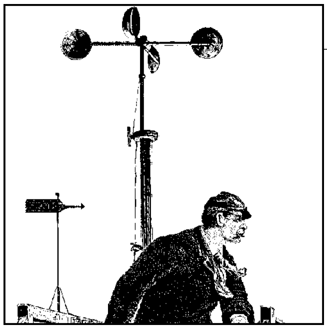

CHAPTER 7

Time, Delays, and Deferred Work

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 operations that control a device; they have to deal with issues such as timing, memory management, hardware access, and more. Fortunately, the kernel exports a number of facilities to ease the task of the driver writer. In the next few chapters, we'll describe some of the kernel resources you can use. This chapter leads the way by describing how timing issues are addressed. Dealing with time involves the following tasks, in order of increasing complexity:

- Measuring time lapses and comparing times
- Knowing the current time
- Delaying operation for a specified amount of time
- Scheduling asynchronous functions to happen at a later time

Measuring Time Lapses

The kernel keeps track of the flow of time by means of timer interrupts. Interrupts are covered in detail in Chapter 10.

Timer interrupts are generated by the system's timing hardware at regular intervals; this interval is programmed at boot time by the kernel according to the value of `HZ`, which is an architecture-dependent value defined in *<linux/param.h>* or a subplatform file included by it. Default values in the distributed kernel source range from 50 to 1200 ticks per second on real hardware, down to 24 for software simulators. Most platforms run at 100 or 1000 interrupts per second; the popular x86 PC defaults to 1000, although it used to be 100 in previous versions (up to and including 2.4). As a general rule, even if you know the value of `HZ`, you should never count on that specific value when programming.

It is possible to change the value of `HZ` for those who want systems with a different clock interrupt frequency. If you change `HZ` in the header file, you need to recompile the kernel and all modules with the new value. You might want to raise `HZ` to get a more fine-grained resolution in your asynchronous tasks, if you are willing to pay the overhead of the extra timer interrupts to achieve your goals. Actually, raising `HZ` to 1000 was pretty common with x86 industrial systems using Version 2.4 or 2.2 of the kernel. With current versions, 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. Besides, some internal calculations are currently implemented only for `HZ` in the range from 12 to 1535 (see <*linux/timex.h*> and RFC-1589).

Every time a timer interrupt occurs, the value of an internal kernel counter is incremented. The counter is initialized to `0` at system boot, so it represents the number of clock ticks since last boot. The counter is a 64-bit variable (even on 32-bit architectures) and is called `jiffies_64`. However, driver writers normally access the `jiffies` variable, an `unsigned long` that is the same as either `jiffies_64` or its least significant bits. Using `jiffies` is usually preferred because it is faster, and accesses to the 64-bit `jiffies_64` value are not necessarily atomic on all architectures.

In addition to the low-resolution kernel-managed jiffy mechanism, some CPU platforms feature a high-resolution counter that software can read. Although its actual use varies somewhat across platforms, it's sometimes a very powerful tool.

Using the jiffies Counter

The counter and the utility functions to read it live in <*linux/jiffies.h*>, although you'll usually just include <*linux/sched.h*>, that automatically pulls *jiffies.h* in. Needless to say, both `jiffies` and `jiffies_64` must be considered read-only.

Whenever your code needs to remember the current value of `jiffies`, it can simply access the `unsigned long` variable, which is declared as volatile to tell the compiler not to optimize memory reads. You need to read the current counter whenever your code needs to calculate a future time stamp, as shown in the following example:

```
#include <linux/jiffies.h>
unsigned long j, stamp_1, stamp_half, stamp_n;

j = jiffies;                      /* read the current value */
stamp_1    = j + HZ;              /* 1 second in the future */
stamp_half = j + HZ/2;            /* half a second */
stamp_n    = j + n * HZ / 1000;   /* n milliseconds */
```

This code has no problem with `jiffies` wrapping around, as long as different values are compared in the right way. Even though on 32-bit platforms the counter wraps around only once every 50 days when `HZ` is 1000, your code should be prepared to face that event. To compare your cached value (like `stamp_1` above) and the current value, you should use one of the following macros:

```
#include <linux/jiffies.h>
int time_after(unsigned long a, unsigned long b);
int time_before(unsigned long a, unsigned long b);
int time_after_eq(unsigned long a, unsigned long b);
int time_before_eq(unsigned long a, unsigned long b);
```

The first evaluates true when *a*, as a snapshot of `jiffies`, represents a time after *b*, the second evaluates true when time *a* is before time *b*, and the last two compare for "after or equal" and "before or equal." The code works by converting the values to signed long, subtracting them, and comparing the result. If you need to know the difference between two instances of `jiffies` in a safe way, you can use the same trick: `diff = (long)t2 - (long)t1;`.

You can convert a jiffies difference to milliseconds trivially through:

```
msec = diff * 1000 / HZ;
```

Sometimes, however, you need to exchange time representations with user space programs that tend to represent time values with `struct timeval` and `struct timespec`. The two structures represent a precise time quantity with two numbers: seconds and microseconds are used in the older and popular `struct timeval`, and seconds and nanoseconds are used in the newer `struct timespec`. The kernel exports four helper functions to convert time values expressed as jiffies to and from those structures:

```
#include <linux/time.h>

unsigned long timespec_to_jiffies(struct timespec *value);
void jiffies_to_timespec(unsigned long jiffies, struct timespec *value);
unsigned long timeval_to_jiffies(struct timeval *value);
void jiffies_to_timeval(unsigned long jiffies, struct timeval *value);
```

Accessing the 64-bit jiffy count is not as straightforward as accessing `jiffies`. While on 64-bit computer architectures the two variables are actually one, access to the value is not atomic for 32-bit processors. This means you might read the wrong value if both halves of the variable get updated while you are reading them. It's extremely unlikely you'll ever need to read the 64-bit counter, but in case you do, you'll be glad to know that the kernel exports a specific helper function that does the proper locking for you:

```
#include <linux/jiffies.h>
u64 get_jiffies_64(void);
```

In the above prototype, the `u64` type is used. This is one of the types defined by *<linux/types.h>*, discussed in Chapter 11, and represents an unsigned 64-bit type.

If you're wondering how 32-bit platforms update both the 32-bit and 64-bit counters at the same time, read the linker script for your platform (look for a file whose name matches *vmlinux*.lds**). There, the `jiffies` symbol is defined to access the least significant word of the 64-bit value, according to whether the platform is little-endian or big-endian. Actually, the same trick is used for 64-bit platforms, so that the `unsigned long` and `u64` variables are accessed at the same address.

Finally, note that the actual clock frequency is almost completely hidden from user space. The macro `HZ` always expands to 100 when user-space programs include *param.h*, and every counter reported to user space is converted accordingly. This applies to *clock(3)*, *times(2)*, and any related function. The only evidence available to users of the `HZ` value is how fast timer interrupts happen, as shown in */proc/interrupts*. For example, you can obtain `HZ` by dividing this count by the system uptime reported in */proc/uptime*.

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, a choice of precision over portability.

In modern processors, the pressing demand for empirical performance figures is thwarted by the intrinsic unpredictability of instruction timing in most CPU designs due to cache memories, instruction scheduling, and branch prediction. As a response, CPU manufacturers introduced a way to count clock cycles as an easy and reliable way to measure time lapses. Therefore, most modern processors include a counter register that is steadily incremented once at each clock cycle. Nowadays, this clock counter is the only reliable way to carry out high-resolution timekeeping tasks.

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 last case, you must be prepared to handle overflows just like we did with the jiffy counter. The register may even not exist for your platform, or it can be implemented in an external device by the hardware designer, if the CPU lacks the feature and you are dealing with a special-purpose computer.

Whether or not the register can be zeroed, we strongly discourage resetting it, even when hardware permits. You might not, after all, be the only user of the counter at any given time; on some platforms supporting SMP, for example, the kernel depends on such a counter to be synchronized across processors. Since you can always measure differences between values, as long as that difference doesn't exceed the overflow time, 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—including the x86_64 platform. 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>* (an x86-specific header whose name stands for "machine-specific registers"), you can use one of these macros:

```
rdtsc(low32,high32);
rdtscl(low32);
rdtscll(var64);
```

The first macro atomically reads the 64-bit value into two 32-bit variables; the next one (“read low half”) reads the low half of the register into a 32-bit variable, discarding the high half; the last reads the 64-bit value into a `long long` variable, hence, the name. All of these macros store values into their arguments.

Reading the low half of the counter is enough for most common uses of the TSC. A 1-GHz CPU overflows it only once every 4.2 seconds, so you won’t need to deal with multiregister variables if the time lapse you are benchmarking reliably takes less time. However, as CPU frequencies rise over time and as timing requirements increase, you’ll most likely need to read the 64-bit counter more often in the future.

As an example using only the low half of the register, the following lines 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 functionality, and kernel headers offer an architecture-independent function that you can use instead of *rdtsc*. It is called *get_cycles*, defined in *<asm/timex.h>* (included by *<linux/timex.h>*). Its prototype is:

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

This 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 to hold the value read.

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

We 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, readable only 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.

---

* 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.

With *gcc* inline assembly, the 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 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.

There is one other thing worth knowing about timestamp counters: they are not necessarily synchronized across processors in an SMP system. To be sure of getting a coherent value, you should disable preemption for code that is querying the counter.

Knowing the Current Time

Kernel code can always retrieve a representation of 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. As shown, 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 or calculate timeouts). In short, looking at `jiffies` is almost always sufficient when you need to measure time intervals. If you need very precise measurements for short time lapses, processor-specific registers come to the rescue (although they bring in serious portability issues).

It's quite unlikely that a driver will ever need to know the wall-clock time, expressed in months, days, and hours; the information is usually needed only by user programs such as *cron* and *syslogd*. Dealing with real-world time is usually best left to user space, where the C library offers better support; besides, such code is often too policy-related to belong in the kernel. There *is* a kernel function that turns a wall-clock time into a `jiffies` value, however:

```
#include <linux/time.h>
unsigned long mktime (unsigned int year, unsigned int mon,
                      unsigned int day, unsigned int hour,
                      unsigned int min, unsigned int sec);
```

To repeat: dealing directly with wall-clock time in a driver is often a sign that policy is being implemented and should therefore be questioned.

While you won't have to deal with human-readable representations of the time, sometimes you need to deal with absolute timestamp even in kernel space. To this aim, *<linux/time.h>* exports the *do_gettimeofday* function. When called, it fills a `struct timeval` pointer—the same one used in the *gettimeofday* system call—with the familiar 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," because it asks the timing hardware what fraction of the current jiffy has already elapsed. The precision varies from one architecture to another, however, since it depends on the actual hardware mechanisms in use. For example, some *m68knommu* processors, Sun3 systems, and other *m68k* systems cannot offer more than jiffy resolution. Pentium systems, on the other hand, offer very fast and precise subtick measures by reading the timestamp counter described earlier in this chapter.

The current time is also available (though with jiffy granularity) from the `xtime` variable, a `struct timespec` value. Direct use of this variable is discouraged because it is difficult to atomically access both the fields. Therefore, the kernel offers the utility function *current_kernel_time*:

```
#include <linux/time.h>
struct timespec current_kernel_time(void);
```

Code for retrieving the current time in the various ways it is available within the *jit* ("just in time") module in the source files provided on O'Reilly's FTP site. *jit* creates a file called */proc/currentime*, which returns the following items in ASCII when read:

- The current `jiffies` and `jiffies_64` values as hex numbers
- The current time as returned by *do_gettimeofday*
- The `timespec` returned by *current_kernel_time*

We chose to use a dynamic */proc* file to keep the boilerplate code to a minimum—it's not worth creating a whole device just to return a little textual information.

The file returns text lines continuously as long as the module is loaded; each *read* system call collects and returns one set of data, organized in two lines for better readability. Whenever you read multiple data sets in less than a timer tick, you'll see the difference between *do_gettimeofday*, which queries the hardware, and the other values that are updated only when the timer ticks.

```
phon% head -8 /proc/currentime
0x00bdbc1f 0x0000000100bdbc1f 1062370899.630126
                              1062370899.629161488
0x00bdbc1f 0x0000000100bdbc1f 1062370899.630150
                              1062370899.629161488
0x00bdbc20 0x0000000100bdbc20 1062370899.630208
                              1062370899.630161336
0x00bdbc20 0x0000000100bdbc20 1062370899.630233
                              1062370899.630161336
```

In the screenshot above, there are two interesting things to note. First, the *current_kernel_time* value, though expressed in nanoseconds, has only clock-tick granularity;

do_gettimeofday consistently reports a later time but not later than the next timer tick. Second, the 64-bit jiffies counter has the least-significant bit of the upper 32-bit word set. This happens because the default value for `INITIAL_JIFFIES`, used at boot time to initialize the counter, forces a low-word overflow a few minutes after boot time to help detect problems related to that very overflow. This initial bias in the counter has no effect, because `jiffies` is unrelated to wall-clock time. In */proc/uptime*, where the kernel extracts the uptime from the counter, the initial bias is removed before conversion.

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 go over them all, and point out the advantages and disadvantages of each.

One important thing to consider is how the delay you need compares with the clock tick, considering the range of `HZ` across the various platforms. Delays that are reliably longer than the clock tick, and don't suffer from its coarse granularity, can make use of the system clock. Very short delays typically must be implemented with software loops. In between these two cases lies a gray area. In this chapter, we use the phrase "long" delay to refer to a multiple-jiffy delay, which can be as low as a few milliseconds on some platforms, but is still long as seen by the CPU and the kernel.

The following sections talk about the different delays by taking a somewhat long path from various intuitive but inappropriate solutions to the right solution. We chose this path because it allows a more in-depth discussion of kernel issues related to timing. If you are eager to find the right code, just skim through the section.

Long Delays

Occasionally a driver needs to delay execution for relatively long periods—more than one clock tick. There are a few ways of accomplishing this sort of delay; we start with the simplest technique, then proceed to the more advanced techniques.

Busy waiting

If you want to delay execution by a multiple of the clock tick, allowing some slack in the value, the easiest (though not recommended) implementation is a loop that monitors the jiffy counter. The *busy-waiting* implementation usually looks like the following code, where `j1` is the value of `jiffies` at the expiration of the delay:

```
while (time_before(jiffies, j1))
    cpu_relax( );
```

The call to *cpu_relax* invokes an architecture-specific way of saying that you're not doing much with the processor at the moment. On many systems it does nothing at all; on symmetric multithreaded ("hyperthreaded") systems, it may yield the core to the other thread. In any case, this approach should definitely be avoided whenever possible. We show it here because on occasion you might want to run this code to better understand 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 fetched from memory any time some C code accesses it. Although technically correct (in that it works as designed), this busy loop severely degrades system performance. If you didn't configure your kernel for preemptive operation, the loop completely locks the processor for the duration of the delay; the scheduler never preempts a process that is running in kernel space, and the computer looks completely dead until time `j1` is reached. The problem is less serious if you are running a preemptive kernel, because, unless the code is holding a lock, some of the processor's time can be recovered for other uses. Busy waits are still expensive on preemptive systems, however.

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. Running a preemptive kernel won't help either, and 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 each time you read a line of text, and lines are guaranteed to be 20 bytes each. If you want to test the busy-wait code, you can read */proc/jitbusy*, which busy-loops for one second for each line it returns.

> Be sure to read, at most, one line (or a few lines) at a time from */proc/jitbusy*. The simplified kernel mechanism to register */proc* files invokes the *read* method over and over to fill the data buffer the user requested. Therefore, a command such as *cat /proc/jitbusy*, if it reads 4 KB at a time, freezes the computer for 205 seconds.

The suggested command to read */proc/jitbusy* is *dd bs=20 < /proc/jitbusy*, optionally specifying the number of blocks as well. Each 20-byte line returned by the file represents the value the jiffy counter had before and after the delay. This is a sample run on an otherwise unloaded computer:

```
phon% dd bs=20 count=5 < /proc/jitbusy
  1686518   1687518
  1687519   1688519
  1688520   1689520
  1689520   1690520
  1690521   1691521
```

All looks good: delays are exactly one second (1000 jiffies), and the next *read* system call starts immediately after the previous one is over. But let's see what happens on a system with a large number of CPU-intensive processes running (and nonpreemptive kernel):

```
phon% dd bs=20 count=5 < /proc/jitbusy
  1911226   1912226
  1913323   1914323
  1919529   1920529
  1925632   1926632
  1931835   1932835
```

Here, each *read* system call delays exactly one second, but the kernel can take more than 5 seconds before scheduling the *dd* process so it can issue the next system call. That's expected in a multitasking system; CPU time is shared between all running processes, and a CPU-intensive process has its dynamic priority reduced. (A discussion of scheduling policies is outside the scope of this book.)

The test under load shown above has been performed while running the *load50* sample program. This program forks a number of processes that do nothing, but do it in a CPU-intensive way. The program is part of the sample files accompanying this book, and forks 50 processes by default, although the number can be specified on the command line. In this chapter, and elsewhere in the book, the tests with a loaded system have been performed with *load50* running in an otherwise idle computer.

If you repeat the command while running a preemptible kernel, you'll find no noticeable difference on an otherwise idle CPU and the following behavior under load:

```
phon% dd bs=20 count=5 < /proc/jitbusy
 14940680  14942777
 14942778  14945430
 14945431  14948491
 14948492  14951960
 14951961  14955840
```

Here, there is no significant delay between the end of a system call and the beginning of the next one, but the individual delays are far longer than one second: up to 3.8 seconds in the example shown and increasing over time. These values demonstrate that the process has been interrupted during its delay, scheduling other processes. The gap between system calls is not the only scheduling option for this process, so no special delay can be seen there.

Yielding the processor

As we have seen, busy waiting imposes a heavy load on the system as a whole; we would like to find a better technique. The first change that comes to mind is to explicitly release the CPU when we're not interested in it. This is accomplished by calling the *schedule* function, declared in <*linux/sched.h*>:

```
while (time_before(jiffies, j1)) {
    schedule( );
}
```

This loop can be tested by reading */proc/jitsched* as we read */proc/jitbusy* above. However, is still isn't optimal. The current process does nothing but release the CPU, but it remains in the run queue. If it is the only runnable process, it actually runs (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) is at least one, and the idle task (process number `0`, also called *swapper* for historical reasons) never runs. 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 is accountable for all the time it consumes.

The behavior of */proc/jitsched* is actually similar to running */proc/jitbusy* under a preemptive kernel. This is a sample run, on an unloaded system:

```
phon% dd bs=20 count=5 < /proc/jitsched
  1760205   1761207
  1761209   1762211
  1762212   1763212
  1763213   1764213
  1764214   1765217
```

It's interesting to note that each *read* sometimes ends up waiting a few clock ticks more than requested. This problem gets worse and worse as the system gets busy, and the driver could end up waiting longer than expected. Once a process releases the processor with *schedule*, there are no guarantees that the process will get the processor back anytime soon. Therefore, calling *schedule* in this manner is not a safe solution to the driver's needs, in addition to being bad for the computing system as a whole. If you test *jitsched* while running *load50*, you can see that the delay associated to each line is extended by a few seconds, because other processes are using the CPU when the timeout expires.

Timeouts

The suboptimal delay loops shown up to now work by watching the jiffy counter without telling anyone. But the best way to implement a delay, as you may imagine, is usually to ask the kernel to do it for you. There are two ways of setting up jiffy-based 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 that it runs within a certain period of time, it can use *wait_event_timeout* or *wait_event_interruptible_timeout*:

```
#include <linux/wait.h>
long wait_event_timeout(wait_queue_head_t q, condition, long timeout);
long wait_event_interruptible_timeout(wait_queue_head_t q,
                      condition, long timeout);
```

These functions sleep on the given wait queue, but they return after the timeout (expressed in jiffies) expires. Thus, they implement a bounded sleep that does not go on forever. Note that the timeout value represents the number of jiffies to wait, not an absolute time value. The value is represented by a signed number, because it sometimes is the result of a subtraction, although the functions complain through a *printk* statement if the provided timeout is negative. If the timeout expires, the functions return `0`; if the process is awakened by another event, it returns the remaining delay expressed in jiffies. The return value is never negative, even if the delay is greater than expected because of system load.

The */proc/jitqueue* file shows a delay based on *wait_event_interruptible_timeout*, although the module has no event to wait for, and uses `0` as a condition:

```
wait_queue_head_t wait;
init_waitqueue_head (&wait);
wait_event_interruptible_timeout(wait, 0, delay);
```

The observed behaviour, when reading */proc/jitqueue*, is nearly optimal, even under load:

```
phon% dd bs=20 count=5 < /proc/jitqueue
  2027024   2028024
  2028025   2029025
  2029026   2030026
  2030027   2031027
  2031028   2032028
```

Since the reading process (*dd* above) is not in the run queue while waiting for the timeout, you see no difference in behavior whether the code is run in a preemptive kernel or not.

wait_event_timeout and *wait_event_interruptible_timeout* were designed with a hardware driver in mind, where execution could be resumed in either of two ways: either somebody calls *wake_up* on the wait queue, or the timeout expires. This doesn't apply to *jitqueue*, as nobody ever calls *wake_up* on the wait queue (after all, no other code even knows about it), so the process always wakes up when the timeout expires. To accommodate for this very situation, where you want to delay execution waiting for no specific event, the kernel offers the *schedule_timeout* function so you can avoid declaring and using a superfluous wait queue head:

```
#include <linux/sched.h>
signed long schedule_timeout(signed long timeout);
```

Here, `timeout` is the number of jiffies to delay. The return value is `0` unless the function returns before the given timeout has elapsed (in response to a signal). *schedule_timeout* requires that the caller first set the current process state, so a typical call looks like:

```
set_current_state(TASK_INTERRUPTIBLE);
schedule_timeout (delay);
```

The previous lines (from */proc/jitschedto*) cause the process to sleep until the given time has passed. Since *wait_event_interruptible_timeout* relies on *schedule_timeout* internally, we won't bother showing the numbers *jitschedto* returns, because they are the same as those of *jitqueue*. 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.

In the example just shown, the first line calls *set_current_state* to set things up so that the scheduler won't run the current process again until the timeout places it back in `TASK_RUNNING` state. To achieve an uninterruptible delay, use `TASK_UNINTERRUPTIBLE` instead. If you forget to change the state of the current process, a call to *schedule_timeout* behaves like a call to *schedule* (i.e., the *jitsched* behavior), setting up a timer that is not used.

If you want to play with the four *jit* files under different system situations or different kernels, or try other ways to delay execution, you may want to configure the amount of the delay when loading the module by setting the *delay* module parameter.

Short Delays

When a device driver needs to deal with latencies in its hardware, the delays involved are usually a few dozen microseconds at most. In this case, relying on the clock tick is definitely not the way to go.

The kernel functions *ndelay*, *udelay*, and *mdelay* serve well for short delays, delaying execution for the specified number of nanoseconds, microseconds, or milliseconds respectively.[*] Their prototypes are:

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

The actual implementations of the functions are in <*asm/delay.h*>, being architecture-specific, and sometimes build on an external function. Every architecture implements *udelay*, but the other functions may or may not be defined; if they are not, <*linux/delay.h*> offers a default version based on *udelay*. In all cases, the delay achieved is at least the requested value but could be more; actually, no platform currently achieves nanosecond precision, although several ones offer submicrosecond

---

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

precision. Delaying more than the requested value is usually not a problem, as short delays in a driver are usually needed to wait for the hardware, and the requirements are to wait for *at least* a given time lapse.

The implementation of *udelay* (and possibly *ndelay* too) uses a software loop based on the processor speed calculated at boot time, using the integer variable `loops_per_jiffy`. If you want to look at the actual code, however, be aware that the *x86* implementation is quite a complex one because of the different timing sources it uses, based on what CPU type is running the code.

To avoid integer overflows in loop calculations, *udelay* and *ndelay* impose an upper bound in the value passed to them. If your module fails to load and displays an unresolved symbol, *__bad_udelay*, it means you called *udelay* with too large an argument. Note, however, that the compile-time check can be performed only on constant values and that not all platforms implement it. As a general rule, if you are trying to delay for thousands of nanoseconds, you should be using *udelay* rather than *ndelay*; similarly, millisecond-scale delays should be done with *mdelay* and not one of the finer-grained functions.

It's important to remember that the three delay functions are busy-waiting; other tasks can't be run during the time lapse. Thus, they replicate, though on a different scale, the behavior of *jitbusy*. Thus, these functions should only be used when there is no practical alternative.

There is another way of achieving millisecond (and longer) delays that does not involve busy waiting. The file *<linux/delay.h>* declares these functions:

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds)
```

The first two functions puts the calling process to sleep for the given number of `millisecs`. A call to *msleep* is uninterruptible; you can be sure that the process sleeps for at least the given number of milliseconds. If your driver is sitting on a wait queue and you want a wakeup to break the sleep, use *msleep_interruptible*. The return value from *msleep_interruptible* is normally `0`; if, however, the process is awakened early, the return value is the number of milliseconds remaining in the originally requested sleep period. A call to *ssleep* puts the process into an uninterruptible sleep for the given number of seconds.

In general, if you can tolerate longer delays than requested, you should use *schedule_timeout*, *msleep*, or *ssleep*.

Kernel Timers

Whenever you need to schedule an action to happen later, without blocking the current process until that time arrives, kernel timers are the tool for you. These timers are used to schedule execution of a function at a particular time in the future, based on the clock tick, and can be used for a variety of tasks; for example, polling a device by checking its state at regular intervals when the hardware can't fire interrupts. Other typical uses of kernel timers are turning off the floppy motor or finishing another lengthy shut down operation. In such cases, delaying the return from *close* would impose an unnecessary (and surprising) cost on the application program. Finally, the kernel itself uses the timers in several situations, including the implementation of *schedule_timeout*.

A kernel timer is a data structure that instructs the kernel to execute a user-defined function with a user-defined argument at a user-defined time. The implementation resides in <*linux/timer.h*> and *kernel/timer.c* and is described in detail in the section "The Implementation of Kernel Timers."

The functions scheduled to run almost certainly do *not* run while the process that registered them 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 timer runs, however, the process that scheduled it could be asleep, executing on a different processor, or quite possibly has exited altogether.

This asynchronous execution resembles what happens when a hardware interrupt happens (which is discussed in detail in Chapter 10). In fact, kernel timers are run as the result of a "software interrupt." When running in this sort of atomic context, your code is subject to a number of constraints. Timer functions must be atomic in all the ways we discussed in the section "Spinlocks and Atomic Context" in Chapter 1, but there are some additional issues brought about by the lack of a process context. We will introduce these constraints now; they will be seen again in several places in later chapters. Repetition is called for because the rules for atomic contexts must be followed assiduously, 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 context), 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 meaningful in atomic mode and cannot be used since the relevant code has no connection with the process that has been interrupted.
- No sleeping or scheduling may be performed. Atomic code may not call *schedule* or a form of *wait_event*, nor may it call any other function that could sleep. For example, calling *kmalloc(..., GFP_KERNEL)* is against the rules. Semaphores also must not be used since they can sleep.

Kernel code can tell if it is running in interrupt context by calling the function *in_interrupt()*, which takes no parameters and returns nonzero if the processor is currently running in interrupt context, either hardware interrupt or software interrupt.

A function related to *in_interrupt()* is *in_atomic()*. Its return value is nonzero whenever scheduling is not allowed; this includes hardware and software interrupt contexts as well as any time when a spinlock is held. In the latter case, `current` may be valid, but access to user space is forbidden, since it can cause scheduling to happen. Whenever you are using *in_interrupt()*, you should really consider whether *in_atomic()* is what you actually mean. Both functions are declared in <*asm/hardirq.h*>

One other important feature of kernel timers is that a task can reregister itself to run again at a later time. This is possible because each `timer_list` structure is unlinked from the list of active timers before being run and can, therefore, be immediately relinked elsewhere. Although rescheduling the same task over and over might appear to be a pointless operation, it is sometimes useful. For example, it can be used to implement the polling of devices.

It's also worth knowing that in an SMP system, the timer function is executed by the same CPU that registered it, to achieve better cache locality whenever possible. Therefore, a timer that reregisters itself always runs on the same CPU.

An important feature of timers that should not be forgotten, though, is that they are a potential source of race conditions, even on uniprocessor systems. This is a direct result of their being asynchronous with other code. Therefore, any data structures accessed by the timer function should be protected from concurrent access, either by being atomic types (discussed in the section "Atomic Variables" in Chapter 1) or by using spinlocks (discussed in Chapter 5).

The Timer API

The kernel provides drivers with a number of functions to declare, register, and remove kernel timers. The following excerpt shows the basic building blocks:

```
#include <linux/timer.h>
struct timer_list {
        /* ... */
        unsigned long expires;
        void (*function)(unsigned long);
        unsigned long data;
};

void init_timer(struct timer_list *timer);
struct timer_list TIMER_INITIALIZER(_function, _expires, _data);

void add_timer(struct timer_list * timer);
int del_timer(struct timer_list * timer);
```

The data structure includes more fields than the ones shown, but those three are the ones that are meant to be accessed from outside the timer code iteslf. The `expires` field represents the `jiffies` value when the timer is expected to run; at that time, the function *function* is called with `data` as an argument. If you need to pass multiple items in the argument, you can bundle them as a single data structure and pass a pointer cast to `unsigned long`, a safe practice on all supported architectures and pretty common in memory management (as discussed in Chapter 15). The `expires` value is not a `jiffies_64` item because timers are not expected to expire very far in the future, and 64-bit operations are slow on 32-bit platforms.

The structure must be initialized before use. This step ensures that all the fields are properly set up, including the ones that are opaque to the caller. Initialization can be performed by calling *init_timer* or assigning `TIMER_INITIALIZER` to a static structure, according to your needs. After initialization, you can change the three public fields before calling *add_timer*. To disable a registered timer before it expires, call *del_timer*.

The *jit* module includes a sample file, */proc/jitimer* (for “just in timer”), that returns one header line and six data lines. The data lines represent the current environment where the code is running; the first one is generated by the *read* file operation and the others by a timer. The following output was recorded while compiling a kernel:

```
phon% cat /proc/jitimer
   time   delta  inirq    pid   cpu command
 33565837    0     0      1269   0   cat
 33565847   10     1      1271   0   sh
 33565857   10     1      1273   0   cpp0
 33565867   10     1      1273   0   cpp0
 33565877   10     1      1274   0   cc1
 33565887   10     1      1274   0   cc1
```

In this output, the `time` field is the value of `jiffies` when the code runs, `delta` is the change in `jiffies` since the previous line, `inirq` is the Boolean value returned by *in_interrupt*, `pid` and `command` refer to the current process, and `cpu` is the number of the CPU being used (always `0` on uniprocessor systems).

If you read */proc/jitimer* while the system is unloaded, you’ll find that the context of the timer is process `0`, the idle task, which is called “swapper” mainly for historical reasons.

The timer used to generate */proc/jitimer* data is run every 10 jiffies by default, but you can change the value by setting the `tdelay` (timer delay) parameter when loading the module.

The following code excerpt shows the part of *jit* related to the *jitimer* timer. When a process attempts to read our file, we set up the timer as follows:

```
unsigned long j = jiffies;

/* fill the data for our timer function */
data->prevjiffies = j;
```

```
    data->buf = buf2;
    data->loops = JIT_ASYNC_LOOPS;

    /* register the timer */
    data->timer.data = (unsigned long)data;
    data->timer.function = jit_timer_fn;
    data->timer.expires = j + tdelay; /* parameter */
    add_timer(&data->timer);

    /* wait for the buffer to fill */
    wait_event_interruptible(data->wait, !data->loops);
```

The actual timer function looks like this:

```
void jit_timer_fn(unsigned long arg)
{
    struct jit_data *data = (struct jit_data *)arg;
    unsigned long j = jiffies;
    data->buf += sprintf(data->buf, "%9li  %3li     %i    %6i   %i   %s\n",
                 j, j - data->prevjiffies, in_interrupt( ) ? 1 : 0,
                 current->pid, smp_processor_id( ), current->comm);

    if (--data->loops) {
        data->timer.expires += tdelay;
        data->prevjiffies = j;
        add_timer(&data->timer);
    } else {
        wake_up_interruptible(&data->wait);
    }
}
```

The timer API includes a few more functions than the ones introduced above. The following set completes the list of kernel offerings:

`int mod_timer(struct timer_list *timer, unsigned long expires);`
: Updates the expiration time of a timer, a common task for which a timeout timer is used (again, the motor-off floppy timer is a typical example). *mod_timer* can be called on inactive timers as well, where you normally use *add_timer*.

`int del_timer_sync(struct timer_list *timer);`
: Works like *del_timer*, but also guarantees that when it returns, the timer function is not running on any CPU. *del_timer_sync* is used to avoid race conditions on SMP systems and is the same as *del_timer* in UP kernels. This function should be preferred over *del_timer* in most situations. This function can sleep if it is called from a nonatomic context but busy waits in other situations. Be very careful about calling *del_timer_sync* while holding locks; if the timer function attempts to obtain the same lock, the system can deadlock. If the timer function reregisters itself, the caller must first ensure that this reregistration will not happen; this is usually accomplished by setting a “shutting down” flag, which is checked by the timer function.

`int timer_pending(const struct timer_list * timer);`
Returns true or false to indicate whether the timer is currently scheduled to run by reading one of the opaque fields of the structure.

The Implementation of Kernel Timers

Although you won't need to know how kernel timers are implemented in order to use them, the implementation is interesting, and a look at its internals is worthwhile.

The implementation of the timers has been designed to meet the following requirements and assumptions:

- Timer management must be as lightweight as possible.
- The design should scale well as the number of active timers increases.
- Most timers expire within a few seconds or minutes at most, while timers with long delays are pretty rare.
- A timer should run on the same CPU that registered it.

The solution devised by kernel developers is based on a per-CPU data structure. The *timer_list* structure includes a pointer to that data structure in its `base` field. If `base` is `NULL`, the timer is not scheduled to run; otherwise, the pointer tells which data structure (and, therefore, which CPU) runs it. Per-CPU data items are described in the section "Per-CPU Variables" in Chapter 8.

Whenever kernel code registers a timer (via *add_timer* or *mod_timer*), the operation is eventually performed by *internal_add_timer* (in `kernel/timer.c`) which, in turn, adds the new timer to a double-linked list of timers within a "cascading table" associated to the current CPU.

The cascading table works like this: if the timer expires in the next 0 to 255 jiffies, it is added to one of the 256 lists devoted to short-range timers using the least significant bits of the `expires` field. If it expires farther in the future (but before 16,384 jiffies), it is added to one of 64 lists based on bits 9–14 of the `expires` field. For timers expiring even farther, the same trick is used for bits 15–20, 21–26, and 27–31. Timers with an expire field pointing still farther in the future (something that can happen only on 64-bit platforms) are hashed with a delay value of `0xffffffff`, and timers with `expires` in the past are scheduled to run at the next timer tick. (A timer that is already expired may sometimes be registered in high-load situations, especially if you run a preemptible kernel.)

When *__run_timers* is fired, it executes all pending timers for the current timer tick. If `jiffies` is currently a multiple of 256, the function also rehashes one of the next-level lists of timers into the 256 short-term lists, possibly cascading one or more of the other levels as well, according to the bit representation of `jiffies`.

This approach, while exceedingly complex at first sight, performs very well both with few timers and with a large number of them. The time required to manage each active timer is independent of the number of timers already registered and is limited to a few logic operations on the binary representation of its `expires` field. The only cost associated with this implementation is the memory for the 512 list heads (256 short-term lists and 4 groups of 64 more lists)—i.e., 4 KB of storage.

The function *__run_timers*, as shown by */proc/jitimer*, is run in atomic context. In addition to the limitations we already described, this brings in an interesting feature: the timer expires at just the right time, even if you are not running a preemptible kernel, and the CPU is busy in kernel space. You can see 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, the kernel timers still work fine.

Keep in mind, however, that a kernel timer is far from perfect, as it suffers from jitter and other artifacts induced by hardware interrupts, as well as other timers and other asynchronous tasks. While a timer associated with simple digital I/O can be enough for simple tasks like running a stepper motor or other amateur electronics, it is usually not suitable for production systems in industrial environments. For such tasks, you'll most likely need to resort to a real-time kernel extension.

Tasklets

Another kernel facility related to timing issues is the *tasklet* mechanism. It is mostly used in interrupt management (we'll see it again in Chapter 10.)

Tasklets resemble kernel timers in some ways. They are always run at interrupt time, they always run on the same CPU that schedules them, and they receive an `unsigned long` argument. Unlike kernel timers, however, you can't ask to execute the function at a specific time. By scheduling a tasklet, you simply ask for it to be executed at a later time chosen by the kernel. This behavior is especially useful with interrupt handlers, where the hardware interrupt must be managed as quickly as possible, but most of the data management can be safely delayed to a later time. Actually, a tasklet, just like a kernel timer, is executed (in atomic mode) in the context of a "soft interrupt," a kernel mechanism that executes asynchronous tasks with hardware interrupts enabled.

A tasklet exists as a data structure that must be initialized before use. Initialization can be performed by calling a specific function or by declaring the structure using certain macros:

```
#include <linux/interrupt.h>

struct tasklet_struct {
      /* ... */
      void (*func)(unsigned long);
      unsigned long data;
};

void tasklet_init(struct tasklet_struct *t,
      void (*func)(unsigned long), unsigned long data);
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data);
```

Tasklets offer a number of interesting features:

- A tasklet can be disabled and re-enabled later; it won't be executed until it is enabled as many times as it has been disabled.
- Just like timers, a tasklet can reregister itself.
- A tasklet can be scheduled to execute at normal priority or high priority. The latter group is always executed first.
- Tasklets may be run immediately if the system is not under heavy load but never later than the next timer tick.
- A tasklets can be concurrent with other tasklets but is strictly serialized with respect to itself—the same tasklet never runs simultaneously on more than one processor. Also, as already noted, a tasklet always runs on the same CPU that schedules it.

The *jit* module includes two files, */proc/jitasklet* and */proc/jitasklethi*, that return the same data as */proc/jitimer*, introduced in the section "Kernel Timers." When you read one of the files, you get back a header and six data lines. The first data line describes the context of the calling process, and the other lines describe the context of successive runs of a tasklet procedure. This is a sample run while compiling a kernel:

```
phon% cat /proc/jitasklet
   time   delta  inirq    pid   cpu command
  6076139    0     0      4370   0   cat
  6076140    1     1      4368   0   cc1
  6076141    1     1      4368   0   cc1
  6076141    0     1         2   0   ksoftirqd/0
  6076141    0     1         2   0   ksoftirqd/0
  6076141    0     1         2   0   ksoftirqd/0
```

As confirmed by the above data, the tasklet is run at the next timer tick as long as the CPU is busy running a process, but it is run immediately when the CPU is otherwise idle. The kernel provides a set of *ksoftirqd* kernel threads, one per CPU, just to run "soft interrupt" handlers, such as the *tasklet_action* function. Thus, the final three runs of the tasklet take place in the context of the *ksoftirqd* kernel thread associated to CPU `0`. The *jitasklethi* implementation uses a high-priority tasklet, explained in an upcoming list of functions.

The actual code in *jit* that implements */proc/jitasklet* and */proc/jitasklethi* is almost identical to the code that implements */proc/jitimer*, but it uses the tasklet calls instead of the timer ones. The following list lays out in detail the kernel interface to tasklets after the tasklet structure has been initialized:

`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 is deferred until the tasklet has been enabled again. If the tasklet is currently running, this function busy-waits until the tasklet exits; thus, after calling *tasklet_disable*, you can be sure that the tasklet is not running anywhere in the system.

`void tasklet_disable_nosync(struct tasklet_struct *t);`

Disable the tasklet, but without waiting for any currently-running function to exit. When it returns, the tasklet is disabled and won't be scheduled in the future until re-enabled, but it may be still running on another CPU when the function returns.

`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. A call to *tasklet_enable* must match each call to *tasklet_disable*, as the kernel keeps track of the "disable count" for each tasklet.

`void tasklet_schedule(struct tasklet_struct *t);`

Schedule the tasklet for execution. If a tasklet is scheduled again before it has a chance to run, it runs only once. However, if it is scheduled *while* it runs, it runs again after it completes; this ensures that events occurring while other events are being processed receive due attention. This behavior also allows a tasklet to reschedule itself.

`void tasklet_hi_schedule(struct tasklet_struct *t);`

Schedule the tasklet for execution with higher priority. When the soft interrupt handler runs, it deals with high-priority tasklets before other soft interrupt tasks, including "normal" tasklets. Ideally, only tasks with low-latency requirements (such as filling the audio buffer) should use this function, to avoid the additional latencies introduced by other soft interrupt handlers. Actually, */proc/jitasklethi* shows no human-visible difference from */proc/jitasklet*.

`void tasklet_kill(struct tasklet_struct *t);`

This function ensures that the tasklet is not scheduled to run again; it is usually called when a device is being closed or the module removed. If the tasklet is scheduled to run, the function waits until it has executed. If the tasklet reschedules itself, you must prevent it from rescheduling itself before calling *tasklet_kill*, as with *del_timer_sync*.

Tasklets are implemented in *kernel/softirq.c*. The two tasklet lists (normal and high-priority) are declared as per-CPU data structures, using the same CPU-affinity mechanism used by kernel timers. The data structure used in tasklet management is a simple linked list, because tasklets have none of the sorting requirements of kernel timers.

Workqueues

Workqueues are, superficially, similar to tasklets; they allow kernel code to request that a function be called at some future time. There are, however, some significant differences between the two, including:

- Tasklets run in software interrupt context with the result that all tasklet code must be atomic. Instead, workqueue functions run in the context of a special kernel process; as a result, they have more flexibility. In particular, workqueue functions can sleep.
- Tasklets always run on the processor from which they were originally submitted. Workqueues work in the same way, by default.
- Kernel code can request that the execution of workqueue functions be delayed for an explicit interval.

The key difference between the two is that tasklets execute quickly, for a short period of time, and in atomic mode, while workqueue functions may have higher latency but need not be atomic. Each mechanism has situations where it is appropriate.

Workqueues have a type of `struct workqueue_struct`, which is defined in *<linux/workqueue.h>*. A workqueue must be explicitly created before use, using one of the following two functions:

```
struct workqueue_struct *create_workqueue(const char *name);
struct workqueue_struct *create_singlethread_workqueue(const char *name);
```

Each workqueue has one or more dedicated processes ("kernel threads"), which run functions submitted to the queue. If you use *create_workqueue*, you get a workqueue that has a dedicated thread for each processor on the system. In many cases, all those threads are simply overkill; if a single worker thread will suffice, create the workqueue with *create_singlethread_workqueue* instead.

To submit a task to a workqueue, you need to fill in a `work_struct` structure. This can be done at compile time as follows:

```
DECLARE_WORK(name, void (*function)(void *), void *data);
```

Where `name` is the name of the structure to be declared, `function` is the function that is to be called from the workqueue, and `data` is a value to pass to that function. If you need to set up the `work_struct` structure at runtime, use the following two macros:

```
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

INIT_WORK does a more thorough job of initializing the structure; you should use it the first time that structure is set up. *PREPARE_WORK* does almost the same job, but it does not initialize the pointers used to link the `work_struct` structure into the workqueue. If there is any possibility that the structure may currently be submitted to a workqueue, and you need to change that structure, use *PREPARE_WORK* rather than *INIT_WORK*.

There are two functions for submitting work to a workqueue:

```
int queue_work(struct workqueue_struct *queue, struct work_struct *work);
int queue_delayed_work(struct workqueue_struct *queue,
                       struct work_struct *work, unsigned long delay);
```

Either one adds `work` to the given `queue`. If *queue_delayed_work* is used, however, the actual work is not performed until at least `delay` jiffies have passed. The return value from these functions is `0` if the work was successfully added to the queue; a nonzero result means that this `work_struct` structure was already waiting in the queue, and was not added a second time.

At some time in the future, the work function will be called with the given `data` value. The function will be running in the context of the worker thread, so it can sleep if need be—although you should be aware of how that sleep might affect any other tasks submitted to the same workqueue. What the function cannot do, however, is access user space. Since it is running inside a kernel thread, there simply is no user space to access.

Should you need to cancel a pending workqueue entry, you may call:

```
int cancel_delayed_work(struct work_struct *work);
```

The return value is nonzero if the entry was canceled before it began execution. The kernel guarantees that execution of the given entry will not be initiated after a call to *cancel_delayed_work*. If *cancel_delayed_work* returns `0`, however, the entry may have already been running on a different processor, and might still be running after a call to *cancel_delayed_work*. To be absolutely sure that the work function is not running anywhere in the system after *cancel_delayed_work* returns `0`, you must follow that call with a call to:

```
void flush_workqueue(struct workqueue_struct *queue);
```

After *flush_workqueue* returns, no work function submitted prior to the call is running anywhere in the system.

When you are done with a workqueue, you can get rid of it with:

```
void destroy_workqueue(struct workqueue_struct *queue);
```

The Shared Queue

A device driver, in many cases, does not need its own workqueue. If you only submit tasks to the queue occasionally, it may be more efficient to simply use the shared, default workqueue that is provided by the kernel. If you use this queue, however, you must be aware that you will be sharing it with others. Among other things, that means that you should not monopolize the queue for long periods of time (no long sleeps), and it may take longer for your tasks to get their turn in the processor.

The *jiq* (“just in queue”) module exports two files that demonstrate the use of the shared workqueue. They use a single `work_struct` structure, which is set up this way:

```
static struct work_struct jiq_work;

    /* this line is in jiq_init( ) */
    INIT_WORK(&jiq_work, jiq_print_wq, &jiq_data);
```

When a process reads */proc/jiqwq*, the module initiates a series of trips through the shared workqueue with no delay. The function it uses is:

```
int schedule_work(struct work_struct *work);
```

Note that a different function is used when working with the shared queue; it requires only the `work_struct` structure for an argument. The actual code in *jiq* looks like this:

```
prepare_to_wait(&jiq_wait, &wait, TASK_INTERRUPTIBLE);
schedule_work(&jiq_work);
schedule( );
finish_wait(&jiq_wait, &wait);
```

The actual work function prints out a line just like the *jit* module does, then, if need be, resubmits the `work_struct` structure into the workqueue. Here is *jiq_print_wq* in its entirety:

```
static void jiq_print_wq(void *ptr)
{
    struct clientdata *data = (struct clientdata *) ptr;

    if (! jiq_print (ptr))
        return;

    if (data->delay)
        schedule_delayed_work(&jiq_work, data->delay);
    else
        schedule_work(&jiq_work);
}
```

If the user is reading the delayed device (*/proc/jiqwqdelay*), the work function resubmits itself in the delayed mode with *schedule_delayed_work*:

```
int schedule_delayed_work(struct work_struct *work, unsigned long delay);
```

If you look at the output from these two devices, it looks something like:

```
% cat /proc/jiqwq
    time  delta preempt   pid cpu command
  1113043     0       0     7   1 events/1
  1113043     0       0     7   1 events/1
  1113043     0       0     7   1 events/1
  1113043     0       0     7   1 events/1
  1113043     0       0     7   1 events/1
% cat /proc/jiqwqdelay
    time  delta preempt   pid cpu command
  1122066     1       0     6   0 events/0
```

```
    1122067     1       0     6   0 events/0
    1122068     1       0     6   0 events/0
    1122069     1       0     6   0 events/0
    1122070     1       0     6   0 events/0
```

When */proc/jiqwq* is read, there is no obvious delay between the printing of each line. When, instead, */proc/jiqwqdelay* is read, there is a delay of exactly one jiffy between each line. In either case, we see the same process name printed; it is the name of the kernel thread that implements the shared workqueue. The CPU number is printed after the slash; we never know which CPU will be running when the */proc* file is read, but the work function will always run on the same processor thereafter.

If you need to cancel a work entry submitted to the shared queue, you may use *cancel_delayed_work*, as described above. Flushing the shared workqueue requires a separate function, however:

```
void flush_scheduled_work(void);
```

Since you do not know who else might be using this queue, you never really know how long it might take for *flush_scheduled_work* to return.

Quick Reference

This chapter introduced the following symbols.

Timekeeping

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

`#include <linux/jiffies.h>`
`volatile unsigned long jiffies`
`u64 jiffies_64`
: The `jiffies_64` variable is incremented once for each clock tick; thus, it's incremented `HZ` times per second. Kernel code most often refers to `jiffies`, which is the same as `jiffies_64` on 64-bit platforms and the least significant half of it on 32-bit platforms.

`int time_after(unsigned long a, unsigned long b);`
`int time_before(unsigned long a, unsigned long b);`
`int time_after_eq(unsigned long a, unsigned long b);`
`int time_before_eq(unsigned long a, unsigned long b);`
: These Boolean expressions compare jiffies in a safe way, without problems in case of counter overflow and without the need to access `jiffies_64`.

`u64 get_jiffies_64(void);`
: Retrieves `jiffies_64` without race conditions.

```
#include <linux/time.h>
unsigned long timespec_to_jiffies(struct timespec *value);
void jiffies_to_timespec(unsigned long jiffies, struct timespec *value);
unsigned long timeval_to_jiffies(struct timeval *value);
void jiffies_to_timeval(unsigned long jiffies, struct timeval *value);
```

Converts time representations between jiffies and other representations.

```
#include <asm/msr.h>
rdtsc(low32,high32);
rdtscl(low32);
rdtscll(var32);
```

x86-specific macros to read the timestamp counter. They read it as two 32-bit halves, read only the lower half, or read all of it into a `long long` variable.

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

Returns the timestamp counter in a platform-independent way. If the CPU offers no timestamp feature, `0` is returned.

```
#include <linux/time.h>
unsigned long mktime(year, mon, day, h, m, s);
```

Returns the number of seconds since the Epoch, based on the six `unsigned int` arguments.

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

Returns the current time, as seconds and microseconds since the Epoch, with the best resolution the hardware can offer. On most platforms the resolution is one microsecond or better, although some platforms offer only jiffies resolution.

```
struct timespec current_kernel_time(void);
```

Returns the current time with the resolution of one jiffy.

Delays

```
#include <linux/wait.h>
long wait_event_interruptible_timeout(wait_queue_head_t *q, condition, signed
  long timeout);
```

Puts the current process to sleep on the wait queue, installing a timeout value expressed in jiffies. Use *schedule_timeout* (below) for noninterruptible sleeps.

```
#include <linux/sched.h>
signed long schedule_timeout(signed long timeout);
```

Calls the scheduler after ensuring that the current process is awakened at timeout expiration. The caller must invoke *set_current_state* first to put itself in an interruptible or noninterruptible sleep state.

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

Introduces delays of an integer number of nanoseconds, microseconds, and milliseconds. The delay achieved is at least the requested value, but it can be more. The argument to each function must not exceed a platform-specific limit (usually a few thousands).

`void msleep(unsigned int millisecs);`
`unsigned long msleep_interruptible(unsigned int millisecs);`
`void ssleep(unsigned int seconds);`

Puts the process to sleep for the given number of milliseconds (or seconds, in the case of *ssleep*).

Kernel Timers

`#include <asm/hardirq.h>`
`int in_interrupt(void);`
`int in_atomic(void);`

Returns a Boolean value telling whether the calling code is executing in interrupt context or atomic context. Interrupt context is outside of a process context, either during hardware or software interrupt processing. Atomic context is when you can't schedule either an interrupt context or a process's context with a spinlock held.

`#include <linux/timer.h>`
`void init_timer(struct timer_list * timer);`
`struct timer_list TIMER_INITIALIZER(_function, _expires, _data);`

This function and the static declaration of the timer structure are the two ways to initialize a `timer_list` data structure.

`void add_timer(struct timer_list * timer);`

Registers the timer structure to run on the current CPU.

`int mod_timer(struct timer_list *timer, unsigned long expires);`

Changes the expiration time of an already scheduled timer structure. It can also act as an alternative to *add_timer*.

`int timer_pending(struct timer_list * timer);`

Macro that returns a Boolean value stating whether the timer structure is already registered to run.

`void del_timer(struct timer_list * timer);`
`void del_timer_sync(struct timer_list * timer);`

Removes a timer from the list of active timers. The latter function ensures that the timer is not currently running on another CPU.

Tasklets

```
#include <linux/interrupt.h>
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data);
void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long),
  unsigned long data);
```

The first two macros declare a tasklet structure, while the *tasklet_init* function initializes a tasklet structure that has been obtained by allocation or other means. The second `DECLARE` macro marks the tasklet as disabled.

```
void tasklet_disable(struct tasklet_struct *t);
void tasklet_disable_nosync(struct tasklet_struct *t);
void tasklet_enable(struct tasklet_struct *t);
```

Disables and reenables a tasklet. Each *disable* must be matched with an *enable* (you can disable the tasklet even if it's already disabled). The function *tasklet_disable* waits for the tasklet to terminate if it is running on another CPU. The *nosync* version doesn't take this extra step.

```
void tasklet_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule(struct tasklet_struct *t);
```

Schedules a tasklet to run, either as a "normal" tasklet or a high-priority one. When soft interrupts are executed, high-priority tasklets are dealt with first, while normal tasklets run last.

```
void tasklet_kill(struct tasklet_struct *t);
```

Removes the tasklet from the list of active ones, if it's scheduled to run. Like *tasklet_disable*, the function may block on SMP systems waiting for the tasklet to terminate if it's currently running on another CPU.

Workqueues

```
#include <linux/workqueue.h>
struct workqueue_struct;
struct work_struct;
```

The structures representing a workqueue and a work entry, respectively.

```
struct workqueue_struct *create_workqueue(const char *name);
struct workqueue_struct *create_singlethread_workqueue(const char *name);
void destroy_workqueue(struct workqueue_struct *queue);
```

Functions for creating and destroying workqueues. A call to *create_workqueue* creates a queue with a worker thread on each processor in the system; instead, *create_singlethread_workqueue* creates a workqueue with a single worker process.

```
DECLARE_WORK(name, void (*function)(void *), void *data);
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

Macros that declare and initialize workqueue entries.

```
int queue_work(struct workqueue_struct *queue, struct work_struct *work);
int queue_delayed_work(struct workqueue_struct *queue, struct work_struct
  *work, unsigned long delay);
```

Functions that queue work for execution from a workqueue.

```
int cancel_delayed_work(struct work_struct *work);
void flush_workqueue(struct workqueue_struct *queue);
```

Use *cancel_delayed_work* to remove an entry from a workqueue; *flush_workqueue* ensures that no workqueue entries are running anywhere in the system.

```
int schedule_work(struct work_struct *work);
int schedule_delayed_work(struct work_struct *work, unsigned long delay);
void flush_scheduled_work(void);
```

Functions for working with the shared workqueue.