Getting and setting the realtime clock and more

Apart from using timers, you can also get and set the current realtime clock, and adjust it gradually. The following functions can be used for these purposes:

Function	Type?	Description
ClockAdjust()	Neutrino	Gradually adjust the time
ClockCycles()	Neutrino	High-resolution snapshot
clock_getres()	POSIX	Fetch base timing resolution
clock_gettime()	POSIX	Get current time of day
ClockPeriod()	Neutrino	Get/set base timing resolution
clock_settime()	POSIX	Set current time of day
ClockTime()	Neutrino	Get/set current time of day

Getting and setting

The functions *clock_gettime()* and *clock_settime()* are the POSIX functions based on the kernel function *ClockTime()*. These functions can be used to get or set the current time of day. Unfortunately, setting this is a "hard" adjustment, meaning that whatever time you specify in the buffer is immediately taken as the *current* time. This can have startling consequences, especially when time appears to move "backwards" because the time was ahead of the "real" time. Generally, setting a clock using this method should be done only during power up or when the time is very much out of synchronization with the real time.

That said, to effect a gradual change in the current time, the function *ClockAdjust()* can be used:

```
int
ClockAdjust (clockid_t id,
             const struct _clockadjust *new,
             const struct _clockadjust *old);
```

The parameters are the clock source (always use CLOCK_REALTIME), and a *new* and *old* parameter. Both the *new* and *old* parameters are optional, and can be NULL. The *old* parameter simply returns the current adjustment. The operation of the clock adjustment is controlled through the *new* parameter, which is a pointer to a structure that contains two elements, *tick_nsec_inc* and *tick_count*. Basically, the operation of *ClockAdjust()* is very simple. Over the next *tick_count* clock ticks, the adjustment contained in *tick_nsec_inc* is added to the current system clock. This means that to move the time forward (to "catch up" with the real time), you'd specify a positive value for *tick_nsec_inc*. Note that you'd never move the time backwards! Instead, if your clock was too fast, you'd specify a small negative number to *tick_nsec_inc*, which would cause the current time to not advance as fast as it would. So effectively, you've slowed down the clock until it matches reality. A rule of thumb is that you shouldn't adjust the clock by more than 10% of the base timing

resolution of your system (as indicated by the functions we'll talk about next, *ClockPeriod()* and friends).

Adjusting the timebase

As we've been saying throughout this chapter, the timing resolution of everything in the system is going to be *no more accurate than* the base timing resolution coming into the system. So the obvious question is, how do you set the base timing resolution? You can use the following function for this:

```
int
ClockPeriod (clockid_t id,
             const struct _clockperiod *new,
             struct _clockperiod *old,
             int reserved);
```

As with the *ClockAdjust()* function described above, the *new* and the *old* parameters are how you get and/or set the values of the base timing resolution. The *new* and *old* parameters are pointers to structures of `struct _clockperiod`, which contains two members, *nsec* and *fract*. Currently, the *fract* member must be set to zero (it's the number of femtoseconds; we probably won't use this kind of resolution for a little while yet!) The *nsec* member indicates how many nanoseconds elapse between ticks of the base timing clock. The default is 10 milliseconds, so the *nsec* member (if you use the "get" form of the call by specifying the *old* parameter) will show approximately 10 million nanoseconds. (As we discussed above, in "Clock interrupt sources," it's not going to be *exactly* 10 milliseconds.)

While you can certainly feel free to try to set the base timing resolution on your system to something ridiculously small, the kernel will step in and prevent you from doing that. Generally, you can set most systems in the 1 millisecond to hundreds of microseconds range.