the latency in the scheduler queue is quite small.

The timer queue

The timer queue is different from the scheduler queue in that the queue (`tq_timer`) is directly available. Also, of course, tasks run from the timer queue are run in interrupt mode. Additionally, you're guaranteed that the queue will run at the next clock tick, thus eliminating latency caused by system load.

The sample code implements */proc/jiqtimer* with the timer queue. For this queue, it must use *queue_task* to get things going:

```
int jiq_read_timer(char *buf, char **start, off_t offset,
                   int len, int *eof, void *data)
{

    jiq_data.len = 0;            /* nothing printed, yet */
    jiq_data.buf = buf;          /* print in this place */
    jiq_data.jiffies = jiffies;  /* initial time */
    jiq_data.queue = &tq_timer;  /* reregister yourself here */

    queue_task(&jiq_task, &tq_timer);     /* ready to run */
    interruptible_sleep_on(&jiq_wait);    /* sleep till completion */

    *eof = 1;
    return jiq_data.len;
}
```

The following is what *head /proc/jiqtimer* returned on a system that was compiling a new kernel:

```
   time  delta interrupt  pid cpu command
45084845   1        1    8783   0 cc1
45084846   1        1    8783   0 cc1
45084847   1        1    8783   0 cc1
45084848   1        1    8783   0 cc1
45084849   1        1    8784   0 as
45084850   1        1    8758   1 cc1
45084851   1        1    8789   0 cpp
45084852   1        1    8758   1 cc1
45084853   1        1    8758   1 cc1
45084854   1        1    8758   1 cc1
45084855   1        1    8758   1 cc1
```

Note, this time, that exactly one timer tick goes by between each invocation of the task, and that an arbitrary process is running.

The immediate queue

The last predefined queue that can be used by modularized code is the immediate queue. This queue is run via the bottom-half mechanism, which means that one additional step is required to use it. Bottom halves are run only when the kernel has been told that a run is necessary; this is accomplished by "marking" the bottom half. In the case of `tq_immediate`, the necessary call is *mark_bh(IMMEDIATE_BH)*. Be sure to call *mark_bh after* the task has been queued; otherwise, the kernel may run the task queue before your task has been added.

The immediate queue is the fastest queue in the system—it's executed soonest and is run in interrupt time. The queue is consumed either by the scheduler or as soon as one process returns from its system call. Typical output can look like this:

```
   time  delta interrupt  pid cpu command
45129449   0        1    8883   0 head
45129453   4        1       0   0 swapper
45129453   0        1     601   0 X
45129453   0        1     601   0 X
45129453   0        1     601   0 X
45129453   0        1     601   0 X
45129454   1        1       0   0 swapper
45129454   0        1     601   0 X
45129454   0        1     601   0 X
45129454   0        1     601   0 X
45129454   0        1     601   0 X
45129454   0        1     601   0 X
45129454   0        1     601   0 X
45129454   0        1     601   0 X
```

It's clear that the queue can't be used to delay the execution of a task—it's an "immediate" queue. Instead, its purpose is to execute a task as soon as possible,

but at a safe time. This feature makes it a great resource for interrupt handlers, because it offers them an entry point for executing program code outside of the actual interrupt management routine. The mechanism used to receive network packets, for example, is based on a similar mechanism.

Please note that you should not reregister your task in this queue (although we do it in *jiqimmed* for explanatory purposes). The practice gains nothing and may lock the computer hard if run on some version/platform pairs. Some implementations used to rerun the queue until it was empty. This was true, for example, for version 2.0 running on the PC platform.

Running Your Own Task Queues

Declaring a new task queue is not difficult. A driver is free to declare a new task queue, or even several of them; tasks are queued just as we've seen with the predefined queues discussed previously.

Unlike a predefined task queue, however, a custom queue is not automatically run by the kernel. The programmer who maintains a queue must arrange for a way of running it.

The following macro declares the queue and expands to a variable declaration. You'll most likely place it at the beginning of your file, outside of any function:

```
DECLARE_TASK_QUEUE(tq_custom);
```

After declaring the queue, you can invoke the usual functions to queue tasks. The call just shown pairs naturally with the following:

```
queue_task(&custom_task, &tq_custom);
```

The following line will run `tq_custom` when it is time to execute the task-queue entries that have accumulated:

```
run_task_queue(&tq_custom);
```

If you want to experiment with custom queues now, you need to register a function to trigger the queue in one of the predefined queues. Although this may look like a roundabout way to do things, it isn't. A custom queue can be useful whenever you need to accumulate jobs and execute them all at the same time, even if you use another queue to select that "same time."

Tasklets

Shortly before the release of the 2.4 kernel, the developers added a new mechanism for the deferral of kernel tasks. This mechanism, called *tasklets*, is now the preferred way to accomplish bottom-half tasks; indeed, bottom halves themselves are now implemented with tasklets.

Tasklets resemble task queues in a number of ways. They are a way of deferring a task until a safe time, and they are always run in interrupt time. Like task queues, tasklets will be run only once, even if scheduled multiple times, but tasklets may be run in parallel with other (different) tasklets on SMP systems. On SMP systems, tasklets are also guaranteed to run on the CPU that first schedules them, which provides better cache behavior and thus better performance.

Each tasklet has associated with it a function that is called when the tasklet is to be executed. The life of some kernel developer was made easier by giving that function a single argument of type `unsigned long`, which makes life a little more annoying for those who would rather pass it a pointer; casting the `long` argument to a pointer type is a safe practice on all supported architectures and pretty common in memory management (as discussed in Chapter 13). The tasklet function is of type `void`; it returns no value.

Software support for tasklets is part of `<linux/interrupt.h>`, and the tasklet itself must be declared with one of the following:

`DECLARE_TASKLET(name, function, data);`
: Declares a tasklet with the given name; when the tasklet is to be executed (as described later), the given function is called with the (unsigned long) data value.

`DECLARE_TASKLET_DISABLED(name, function, data);`
: Declares a tasklet as before, but its initial state is "disabled," meaning that it can be scheduled but will not be executed until enabled at some future time.

The sample *jiq* driver, when compiled against 2.4 headers, implements */proc/jiqtasklet*, which works like the other *jiq* entries but uses tasklets; we didn't emulate tasklets for older kernel versions in *sysdep.h*. The module declares its tasklet as

```
void jiq_print_tasklet (unsigned long);
DECLARE_TASKLET (jiq_tasklet, jiq_print_tasklet, (unsigned long)
   &jiq_data);
```

When your driver wants to schedule a tasklet to run, it calls *tasklet_schedule*:

```
tasklet_schedule(&jiq_tasklet);
```

Once a tasklet is scheduled, it is guaranteed to be run once (if enabled) at a safe time. Tasklets may reschedule themselves in much the same manner as task queues. A tasklet need not worry about running against itself on a multiprocessor system, since the kernel takes steps to ensure that any given tasklet is only running in one place. If your driver implements multiple tasklets, however, it should be prepared for the possibility that more than one of them could run simultaneously. In that case, spinlocks must be used to protect critical sections of the code (semaphores, which can sleep, may not be used in tasklets since they run in interrupt time).

The output from */proc/jiqtasklet* looks like this:

```
   time  delta interrupt  pid cpu command
45472377   0        1    8904   0 head
45472378   1        1       0   0 swapper
45472379   1        1       0   0 swapper
45472380   1        1       0   0 swapper
45472383   3        1       0   0 swapper
45472383   0        1     601   0 X
45472383   0        1     601   0 X
45472383   0        1     601   0 X
45472383   0        1     601   0 X
45472389   6        1       0   0 swapper
```

Note that the tasklet always runs on the same CPU, even though this output was produced on a dual-CPU system.

The tasklet subsystem provides a few other functions for advanced use of tasklets:

`void tasklet_disable(struct tasklet_struct *t);`
: This function disables the given tasklet. The tasklet may still be scheduled with *tasklet_schedule*, but its execution will be deferred until a time when the tasklet has been enabled again.

`void tasklet_enable(struct tasklet_struct *t);`
: Enables a tasklet that had been previously disabled. If the tasklet has already been scheduled, it will run soon (but not directly out of *tasklet_enable*).

`void tasklet_kill(struct tasklet_struct *t);`
: This function may be used on tasklets that reschedule themselves indefinitely. *tasklet_kill* will remove the tasklet from any queue that it is on. In order to avoid race conditions with the tasklet rescheduling itself, this function waits until the tasklet executes, then pulls it from the queue. Thus, you can be sure that tasklets will not be interrupted partway through. If, however, the tasklet is not currently running and rescheduling itself, *tasklet_kill* may hang. *tasklet_kill* may not be called in interrupt time.

Kernel Timers

The ultimate resources for time keeping in the kernel are the timers. Timers are used to schedule execution of a function (a timer handler) at a particular time in the future. They thus work differently from task queues and tasklets in that you can specify *when* in the future your function will be called, whereas you can't tell exactly when a queued task will be executed. On the other hand, kernel timers are similar to task queues in that a function registered in a kernel timer is executed only once—timers aren't cyclic.

There are times when you need to execute operations detached from any process's context, like turning off the floppy motor or finishing another lengthy shutdown operation. In that case, delaying the return from *close* wouldn't be fair to the application program. Using a task queue would be wasteful, because a queued task must continually reregister itself until the requisite time has passed.

A timer is much easier to use. You register your function once, and the kernel calls it once when the timer expires. Such a functionality is used often within the kernel proper, but it is sometimes needed by the drivers as well, as in the example of the floppy motor.

The kernel timers are organized in a doubly linked list. This means that you can create as many timers as you want. A timer is characterized by its timeout value (in jiffies) and the function to be called when the timer expires. The timer handler receives an argument, which is stored in the data structure, together with a pointer to the handler itself.

The data structure of a timer looks like the following, which is extracted from `<linux/timer.h>`):

```
struct timer_list {
    struct timer_list *next;          /* never touch this */
    struct timer_list *prev;          /* never touch this */
    unsigned long expires;            /* the timeout, in jiffies */
    unsigned long data;               /* argument to the handler */
    void (*function)(unsigned long);  /* handler of the timeout */
    volatile int running;             /* added in 2.4; don't touch */
};
```

The timeout of a timer is a value in jiffies. Thus, `timer->function` will run when `jiffies` is equal to or greater than `timer->expires`. The timeout is an absolute value; it is usually generated by taking the current value of `jiffies` and adding the amount of the desired delay.

Once a `timer_list` structure is initialized, *add_timer* inserts it into a sorted list, which is then polled more or less 100 times per second. Even systems (such as the Alpha) that run with a higher clock interrupt frequency do not check the timer list more often than that; the added timer resolution would not justify the cost of the extra passes through the list.

These are the functions used to act on timers:

`void init_timer(struct timer_list *timer);`
: This inline function is used to initialize the timer structure. Currently, it zeros the `prev` and `next` pointers (and the `running` flag on SMP systems). Programmers are strongly urged to use this function to initialize a timer and to never explicitly touch the pointers in the structure, in order to be forward compatible.

`void add_timer(struct timer_list *timer);`
This function inserts a timer into the global list of active timers.

`int mod_timer(struct timer_list *timer, unsigned long expires);`
Should you need to change the time at which a timer expires, *mod_timer* can be used. After the call, the new `expires` value will be used.

`int del_timer(struct timer_list *timer);`
If a timer needs to be removed from the list before it expires, *del_timer* should be called. When a timer expires, on the other hand, it is automatically removed from the list.

`int del_timer_sync(struct timer_list *timer);`
This function works like *del_timer*, but it also guarantees that, when it returns, the timer function is not running on any CPU. *del_timer_sync* is used to avoid race conditions when a timer function is running at unexpected times; it should be used in most situations. The caller of *del_timer_sync* must ensure that the timer function will not use *add_timer* to add itself again.

An example of timer usage can be seen in the *jiq* module. The file */proc/jitimer* uses a timer to generate two data lines; it uses the same printing function as the task queue examples do. The first data line is generated from the *read* call (invoked by the user process looking at */proc/jitimer*), while the second line is printed by the timer function after one second has elapsed.

The code for */proc/jitimer* is as follows:

```
struct timer_list jiq_timer;

void jiq_timedout(unsigned long ptr)
{
    jiq_print((void *)ptr);            /* print a line */
    wake_up_interruptible(&jiq_wait);  /* awaken the process */
}


int jiq_read_run_timer(char *buf, char **start, off_t offset,
                   int len, int *eof, void *data)
{

    jiq_data.len = 0;      /* prepare the argument for jiq_print() */
    jiq_data.buf = buf;
    jiq_data.jiffies = jiffies;
    jiq_data.queue = NULL;      /* don't requeue */

    init_timer(&jiq_timer);              /* init the timer structure */
    jiq_timer.function = jiq_timedout;
    jiq_timer.data = (unsigned long)&jiq_data;
    jiq_timer.expires = jiffies + HZ; /* one second */
```

```
        jiq_print(&jiq_data);   /* print and go to sleep */
        add_timer(&jiq_timer);
        interruptible_sleep_on(&jiq_wait);
        del_timer_sync(&jiq_timer);  /* in case a signal woke us up */

        *eof = 1;
        return jiq_data.len;
    }
```

Running *head /proc/jitimer* gives the following output:

```
        time  delta interrupt  pid cpu command
    45584582   0        0    8920   0 head
    45584682 100        1       0   1 swapper
```

From the output you can see that the timer function, which printed the last line here, was running in interrupt mode.

What can appear strange when using timers is that the timer expires at just the right time, even if the processor is executing in a system call. We suggested earlier that when a process is running in kernel space, it won't be scheduled away; the clock tick, however, is special, and it does all of its tasks independent of the current process. You can try to look at what happens when you read */proc/jitbusy* in the background and */proc/jitimer* in the foreground. Although the system appears to be locked solid by the busy-waiting system call, both the timer queue and the kernel timers continue running.

Thus, timers can be another source of race conditions, even on uniprocessor systems. Any data structures accessed by the timer function should be protected from concurrent access, either by being atomic types (discussed in Chapter 10) or by using spinlocks.

One must also be very careful to avoid race conditions with timer deletion. Consider a situation in which a module's timer function is run on one processor while a related event (a file is closed or the module is removed) happens on another. The result could be the timer function expecting a situation that is no longer valid, resulting in a system crash. To avoid this kind of race, your module should use *del_timer_sync* instead of *del_timer*. If the timer function can restart the timer itself (a common pattern), you should also have a "stop timer" flag that you set before calling *del_timer_sync*. The timer function should then check that flag and not reschedule itself with *add_timer* if the flag has been set.

Another pattern that can cause race conditions is modifying timers by deleting them with *del_timer*, then creating a new one with *add_timer*. It is better, in this situation, to simply use *mod_timer* to make the necessary change.

Backward Compatibility

Task queues and timing issues have remained relatively constant over the years. Nonetheless, a few things have changed and must be kept in mind.

The functions *sleep_on_timeout*, *interruptible_sleep_on_timeout*, and *schedule_timeout* were all added for the 2.2 kernel. In the 2.0 days, timeouts were handled with a variable (called `timeout`) in the task structure. As a result, code that now makes a call like

```
interruptible_sleep_on_timeout(my_queue, timeout);
```

used to be implemented as

```
current->timeout = jiffies + timeout;
interruptible_sleep_on(my_queue);
```

The *sysdep.h* header recreates *schedule_timeout* for pre-2.4 kernels so that you can use the new syntax and run on 2.0 and 2.2:

```
extern inline void schedule_timeout(int timeout)
{
    current->timeout = jiffies + timeout;
    current->state = TASK_INTERRUPTIBLE;
    schedule();
    current->timeout = 0;
}
```

In 2.0, there were a couple of additional functions for putting functions into task queues. *queue_task_irq* could be called instead of *queue_task* in situations in which interrupts were disabled, yielding a (very) small performance benefit. *queue_task_irq_off* is even faster, but does not function properly in situations in which the task is already queued or is running, and can thus only be used where those conditions are guaranteed not to occur. Neither of these two functions provided much in the way of performance benefits, and they were removed in kernel 2.1.30. Using *queue_task* in all cases works with all kernel versions. (It is worth noting, though, that *queue_task* had a return type of `void` in 2.2 and prior kernels.)

Prior to 2.4, the *schedule_task* function and associated *keventd* process did not exist. Instead, another predefined task queue, `tq_scheduler`, was provided. Tasks placed in `tq_scheduler` were run in the *schedule* function, and thus always ran in process context. The actual process whose context would be used was always different, however; it was whatever process was being scheduled on the CPU at the time. `tq_scheduler` typically had larger latencies, especially for tasks that resubmitted themselves. *sysdep.h* provides the following implementation for *schedule_task* on 2.0 and 2.2 systems:

```
extern inline int schedule_task(struct tq_struct *task)
{
        queue_task(task, &tq_scheduler);
        return 1;
}
```

As has been mentioned, the 2.3 development series added the tasklet mechanism; before, only task queues were available for "immediate deferred" execution. The bottom-half subsystem was implemented differently, though most of the changes are not visible to driver writers. We didn't emulate tasklets for older kernels in *sysdep.h* because they are not strictly needed for driver operation; if you want to be backward compatible you'll need to either write your own emulation or use task queues instead.

The *in_interrupt* function did not exist in Linux 2.0. Instead, a global variable `intr_count` kept track of the number of interrupt handlers running. Querying `intr_count` is semantically the same as calling *in_interrupt*, so compatibility is easily implemented in *sysdep.h*.

The *del_timer_sync* function did not exist prior to development kernel 2.4.0-test2. The usual *sysdep.h* header defines a minimal replacement when you build against older kernel headers. Kernel version 2.0 didn't have *mod_timer*, either. This gap is also filled by our compatibility header.

Quick Reference

This chapter introduced the following symbols:

`#include <linux/param.h>`
`HZ` The `HZ` symbol specifies the number of clock ticks generated per second.

`#include <linux/sched.h>`
`volatile unsigned long jiffies`
The `jiffies` variable is incremented once for each clock tick; thus, it's incremented `HZ` times per second.

`#include <asm/msr.h>`
`rdtsc(low,high);`
`rdtscl(low);`
Read the timestamp counter or its lower half. The header and macros are specific to PC-class processors; other platforms may need `asm` constructs to achieve similar results.

`extern struct timeval xtime;`
The current time, as calculated at the last timer tick.

```
#include <linux/time.h>
void do_gettimeofday(struct timeval *tv);
void get_fast_time(struct timeval *tv);
```
The functions return the current time; the former is very high resolution, the latter may be faster while giving coarser resolution.

```
#include <linux/delay.h>
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```
The functions introduce delays of an integer number of microseconds and milliseconds. The former should be used to wait for no longer than one millisecond; the latter should be used with extreme care because these delays are both busy-loops.

```
int in_interrupt();
```
Returns nonzero if the processor is currently running in interrupt mode.

```
#include <linux/tqueue.h>
DECLARE_TASK_QUEUE(variablename);
```
The macro declares a new variable and initializes it.

```
void queue_task(struct tq_struct *task, task_queue *list);
```
The function registers a task for later execution.

```
void run_task_queue(task_queue *list);
```
This function consumes a task queue.

```
task_queue tq_immediate, tq_timer;
```
These predefined task queues are run as soon as possible (for `tq_immediate`), or after each timer tick (for `tq_timer`).

```
int schedule_task(struct tq_struct *task);
```
Schedules a task to be run on the scheduler queue.

```
#include <linux/interrupt.h>
DECLARE_TASKLET(name, function, data)
DECLARE_TASKLET_DISABLED(name, function, data)
```
Declare a tasklet structure that will call the given function (passing it the given `unsigned long` data) when the tasklet is executed. The second form initializes the tasklet to a disabled state, keeping it from running until it is explicitly enabled.

```
void tasklet_schedule(struct tasklet_struct *tasklet);
```
Schedules the given tasklet for running. If the tasklet is enabled, it will be run shortly on the same CPU that ran the first call to *tasklet_schedule*.

```
tasklet_enable(struct tasklet_struct *tasklet);
tasklet_disable(struct tasklet_struct *tasklet);
```
These functions respectively enable and disable the given tasklet. A disabled tasklet can be scheduled, but will not run until it has been enabled again.

```
void tasklet_kill(struct tasklet_struct *tasklet);
```
Causes an "infinitely rescheduling" tasklet to cease execution. This function can block and may not be called in interrupt time.

```
#include <linux/timer.h>
void init_timer(struct timer_list * timer);
```
This function initializes a newly allocated `timer`.

```
void add_timer(struct timer_list * timer);
```
This function inserts the `timer` into the global list of pending timers.

```
int mod_timer(struct timer_list *timer, unsigned long
      expires);
```
This function is used to change the expiration time of an already scheduled timer structure.

```
int del_timer(struct timer_list * timer);
```
del_timer removes a timer from the list of pending timers. If the timer was actually queued, *del_timer* returns 1; otherwise, it returns 0.

```
int del_timer_sync(struct timer_list *timer);
```
This function is similar to *del_timer*, but guarantees that the function is not currently running on other CPUs.