Timer/Counter Notes

Andrew H. Fagg

Embedded systems often require mechanisms for counting the occurrence of events and for performing tasks at regular intervals. Embedded processors are often equipped with hardware support for this functionality. By providing this support, we can ensure that events are not missed (within limits, of course) and that timing of behavior occurs at regular intervals.

While the discussions here will often refer to specific hardware mechanisms provided by the Atmel MegaX family of 8-bit processors, the concepts are applicable to a range of embedded processors. Also note that the software examples will often make use of functionality provided by the "OUlib" library (see
http://www-symbiotic.cs.ou.edu/doc/oulib/html/modules.html for details).

1 Counters

Counters, as the name suggests, are hardware mechanisms for counting some form of event. At the heart of the counter is a *special purpose register* that stores the current value of the counter. Any time that a certain event occurs, the value of this counter is incremented (+1 is added to the value). The type of event that causes this increment is typically configurable through other special purpose registers.

Because the counter value is stored in a special purpose register, this implies that the value can also be read from or written to by the executing code. Also, different counters will store values of different sizes. The typical sizes for 8-bit microcontrollers, such as the Atmel MegaX line, are 8 and 16 bits (1 and 2 bytes). Because of this finite size, the counter can only count to a maximum value (255 for an 8-bit counter, and 65535 for a 16-bit counter). Once the counter reaches this maximum value, and a new event occurs, the counter resets back to zero. From here, the counter continues to increment with each event.

Counters are useful in many different contexts. For example, we may be interested in how far a wheel has turned in some given period of time. An *encoder* is a circular device that attaches to the axle of the wheel and has alternating opaque and transparent wedges

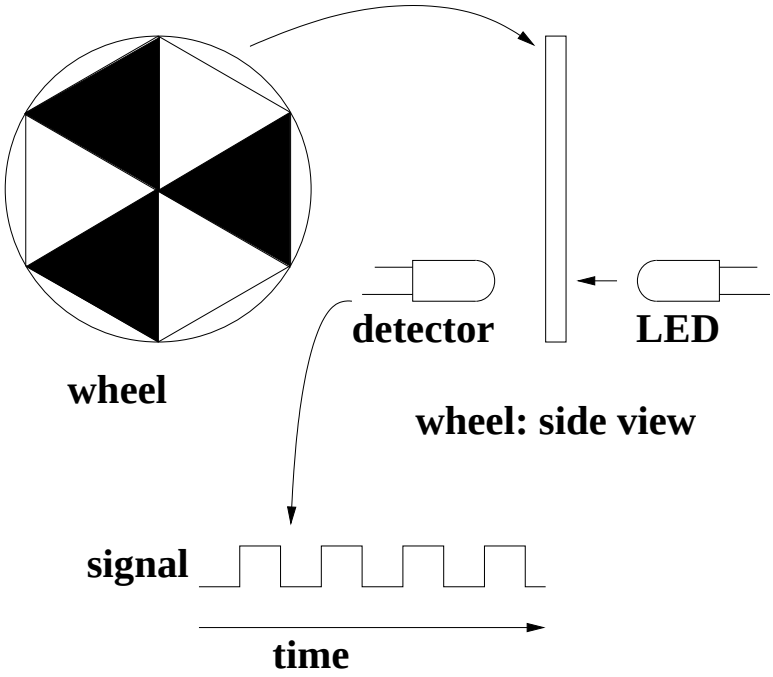

Figure 1: A wheel equipped with a simple encoder. The encoder wheel is partitioned into regions that are alternating opaque and transparent. When the wheel is turning at a constant velocity, the detector observes a digital signal of some frequency and a duty cycle of 50%.

at regular intervals (imagine partitioning a transparent circle into $n = 2^k$ wedges and filling every other one). An example is illustrated in Figure 1. An LED is placed on one side of the circle and a digital detector on the other. If the wheel is turning at a constant velocity, then the detector will "see" the LED exactly half of the time, while a transparent wedge sits between the LED and the detector. However, once an opaque wedge moves between them, the detector will no longer see the LED. Because LED is only visible half of the time, we say that the detector produces a signal with 50% *duty cycle*. Furthermore, as the speed of the wheel increases, the *frequency* of the signal will increase, but the duty cycle will remain the same.

A *counter* is useful in this context because it can, for example, count the number of times that the detector signal transitions from high to low (in other words, the event source of the counter is the detector). Hence, the counter value can give us an estimate for how far that the wheel has turned. We will leave the problem of computing the distance that the circumference of the wheel has traveled as an exercise to the reader.

Figure 2 shows a snippet of code that illustrates how this may be used in practice. Here,

```
int main(void) {
  // Initialization
  timer0_config(TIMER0_EXT_FALLING};  // OUlib: set input of counter 0 to
                                      //  be pin PD4 one count for every
                                      // high-to-low transition

  :
  :

  timer0_set(0);    // OUlib: Set the timer 0 counter to a value of zero
  turn_on_motor();  // Turn on the motor attached to this encoder

  // Wait for the encoder to count to 100
  while(timer0_get() < 100) {};  // timer0_get() is provided by OUlib

  turn_off_motor(); // Turn off

  // The motor has now moved by 100 counts
}
```

Figure 2: Example code for controlling a motor to move a specified distance.

the encoder signal has been connected to pin *PD4*, which is a configurable input into the timer 0 hardware. In the example, this configuration is accomplished in the *initialization* section of the code. After initialization (and perhaps sometime later), the code resets the timer 0 counter and calls some function that turns on the motor that is attached to the encoder of interest. Every time that the wheel moves from a transparent section to an opaque one (from the perspective of the LED/detector pair), the Timer 0 counter is incremented by the hardware.

The code waits for the number of counts to reach 100 before turning off the motor. If we have the encoder illustrated in Figure 1, then the wheel will have turned a total of 100/3 times. The actual distance traveled by the wheel will depend on its diameter.

2 Configuring the Counters as Timers

In addition to counting external events, it is possible to also count events that are internal to the processor. In particular, by using the system clock (or a derivative thereof), the counter can be incremented at regular intervals. In this form, the counter becomes a form of *timer*.

Often, the system clock is running at some "high" frequency (e.g., in our microcontrollers, we might see clocks of 16 MHz or 20 MHz). However, it is often the case that we may want our timer to count at much slower rates. This is handled in microcontrollers through the use of hardware *prescalers* that divide the system clock down to some reasonable frequency. Figure 3 shows the relationship between the system clock, the prescaler and the counter value.

Prescalers are implemented as counters in and of themselves. Recall that if some counter is being incremented at a regular frequency, f, then bit 0 of the counter is exhibiting a regular signal at a frequency of $f/2$. Furthermore, bit 1 has a frequency of $f/4$. By "tapping into" the prescaler counter at different bits, we can divide the system clock by a range of different divisors (where the divisor takes a form of 2^i for some i).

The *Signal Selector* of Figure 3 is implemented as a multiplexer, selecting one of the available event sources to be provided to the counter. Which event source is selected is determined by the configuration of the counter. In the Atmel MegaX line of microcontrollers, this configuration is specified using another special purpose register (*Counter Configuration* in the Figure). Furthermore, OUlib provides the **timerX_config()** function to alter this selection.

It is often the case that not all possible divisors are available for selection. For example, in the Atmel Mega8 processor, *Timer 0* can be configured with prescalers **only** of: 1, 8, 64, 256 and 1024.

We would now like to answer questions involving the rate at which our counter is counting or the amount of time that a certain number of counts will take. In order to talk about these ideas, we first must be able to distinguish between cycles/second of the main system clock and of the prescaled signal. Here, we have artificially defined the unit of a "tick" as a single cycle of the system clock and "tock" as single cycle of the prescaled signal. Hence, we can express prescaler values in terms of *ticks per tock*, i.e, how many system clock cycles compose a single prescaled cycle.

Example: assume a system clock of $16,000,000\ ticks/sec$ and a prescaler of $64\ ticks/tock$. What is the period of a single increment of the Timer 0 counter?

Answer:

$$\frac{64\ ticks/tock}{16,000,000\ ticks/sec} \times 1\ tock \quad = \quad 4\ \mu s$$

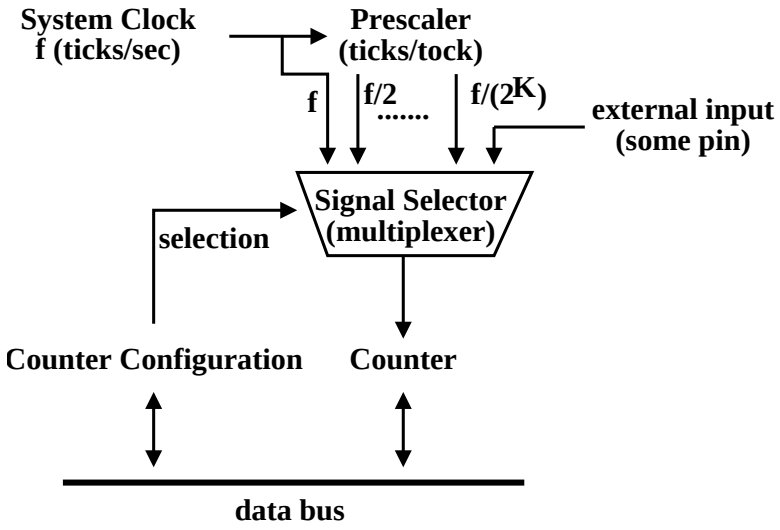

Figure 3: An example Counter/Timer implementation.

Example: What is the period of 250 increments of the Timer 0 counter?
Answer:

$$\frac{64\ ticks/tock}{16,000,000\ ticks/sec} \times 250\ tocks \quad = \quad 1\ ms$$

2.1 Using the Timer to Measure an Interval of a Specific Length

Figure 4 illustrates a snippet of code that can be used to produce a pulse on a pin that is 1 ms in duration. In the initialization section, we configure the timer 0 counter to receive input from the system clock, divided by 64. At this point in time, the timer 0 counter will begin to increment once every 4 μs We also initialize port C, pin 5 as an output pin.

At the point in time in our code that we require the 1 ms pulse, we set the timer's counter to a value of zero and set the pin into an *on* state. We then wait for the counter to reach a value of 250. Note that while we are waiting, it is possible for our main program to perform other tasks. However, care must be taken to ensure that these tasks do not take too much time. If they do, we run the risk of the counter reaching 251 or more ... or worse, reaching zero, at which point, we will continue to wait. After the waiting is complete, we turn the pin into an *off* state.

Note that for a given choice of prescaler, we are left with a certain resolution of the counter (one count corresponds to a fixed period of time). In addition, we have a maximum

```
int main(void) {
  // Initialization
  timer0_config(TIMER0_PRE_64);  // OUlib: set input of counter 0 to
                                 //  be the system clock divided by 64
  DDRC = 0x20;   // Configure pin 5 as an output

  :
  :

  // Assume that port C, pin 5 is already in logic 0 state


  timer0_set(0);    // OUlib: Set the timer 0 counter to a value of zero

  // Turn on the pin
  PORTC |= 0x20;

  // Could do some additional computations here

  while(timer0_get() < 250) {  // timer0_get() is provided by OUlib
    // Could also do some computations here (but they need to be short)
  };

  // Turn off the pin
  PORTC &= ~0x20;

  :
  :


}
```

Figure 4: Example code for producing a 1 ms pulse on pin 5 of port C.

value that we can count up to before the counter *turns over* back to a value of zero. Suppose that we want instead to produce a pulse that is exactly 3.2 ms in duration. What prescaler should we choose and how many tocks should we wait for?

We set up the problem as follows:

$$\frac{p\ ticks/tock}{16,000,000\ ticks/sec} \times x\ tocks \quad = \quad 3.2\ ms.$$

The remaining question is: what are the appropriate choices for p and x? p can take on only values of 1, 8, 64, 256 and 1024 for Timer 0. Furthermore, for x to be interesting in our code, $0 < x < 255$.

Answer: there are two viable solutions: $p = 256; x = 200$ and $p = 1024; x = 50$. Which one we choose will depend on our specific requirements. Since $p = 256$ has the highest timing resolution, in this particular situation, we would probably make this choice.

2.2 Choosing the Right Timer

Timer 0 of the Atmel Mega8 processor can be configured in a variety of ways in order to achieve different timing resolutions and durations. However, the choices are limited. For example, if we wanted to time a duration of 10 *sec*, we would have to substantially modify the implementation of Figure 4 in order accomplish this.

These limitations are addressed in the Atmel line of processors (as well as other types of processors) by introducing several different timers that have different capabilities. The Mega8 processor (and many in the Atmel line) are equipped with at least three timers:

Timer	**Counter Bits**	**Prescalers**
Timer 0	8	1, 8, 64, 256, 1024
Timer 1	16	1, 8, 64, 256, 1024
Timer 2	8	1, 8, 32, 64, 128, 256, 1024

When we are faced with producing an implementation that requires specific timing, we have three different choices to make: the timer, the prescaler and the number of counts to wait for. It is key to look at all reasonable combinations of these choices before settling on a particular one. Note that it is rare for us to achieve precisely the required timing. However, we can often get very close with the right set of choices. How close we need to be depends a lot on the problem that we are trying to solve.

3 Interrupts

Two challenges with the implementation of the pulse generator in Figure 4 are 1) the additional code that we are executing inside of the while loop must not take too long, and 2) we

have to keep checking the state of the timer. The latter of these problems is what we refer to as *busy waiting*. In this case, we know that the event of the timer reaching 156 will come soon. However, we can also have situations in which the event may come at some unknown future time, such as the pushing of a button. Constantly having to "spin" and check for the occurrence of the events in this can be very wasteful of valuable CPU time and distract us from doing other tasks.

The solution to these problems is the use of *interrupts*. Interrupts are events that are detected by the microprocessor hardware that cause the processor (under the right conditions) to 1) stop executing the code in the main program, 2) execute a special piece of code called an *interrupt service routine* (ISR) that deals with the event in some way, and 3) return to the main program and continue execution. When things are done properly, the main program will not even know that this interruption has occurred, except that a small amount of time has gone by that it cannot account for. For modern compilers and ISR support (e.g., gcc for our Atmel MegaX processors), the details of "getting it right" are handled for you automatically.

3.1 Interrupt Sources

Interrupts can be raised from a variety of different sources, and a unique ISR can be attached to each of these *event types*. Interrupt sources include: information arriving from an input device (such as a serial port), a pin transition from a low to a high state, and a timer achieving a particular state. In order to execute the associated ISR in response to an event, one must first *enable* the interrupt. This is typically handled by first modifying the associated bit in a special purpose register. In addition, the *global interrupt* bit must be turned on. This latter bit allows the program to temporarily turn off all interrupts so that critical code components (that cannot be interrupted) may be executed.

OUlib/Atmel notes:

1. Interrupt enabling/disabling is provided by a set of OUlib functions (see the documentation).
2. Global interrupts are enabled using **sei();**.
3. Global interrupts are disabled using **cli();**.

3.2 Communicating Between ISRs and the Main Program

Because ISRs are executed *asynchronously* from the main program (i.e, the main program does not explicitly call the ISR), there is no direct way to pass parameters to the ISR

or to receive a return value. Nevertheless, we often need to communicate between the two. Without support from an operating system, such is the case with embedded microcontrollers such as the Atmel MegaX line, we are faced with doing this communication using global variables.

Here are a couple of notes about global variable-based communication:

- Declare all shared global variables as **volatile**. Recall that as a variable is manipulated, it is first transferred from main memory to a general purpose register. This means that we briefly have two copies of the value. This keyword tells the compiler that each time the program accesses the variable, it must be read immediately from memory, and not assume that the value stored in the general purpose register is up to date. Likewise, as soon as the general purpose register value is modified, it will be written back out to memory.

  An example declaration:

  ```
  volatile uint8_t duration;
  ```

- If interrupts are enabled, the main program may be interrupted at any time. This is particularly dangerous if the main program is in the middle of a change to or a read from a shared global variable (or set of variables). Care must be taken in order to ensure that interruptions do not disrupt the actions of the main program. For example, suppose that a shared variable is declared as a **int16_t** and that we are using an 8-bit processor such as an Atmel MegaX. To use this value, the main program will first copy it into a pair of general purpose registers. The top 8 bits will be copied in one instruction, followed by the bottom 8 bits in the next instruction. If an ISR interrupts this copy operation between the two byte transfers and modifies the variable, then the main program will "see" the old value of the first 8 bits and the new value of the lower 8 bits.

  While there are multiple solutions to this problem, one of the simplest is the following. Before the main program reads from or writes to shared global variables, the ISR is disabled. As soon as the accesses are complete, the ISR is then re-enabled. While the specifics will vary from situation to situation, it is often good to keep the time period during with the ISR is disabled as short as possible. This means no delays, no input/output operations, and no expensive operations.

4 Counter-Driven Interrupts

Each of the Atmel timers can be configured to raise a *counter overflow interrupt* any time that the counter transitions from its max value back to zero. When configured to receive input from the system clock (through a prescaler), the timers will (by default) generate regular overflow interrupts. For timers 0 and 2, we will have 256 *interrupts/tock*; for timer 1, we have 256^2 *interrupts/tock*.

Figure 5 shows a snippet of code that produces a regular periodic signal on pin 5 of port C. The main program initializes the hardware by first configuring the prescaler (to 64 *ticks/tock*), and then enabling the Timer 0 Overflow interrupt and enabling global interrupts. After initialization, all of the action on the pin will be due exclusively to the ISR. In this case, the main program will do nothing but spin in a loop.

The interrupt service routine is declared using the **ISR** macro, which takes as input the name of the event to which to attach ISR the code. A full list of event names is given in the avr-gcc LibC documentation. This particular ISR simply changes the state of the output line of port C, pin 5 (we also call this *toggling the state of the pin*). The duty cycle of the resulting signal is 50% and the frequency of the signal is:

$$\frac{16,000,000\ ticks/sec}{64\ ticks/tock \times 256\ tocks/interrupt \times 2\ interrupts/cycle} = 488.28\ cycles/sec.$$

4.1 Changing the Interrupt Frequency On-the-Fly

In addition to manipulating the configuration of the timer by changing the prescaler, it is also possible for the ISR itself to affect the time at which the next interrupt will occur. An example is given in Figure 6. Here, the main program is the same as in Figure 5. However, the ISR has been modified. The state of the output pin is still altered every interrupt. But, we have introduced a new variable, **state**, that also alternates between a value of zero and one. When the **state == 1**, the ISR behaves as before, producing a low signal that has a duration of:

$$\frac{64\ ticks/tock \times 256\ tocks/interrupt}{16,000,000\ ticks/sec} = 1.024\ ms.$$

However, when **state == 0**, the ISR resets the Timer 0 counter to 200. Without this change, the counter would have to count from zero back around to zero before the next interrupt is generated (a total of 256 counts). With the change, the counter need only count from 200 to zero, a total of 56 counts. So, the period of the high phase of the signal is:

```
// ISR declaration.  The set of possible ISRs is given in the
//  avr-gcc libC documentation
ISR(TIMER0_OVF_vect) {
  // Flip the state of the output pin to the opposite value
  PORTC ^= 0x20;
}


int main(void) {
  // Initialization
  timer0_config(TIMER0_PRE_64);  // OUlib: set input of counter 0 to
                                 //  be the system clock divided by 64
  DDRC = 0x20;                   // Configure pin 5 as an output


  timer0_enable();  // Enable the Timer 0 interrupt
  sei();            // Enable global interrupts

  while(1) {
    // Add code to do something else
  };


}
```

Figure 5: Example code for producing a 488.28 Hz signal with 50% duty cycle.

$$\frac{64\ ticks/tock \times 56\ tocks/interrupt}{16,000,000\ ticks/sec} \quad = \quad .0224\ ms.$$

This makes for a repeating signal that has a total duration of $1.024 + 0.224 = 1.248\ ms$ and a duty cycle of $0.224/1.248 = 18\%$.

```
// ISR declaration.  The set of possible ISRs is given in the
//  avr-gcc libC documentation

ISR(TIMER0_OVF_vect) {
  static uint8_t state = 0;

  // Flip the state of the output pin to the opposite value
  PORTC ^= 0x20;


  if(state == 0) {
    // In this case, dramatically shorten the time before the
    //  next interrupt
    timer0_set(200);
  }

  // Flip to the opposite state
  state ^= 1;
}


int main(void) {
  // Same as previous example
}
```

Figure 6: Example code for producing a 801.28 Hz signal with 18% duty cycle.