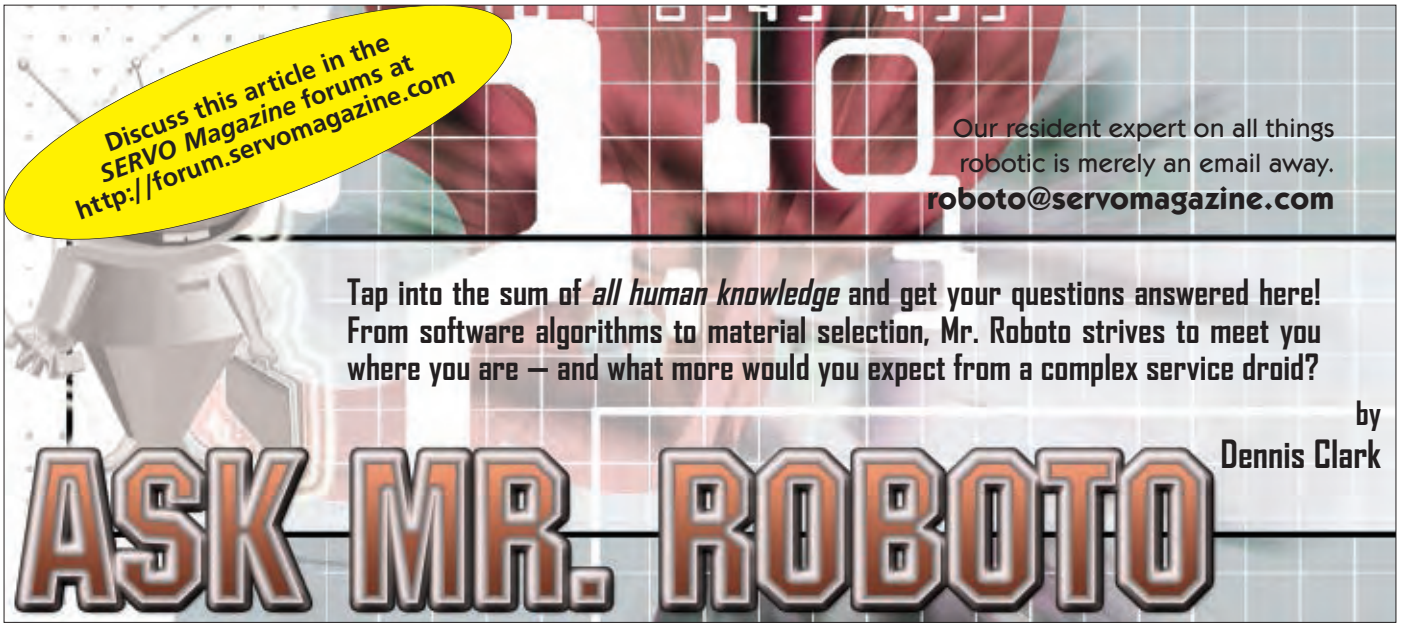

This month's column is dedicated to software issues that many of you are struggling with, attempting to go around, or are just plain avoiding by watching another edition of The Daily Show, rather than wrestling a problem to the ground and pinning it.

So, this time I'm going to answer these software questions that I've been queuing up for just such an occasion. I've chosen timers and Interrupt Service Routines (ISR) for my topics, using as many compilers as I can on the two most commonly used platforms I get questions about: the Atmel ATmega and PIC series chips. Contrary to popular belief, Mr. Roboto has not used every compiler or chip extensively, just some here and there. Rather than repeat every question and answer, I'll break the questions down into two categories and go over the nuts and bolts of how to solve the problems.

You will have to bear with me here. Working with timers and interrupts is rather like tying your shoelaces. Once you are "in the know," the process seems simple and is automatic, but explaining just exactly how and why one is doing each step is very laborious. So, I will start at "level 0" and work forward. I hope that you will then understand and be able to apply the process to your own project, and not just have to copy the code you find in these pages.

Q. So, let's get started with our first generically posed question. How do I set up a timer?

A. I'll start with the always popular ATmega168 or ATmega328 parts. These are commonly used on a variety of popular robot controllers like the Pololu Orangutan and the Arduino series. I use them pretty often myself. So, how do you set up a timer? Let's take a look. I promise, it won't hurt and after you do it a few times, it will be automatic to you.

The Compiler

I'm going to use the avr-gcc as my compiler of choice. It is a good compiler, runs on Windows, Mac OSX, and Linux, and it's free. (What more could you want?) You don't have that? Well, go back through your *SERVO Magazine* archives and find where I detail how to set it up, either using *Winavr* or *Eclipse* (my choice) in the August '08 article.

Homework

I am not going to reproduce the entire 376 page document Atmel provides for us to configure and use the ATmega168/328 parts. You should get a copy of it from your favorite parts supplier or Atmel (**www.Atmel.com**) to use as a reference while you read this article. Another good place to get the ATmega168 datasheet is your parts distributor; it will be easier to find there. I use Digi-Key (**www.digikey.com**). Simply search for the ATmega168 and click on the PDF symbol where the part is described.

The Timers

There are three timers in the ATmega168/328 parts: TMR0 and TMR2 (eight-bit), and TMR2 (16-bit). All of these timers can do PWM, or they can be configured to just be timers. TMR2 can also be configured as an *Input Capture*, but that is another story. I'll look at TMR2 for this discussion.

While all of the timers can take either an I/O pin input or use the system clock to advance the timer, TMR2 can also use an external crystal, so this one can be used with a 32.768 KHz source to run a real time clock. We're not going there this time. We'll select the system clock, which in the case of an Arduino is typically 16 MHz. So, I'll just use 16 MHz as our clock source. If you use another system frequency, substitute that frequency in when doing the timing calculations that I'll show you.

The Timer Registers

To set a timer, you need to fiddle with several registers. The following registers set up the clock source, pre-scaling, and what the timer is used for. Each timer will have its own unique set of registers. They will all use this naming convention; just change the number to match the timer/counter number.

TCCR2A: Timer/Counter Control Register A. Configures PWM configuration on OC2A/OC2B pins, if we use them for PWM.

TCCR2B: Timer/Counter Control Register B. Configures Timer 2 pre-scale settings and a little more of the PWM settings, if used.

TCNT2: Timer/Counter Register. This holds the current timer/counter value.

ASSR: Asynchronous Status Register. Allows selection of external clock sources.

These next registers are used to match PWM periods — more on that later; they aren't needed if you are just using the timer.

OCR2A: Output Compare Register 2A. Match value for PWM on OC2A pin.

OCR2B: Output Compare Register 2B. Match value for PWM on OC2B pin.

This next set of registers allow us to set interrupts on the timer conditions and check for interrupt flags.

TIMSK2: Timer/Counter Interrupt Mask Register. Configures the type of interrupt (if any) you want to have on PWM or Timer conditions.

TIFR2: Timer/Counter Interrupt Flag Register. We can look here for flags set by configured interrupts.

Setting Up a Timer

We are going to start by just setting up a timer — not PWM — so these settings will reflect that. **Listing 1** shows the init routine. I'll comment on what was done.

1). Always clear your flags before you turn something on, especially interrupts.

2). Have the timer match on the OC2A setting. Table 17-8, mode 2, CTC is the counter/timer clock match on OCRA. When this match occurs, the clock is cleared back to zero to start over.

3). Use the internal clock with no prescale. Table 17-9 shows the prescale settings. Use other settings if you want a slower clock.

4). Clear the timer to zero; start from a known place.

5). Here we set the interrupt. A timer just running by itself is useful for some things; don't use interrupts if you don't need them. Here we will use the timer match to fire an interrupt (more on that later). I've shown two ways to set a bit in a register.

6). 160 = 10 µs (microseconds) is the timer match with a 16 MHz system clock. I did this because it was a good value to use for controlling an RC servo pulse.

By now you're thinking, "Why use an interrupt?" There are good reasons for your timer to cause an interrupt. One is to provide a background *ticker* for timing your program. I find that such a timer is handy for timing state machines and checking for timeouts in various places in the program. Every time the interrupt goes off, you can increment a large counter in your program. In this way, you can check timing for many things without blocking program flow. **Listing 2** shows what the ISR for just such a timer might look like.

My ISR does two things: it controls the position of an RC servo (match count), and it handles my system timer tic (1_1ms) at a 1 ms resolution. Handy, isn't it? If all you

Listing 1: Setting Up the Timer.

```
TIFR2 = 0;                                  // (#1)
TCCR2A = 0x02;                              // (#2)
TCCR2B = 0x01;                              // (#3)
TCNT2  = 0;                                 // (#4)
TIMSK2 |= (unsigned char)_BV(OCIE2A);       // (#5)
TIMSK2 |= (unsigned char)(1 <<OCIE2A);      // (#5)
OCR2A = 160;                                // (#6)
```

Listing 2: Timer ISR.

```
ISR(TIMER2_COMPA_vect)
/*
 * 10 microsecond ISR on TIMER2, a 8 bit clock
 */
{
        static uint8_t        t_10us;
        static uint16_t       match;        // The single servo used
        static uint8_t        next_pulse;

t_10us++;
        if (t_10us == 100)                  // 1ms background tic
        {
                t_1ms++;
                t_10us = 0;
        }

        match++;                            // increment every 1
        if (match > next_pulse)
                SPIN = 0;                   // drop servo bit
        else
                SPIN = 1;                   // raise servo pin high

        if (match > 1900)
        {
                next_pulse = servo;
                match = 0;                  // restart servo timer
        }
}
```

Listing 3: Setting Up PWM.

```
void InitMotors(void)
/*
 * Set up the motors and PWM and such.
 */
{
        DDRD |= (unsigned char)(1 <<PD5);   // (#1)
        DDRD |= (unsigned char)(1 <<PD6);

        TCCR0A = 0xF3;                      // (#2)
        TCCR0B = 0x03;                      // (#3)

        OCR0A = 0;                          // (#4)
        OCR0B = 0;
}
```

needed to do was keep a 1 ms background tic, you could set up the timer to interrupt at a 1 ms rate by choosing a pre-scale value and match count that would cause the interrupt every 1 ms. (Hint: 16 MHz/128 * 125 = 1 ms).

That wasn't so hard, was it? The most difficult part of this process is probably finding out how avr-gcc wants you do set up interrupts (which I also showed you). But how did I find the name of the interrupt so it would call the correct ISR? Okay, you got me. That was *really* hard to find, so I went to good ol' Google, and it found **www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html** which set me right up. Somewhere in the avr-gcc documentation it says this. I was sure, but I didn't find it without Google! At least now *you* don't have to search for it.

Using a Timer for PWM

Our first step has nothing to do with timers at all! We're going to turn off the *comparator module* on the chip. This module deals with comparing analog voltages and causing things to happen by comparing them. This module defaults to on and will cause you no end of headaches with your I/O if you don't disable it!

```
ACSR = 0x80;          // turn off the comparator
```

This time, I'll pick TMR0 for our PWM. In this series of microcontrollers, a PWM block typically controls two PWM output pins. In this case, our PWM outputs will be called OC0A and OC0B, for *Output Compare Zero A* and *Output Compare Zero B*. We'll use similarly named registers in a different way. In fact, we could just as easily have used TMR2 or even TMR1 for PWM, but I thought that I'd change it up a little. Yes, this is just an eight-bit PWM, but for most of us, we use about three speeds: off, slow, and fast. Eight bits does that just fine.

Timer0 PWM Registers

TCCR0A: Timer/Counter Control Register A. Configures PWM configuration on OC0A/OC0B pins, if we use them for PWM.

TCCR0B: Timer/Counter Control Register B. Configures Timer 0 pre-scale settings and a little more of the PWM settings, if used.

OCR0A: Output Compare Register 2A. Match value for PWM on OC2A pin.

OCR0B: Output Compare Register 2B. Match value for PWM on OC2B pin.

DDRD: Data Direction Register PORTD. Set the direction of data transfer for the pins on port B. In this case, OC0A and OC0B are PD6 and PD5, respectively.

Setting Up the PWM

Above we disabled the comparator module. This assures us that our I/O pins are really digital. Now, we'll configure the registers above to PWM some motors. Since we're going to send a PWM signal out, we'll need to deal with the data direction registers, as well as the timer registers. **Listing 3** shows how to initialize all of the needed registers.

Set the DDRD register for outputs on PD5 (OC0B) and PD6 (OC0A). Set the fast PWM mode, clear the output on match, count up from the bottom (0). Fast PWM will only count up from 0, match (clear output), count the rest of the period, then restart at 0 again with the output set to 1. Set the prescale to 64 which on a 16 MHz part means the clock will be 250 kHz divided by 256 (an eight-bit count) which translates to a PWM rate of 977 Hz. Set both PWMs to 0. We should always start out stopped, don't you think?

At this point, all you need to set a motor speed is to put a non-zero value into OCR0A or OCR0B, and the motor will go that speed. This is *simple* motor speed control! If I was going to use a PID algorithm, I would use the PWMs on TMR1 which is 16 bits of resolution; this would give me a smoother PID loop with more control. For simple motor control, eight bits is plenty.

Wrap-Up for the ATmega168/328

Using these techniques, you can build your next robot with a background timer that you don't need to pay attention to — except to read it. I like to use a 32-bit variable and either a 1 ms or 10 ms tic; these will last a good long time. (Get out your calculators class! There will be a quiz: How long will it take for a 32-bit counter to turn over to zero when incremented every one millisecond?) You should do some defensive programming, however, to make sure you don't roll over to zero between when you take a time reading and wait for a certain period of time to pass. When dealing with PWM, it is even simpler than setting up a timer with an ISR. Now, go do it!

Next Time, PIC Microcontrollers

I'll be writing about a few different PIC micros and a couple of different compilers next month. PICs are different than the ATmega parts, but not more difficult to use. Well, we've come to the end of another Mr. Roboto! Keep those letters rolling in to roboto@servomagazine.com. SV