It’s About Time

Chuck Baird
http://www.avrfreaks.net, user name zbaird

Introduction

In the world of robotics and embedded programming there are many occasions when we need to wait to do (or while doing) something – for example, when flashing an LED, energizing a solenoid for a certain length of time, or delaying briefly while a servo repositions. At times we need accurate delays, but often we are dealing with situations where approximate times are good enough. This article will look at a few techniques, some accurate and some approximate, using single or multiple “virtual” timers, which may be adapted to fulfill a variety of timing needs.

Have three sets of servos that must move back and forth once every three seconds but each one is out of phase with the others by a second? Want to flash an LED 6 times, starting in 1 second, with a 0.8 second cycle time? Want to look around for the line to follow by alternating left or right turns in larger radii each time? No problem.

The following discussion will give an overview of each technique and how to use it. The accompanying files provide more detail, examples, and extensively commented C code that will more fully explain and demonstrate each method.

Built in delays

At the easy end of the timing spectrum, some compilers include macros or library functions that perform delays. You include an initial declaration which states your processor clock speed and hard code the delay:

```
#define F_CPU 8000000UL
...
_delay_ms(300);          // wait 300 milliseconds
```

Based on the two numbers the compiler constructs code sequences of the proper number of instructions to create the delay. Because the compiler controls the number of generated instructions, delays built in this way can be quite accurate. However, before using such tools be sure to read the fine print – the argument may need to be an actual number, not a variable. For variable delays you can put a “wrapper” function around a hard coded delay, using an argument to execute the delay a certain number of times:

```
void tenths_second(unsigned int sec10) {
   while (sec10--) _delay_ms(100);      // 0.1 second delay
}
```

You lose a tiny bit of accuracy because of the function overhead, but perhaps that is an acceptable compromise given the added flexibility and convenience it provides.

Easy delays the hard way

In the absence of a built in macro or function we can write simple code to do the same thing although the accuracy may suffer. Often we don't care about the exact timing of a delay, so this may be acceptable. If we aren't simultaneously needing to do anything else we can allow the delay to eat all the processor cycles without losing any performance. This is how the built in delays work.

One easy method of doing this is to write a function which just loops:

```
void wait(void) {         // may not work - see text
   unsigned int t;

   for (t = 0; t < 50000; t++) ;
}
```

Using it might look something like this:

```
flash = 10;               // flash 10 times
while (flash--) {         // loop until flash == 0
   turnLEDon();           // light up the LED
   wait();                // kill some time
   turnLEDoff();          // turn the LED off
   wait();                // kill some more time
}
```

When we call the function, it just counts up to some value and then returns. We can delay more by making the top count larger, or delay less by making it smaller. The length of time it takes depends on several things, most notably our processor's clock speed. While we could calculate or estimate the delay, in many cases determining the count value by trial and error suffices.

But beware – there is something potentially terribly wrong with this method of wasting time. If you use an optimizing compiler, there is a very good chance the compiler will notice that nothing useful happens in the loop, and actually remove it all together. Your wait() function will be rendered even more void that its type suggests, and when called will immediately return to the caller.

An easy way to force the optimizer to leave your code intact is to declare the loop variable volatile. The code examples give details for outsmarting your compiler.

If you need longer delays, greater than can be achieved with a single counter, you may increase its size (char → int → long) and/or use two or more nested loops. And you can always pass the counter value(s) to the function as arguments to allow variable length delays. These constructions are included in the examples.

Would you like interrupts with that?

The methods so far have simply "burned time" – the delay comes from the processor being completely busy using up the cycles. We can construct more elegant and versatile timing solutions by using timer interrupts.

Recall that interrupts are internally or externally generated signals that cause the processor to temporarily suspend its normal instruction flow and execute code called an interrupt handler or interrupt service routine (ISR). Timers are hardware peripherals within the MCU that can, among other things, independently generate single or ongoing interrupt signals at precise times.

Speaking of interrupts, if interrupts are enabled the delays mentioned so far can be rendered somewhat inaccurate by the amount of time spent in the ISRs. In effect we have been counting processor cycles, but responding to each interrupt adds some cycles which do not get counted. While it is possible to disable interrupts during the delays, that could potentially give rise to other problems.

Using timer interrupts takes care of this, because we will no longer delay by executing a certain number of instructions. We essentially start a separate clock, and when our desired time elapses we take the appropriate action. In the meantime, the processor is free to perform other useful work.

Hardware timers

Most MCUs include user programmable timers with several flexible features. Exactly what they do and how they are used depends on the specific MCU, and the exact syntax of the resulting code depends on the compiler.

We will consider two scenarios. In the first, we wait a specified amount of time, do something, and then we are done – we time a single interval. In the second, we let the timer generate an ongoing stream of regular interrupts and use them as a basis for timing a variety of things.

The first scenario is fairly straight forward. To start a delay we initialize a timer with our interval value and start it running. At the end of the interval an interrupt occurs and the processor calls the interrupt service routine. In that ISR we disable the timer, and either take care of what needs to happen or set a flag indicating the interval has ended. When we exit the ISR the original code sequence continues, and the timer is again available for our next use.

The second scenario is slightly more complicated but has greater potential, and the rest of this article explores three methods built around it. We use a continuous stream of one millisecond (1 KHz) interrupts that are started once with an initialization routine. That is an arbitrary interrupt frequency which could be any reasonable value, although you don't want interrupts so often that the interrupt overhead itself becomes an issue. You also certainly don't want to still be servicing one interrupt when the next one of the same type occurs. The choice of an appropriate frequency for your interrupts will depend on your application and the timing resolution you need.

As always, the accompanying files have more details, examples, and the code itself.

My clock is ticking – now what?

With the ability to automatically execute a function (the ISR) once every millisecond, let's look at a simple method of utilizing it. We declare a global unsigned int called pause, and declare it volatile. It may change inside the interrupt handler, and volatile tells the compiler to always use the current, not a cached, value. If you would like to spend some quality debugging time, omit the volatile qualifier.

```
volatile unsigned int pause;     // our global interrupt counter
```

Inside the interrupt handler, if pause is nonzero, we will decrement it. If it is zero, we will leave it alone. The actual syntax of the ISR definition will be compiler dependent.

```
#pragma interrupt_handler our_ISR:iv_TIM0_COMP
void our_ISR(void) {
   if (pause) pause--;
}
```

Finally we will write a function called iwait() which just waits for pause to go to zero. The optimizer will leave this code alone because pause is declared volatile.

```
void iwait(void) {
   while (pause) ;
}
```

And that's all there is to it. Other than the timer initialization it takes just these few lines of code to time about anything. To use this, we simply set pause equal to the number of milliseconds we wish to delay, and then call iwait().

```
pause = 100;    // one tenth of a second
iwait();        // wait for it to pass
```

Of course we could always combine them and pass the delay time as an argument:

```
void wait_ms(unsigned int t) {
   pause = t;                  // number of milliseconds
   while (pause) ;             // wait for Godot
}
```

However, there are a couple of problems with this approach. First, when we set our initial value for the variable pause, we don't know how soon thereafter the next interrupt is going to fire. It may be one clock cycle later, or it may be almost a full millisecond later. The first tic is going to be of an unpredictable length, and consequently our timing is going to be, on average, about half a tic (millisecond) off. A solution to this is given with the code files.

The other problem is that we are still simply burning time with the approach above, even though the timer is doing our counting. What if we have things to do, promises to keep?

Well, using the iwait() function and/or watching pause in a tight loop aren't the only ways to check for our interval being exhausted. We can always just use the variable pause as a flag:

```
pause = 200;       // set up delay time, as before
while (pause) {    // loop while doing all those useful things
   ...
   // some good stuff here
   ...
}
// come here when the interval is over
```

or:

```
pause = 500;               // do something in a half second
while (1) {                // loop forever
   if (pause) {
      // come here most of the time
   } else {
      // come here when each half second is up
      // (break out of the while loop if you wish)
      pause = 500;         // delay another half second
   }
}
```

To the future – and beyond, part one

Next we will consider a couple of more advanced methods which provide some useful flexibility – the management of multiple future events using a single timer. Details and examples are included with the source code.

The first approach will allow up to 8 "virtual" timers, each one based on our familiar 1 millisecond timer interrupt stream. Eight is a convenient size, although arbitrary.

We use a global variable containing 8 bit flags, one per virtual timer. When each timer is started, its bit is cleared, and when the timer runs out, its flag is set. An initialization routine starts the hardware timer and clears the bit flags.

To use one of the timers we call start_timer() with the timer number (0 to 7) and the number of milliseconds we want to delay before the timer expires and sets its flag. A tic count of 0 may be used to stop a running timer. The start_timer() function returns a value which is ANDed with the bit flags variable to check if the timer has elapsed.

We again have the problem about the variable length of time from starting a timer to the next interrupt. Unfortunately there is no easy fix this time – each of the eight timers can be started at any time within a tic interval, but all will be updated simultaneously on each interrupt. As a consequence, the first tic for each timer will be on average a half a tic width long, making all the timers off by a variable amount. Increasing the interrupt frequency will reduce the absolute error, but not remove its effect.

To the future – and beyond, part deux

Our second approach is to define and manage "events" which will happen at some time in the future. An event consists of a call to a function which we will write, and within it we can do whatever we like. We schedule an event by specifying the function name and when it is to execute. Limited only by array sizes, we can have any number of events outstanding, waiting to be executed when their time comes.

This method is again implemented by generating timer interrupts at the resolution we wish to use. We maintain a list of functions (events) to call and counters for the number of tics before we call each one. During each interrupt we decrement all the counters, and if any goes to zero, it is time to call its function.

There is potentially a serious problem calling the functions from within the ISR, since in general it is a bad idea to loiter too long in an interrupt handler. We take a better approach and implement a mechanism to call the functions as foreground tasks, not from within the ISR

Additionally, individual or all pending events may be cancelled if the need arises. And of course our old friend the initial tic latency problem is alive and well for each event.

Summary

We have examined a variety of timing techniques and some of their advantages and disadvantages, and constructed two general approaches that permit managing several timing needs simultaneously. With this foundation you can probably devise many other alternatives and variations that will serve all your timing needs.