QNX Timer Tutorial

Modeling of Real-Time Systems
4010-462-01

Joe Richardson

Introduction

The QNX libraries provide mechanisms called timers. These timers are used to execute a task using the passage of time as its trigger. Timers can be either periodic or single execution. There are a few different event types that a timer can act on. Those events are with signals, signal codes, threads, and pulses. This tutorial will focus on how to set up a timer that executes a thread when it the timer times out. This is because there a few key details missing in the documentation for thread based timers. In this tutorial we will demonstrate how to use timers to control an LED on the Spartan 3 FPGA board from QNX running on the Diamond Systems Athena boards, purple boxes.

Wiring

The first step in this tutorial will be wiring the two devices together. This will be done via the data acquisition port of the Diamond Systems Athena board the FPGA A2 connector. The data acquisition port on the Diamond Systems Athena board is marked and looks like an IDE port. The A2 connector on the FPGA is the black rectangular series of pinouts directly above the seven segment display. Now the data acquisition port needs to be connected to a breakout board. This will make it much easier to wire the two devices together. When that is done use the table below to connect the FPGA to the breakout board.

Purpose	Data Acquisition port Pin	FPGA A2 Connector Pin
Common Ground	28	1
LED Number 4	17	16

***Note**: The pins used on the data acquisition port are specific to this tutorial and others can be used in general, except for the common ground.

Implementation

Implementation of the timers is actually fairly simple, once they are understood. For this tutorial let's start by defining a header for the class that will use the timer. Below is a section of that header file.

```
public:

        const static int DATA_BASE_ADDRESS = 0x280;
        const static int DATA_PORT_C = DATA_BASE_ADDRESS + 10;
        const static int DATA_DIRECTION = DATA_BASE_ADDRESS + 11;
        const static int IOCR_SETUP = 0x8; // 00001000 -- use low bits

private:

        struct itimerspec timerSpec;

        struct sigevent event;

        timer_t timerID;
```

The public section of the section shown above simply defines needed constants. The fist is just the base address for using the data acquisition port of the Diamond Systems Athena board. Next we define where port c is relative to the base address. We will be using this port to communicate with the FPGA LED. Then we define the address for the direction. This will be used to notify the Diamond Systems Athena board if we are reading or writing to the data acquisition port. Finally, there is a variable called IOCR_SETUP. This variable is going to be used to specify that we are using the low bits of port C. This is because port C is special and allows us to split the port into two separate sections, high bits and low bits. In this tutorial we will be using the low bits.

In the private section of the header code snippet there are three variables shown. The first variable is a struct that will be used to define the parameters of the timer. This struct has four values; it_value.tv_sec, it_value.tv_nsec, it_interval.tv_sec, and it_interval.tv_nsec. The first two values, "it_value.*", are for the amount of time to pass before the timer will expire. The first in a seconds value and the second is a nanoseconds value. The second two values, "it_interval.*", are used to define the period or frequency of this timer. Again the first value is in seconds and the second is in nanoseconds. An important thing to note is that the interval values need to be set to zero for a single execution timer. The second variable in the header is a struct for the signal event. This will be used to tell the system what to do when the timer expires. Finally, there is a variable that will hold the id of the timer that will be created. This id is much like a process id in an operating system, in which ids are available in pool and are assigned upon request and are returned to the pool when the process terminated.

With those basics and a header file we can now move on to implementing the class in a cpp file. Below is a section of that implementation that is used for controlling the LED we will be using to demonstrate a timer.

```
void TimerTest::startLED() {
	int privity_err = ThreadCtl(_NTO_TCTL_IO, NULL);

	if(privity_err != -1) {
		direction_handle = mmap_device_io(1, DATA_DIRECTION);
		led_handle = mmap_device_io(1, DATA_PORT_C);
	}

	active = true;
	pthread_create(&(this->thread), NULL, &TimerTest::runThread, this);
}

void* TimerTest::runThread(void* arg) {
	TimerTest* timerObject = (TimerTest*)arg;

	while(timerObject->isActive()) {
		out8(timerObject->getDirectionHandle(), IOCR_SETUP);
		out8(timerObject->getLEDHandle(), timerObject->getLEDValue());

		delay(250);
	}

	return NULL;
}
```

The startLED method starts by attempting to give the current thread root access to the hardware. If that is successful a handle to the direction port and a handle to port C are created. Finally, it creates a thread that runs the runThread method. In the runThread method all that is done is the writing out of a value to the LED very quarter of a second until the thread is stopped.

Now that we have a thread polling a variable and writing that variable to the LED we can move on to the timer actual implementation of the timer. Below is another section of the class. This section will provide the details of how to use and implement the timer.

```
TimerTest::TimerTest() {
	active = false;
	ledValue = false;

	SIGEV_THREAD_INIT(&event, TimerTest::timerExpired, this, 0);

	timerID = -1;
}

void TimerTest::startTimer(int timeOutSec, int timeOutNsec, int periodSec, int
periodNsec) {
	if(timerID != -1)
		stopTimer();

	timerSpec.it_value.tv_sec = timeOutSec;
	timerSpec.it_value.tv_nsec = timeOutNsec;
	timerSpec.it_interval.tv_sec = periodSec;
	timerSpec.it_interval.tv_nsec = periodNsec;

	timer_create(CLOCK_REALTIME, &event, &timerID);
	timer_settime(timerID, 0, &timerSpec, NULL);
}

void TimerTest::stopTimer() {
	timer_delete(timerID);
	timerID = -1;
}

void TimerTest::timerExpired(sigval arg) {
	TimerTest* timerObject = (TimerTest*)arg.sival_ptr;
	timerObject->toggleLEDValue();
}
```

The first step in setting up the timer is found in the constructor. In this tutorial we will be using threads for our timers so the SIGEV_THREAD_INIT method. The first parameter to this method is the address of the signal event that will be used to store the results of this method call. In our case it will be the event we defined in the header. The second parameter is the address of the method to invoke when the timer expires. The third parameter is the argument that will be passed into the provided method. The final parameter is a code to be interpreted by the signal handler. This must be in the range from SI_MINAVAIL through SI_MAXAVAIL.

Next the timer has to be initialized. This is done in the startTimer method. The first step in this method is to set the values passed in to the method on the itimerspec struct defined in the header. This if followed by a call to timer_create. This method is responsible for creating a timer instance. This method is not safe to call from a cancellation point or from

an interrupt handler. To give an analogy for this method, consider it equivalent to a constructor call for the timer. The first parameter to method is a clock id. In QNX there are currently two clock ids, CLOCK_REALTIME and CLOCK_SOFTTIME. CLOCK_REALTIME used the standard POSIX-defined timer for the clock used to determine the passage of time. Currently, CLOCK_SOFTTIME is the same as CLOCK_REALTIME, but there seem to be plans to change this in the future. Since we are dealing with POSIX threads anyway we will us CLOCK_REALTIME. The next parameter is the address of the event to invoke when the timer expires, we will use the event setup in the constructor. The final parameter to this method is the address to store the timer that will be created. Now with an initialized timer we will need to tell the system to turn on the timer. This is done by call to the timer_settime method, note that this method is also not safe to call from a cancellation point or from an interrupt handler. The first parameter to this method is the id of the timer to start. The second is a flag that describes the type of timer that is being armed. We will use zero to represent that this is relative timer, meaning the expatriation is relative to the time at the moment this method is called. The third parameter is the time specification we created in the beginning of the method. The final parameter is another time specification struct. This can be NULL, or a pointer to a `itimerspec` structure that the function fills in with the timer's former time until expiry. In this tutorial this parameter will be NULL since we do not care about the timer's former time value. Now the timer is running an will expire and rerun based on the time specification we created.

The stopTimer method is a convenience method that is provided. This method obviously is responsible for ending a timer without executing its time out signal event or rerunning it. This is done by calling the delete_timer function and passing it the timer id to delete. Again this is a method that must not be called from a cancellation point or from an interrupt handler.

The final method, timerExpired, is the method that will be called by the timer when it times out. For this tutorial the method will simply toggle the value to the LED. This means that each time the timer expires the LED will turn off if it was on and turn on if it was off.

Conclusion

Setting up a timer using the defined QNX libraries is a very simple task. That is once all the needed methods are known and understood. From their documentation alone it is very hard to gain a clear understanding without a lot of trial and error. This tutorial should provide you with the necessary understanding to avoid a good deal of this experimentation. The full source files for this tutorial and a simple application allowing the timer class to be tested have been provide to a fuller understanding of this tutorial.