CpE 213 Assignment 10
Due date: Tuesday November 22 at 11:59 pm
Email submissions are acceptable.

Timers and counters in C

Purpose

This example supplements the material in Chapter 9 of Mazidi on Timer Operation. It compares pure software solutions to combined hardware/software solutions to the problems of time delay and event counting.

Time delay and event counting

Many microcontroller applications can be categorized as either time delays or event detection/counting. Time delays can be implemented with a pure software approach (neglecting the hardware required to actually run the software of course) or can be supplemented by a variety of hardware assists. We will examine a range of solutions appropriate for the 8051 family. The same is true for event detection and counting although for external events we will need some minimum amount of hardware to simply interface with the external device causing the event.

We already know that 8051 instructions take from 1 to 4 instruction cycles to execute. With a 12 Mhz clock and 12 clock cycles per instruction cycle this corresponds to a range of 1 to 4 µS per instruction with 1 to 2 µS being typical. We can use this to advantage in producing a wide range of time delays. A simple for loop can be used to provide an adjustable delay as shown in Listing 1.

Listing 1

```
uchar i;
 for (i=0;i<DELAY;i++);
```

This loop takes a total of 44 states to execute or 44 µS with a 12 Mhz clock. Examination of the assembly code produced for this code fragment, shown in Listing 2, reveals that the for loop consists of five instructions, two of which are executed once and three are executed DELAY times. The while loop adds the SJMP at the end which is also executed once. The INC, MOV, and CJNE each take 1, 1, and 2 cycles respectively. The CLR, MOV, and SJMP also take 1, 1, and 2 cycles. Adding the cycles thus gives 1+1+2+DELAY(1+1+2) or 4*DELAY+4. DELAY is defined as 10 so the total delay is 44 states. If we increase the value of DELAY to a maximum of 255 we would expect a total delay of 4*255+4 or 1024 µS. Longer delays can be implemented by using a uint instead of a uchar, nested for loops, adding dummy code to the loop, etc. It should be obvious that the shortest delay we can implement using this approach is 8 states. We could 'fine tune' things a bit by using a decrementing counter. If we do that, a DELAY value of 10 will produce a 24 state delay. That is because the INC, MOV, and CJNE can be replaced by a single DJNZ instruction.

Listing 2

```
0000 E4       ?C0001:  CLR     A
0001 F500              MOV     i,A
0003 0500     ?C0003:  INC     i
0005 E500              MOV     A,i
0007 B40AF9            CJNE    A,#0AH,?C0003
000A 80F4              SJMP    ?C0001
```

So, what can you do with a time delay? For one, you can generate square waves of various frequencies and duty cycles. A square wave with fixed frequency and variable duty cycle is called a *pulse width modulated* signal or PWM signal for short. Pulse width modulation is ubiquitous in microcontroller applications because it can be used for a wide variety of purposes. These include dc motor control, power supplies, three phase power control, lamp dimmers, sound generation, and TV remote controls to name a few.

One of the fastest possible square waves that we can generate with software on an 8051 can be generated with the single line of C: `while(1)SQRWV=~SQRWV;`. SQRWV is defined as P1.0 with the code `sbit SQRWV= P1^0;`. The assembly code is shown in Listing 3. CPL is one cycle while SJMP is 2 cycles, thus the square wave generated by SQRWV has a period of 6 states and a duty cycle of 50%. Compare this with example 4.2 on page 90 of ISM. That example can be generated with the C statement: `while(1) { SQRWV=1; SQRWV=0;};`

Listing 3

```
0006         ?C0006:
0006 B290              CPL     SQRWV
0008 80FC              SJMP    ?C0006
```

A time delay is implemented with a counter that simply counts internal *events*, ie instruction cycles. Our for loop has an index (i) that counts how many times we've gone through the loop and the loop takes a certain amount of time to execute. It's that simple. A timer is a time counter. From here it's a simple step to a counter that counts other events besides instruction, or indirectly, clock cycles. But first we need a way to simply detect the events we are going to count. Let's count rising edges on Port 1 bit 0 for example. These edges might come from a square wave whose duty cycle or frequency we want to measure. Or they might be coming from a sensor or generated by key presses on a keyboard. We don't really care, as long as we have a source of rising edge events over a period of time to measure.

If we really want to measure rising edges, we need to look for back-to-back readings of 0 and 1 from our signal. If the context guarantees a 0, we can dispense with the 0 reading and only look for a 1. That might happen, for example, if we start a process and wait for it to complete. Starting the process may guarantee that the completion flag is clear for awhile so we only need to watch for the completion flag to set. This situation occurs when working with communication devices like UART's for example. We won't make any such assumptions though. Instead, we'll assume that the signal at P1.0 is free running and outside of our control. Perhaps it's coming from an IR sensor receiving a signal from a TV remote control. Rising edges are thus defined as an input of '0' followed by an input of '1'. The code in listing 4 can be used to wait for a rising edge at SQRWV. SQRWV was defined to be P1.0 as before. The variables sigcur and sigprev were declared as type bit with the statement `bit sigcur, sigprev`. The work is all done with a 'do while' statement. It saves the current reading as the previous reading, reads a new value of SQRWV, and terminates when sigprev==0 and sigcur==1. Notice the use of the bang (!) operator to change the sense of the condition so that the do-while will terminate when the condition (~sigprev&sigcur) becomes true, not the other way around. The squiggle (~) operator is used to take the bitwise complement of sigprev. It's easy to confuse squiggle with bang but they're not interchangable. In this case they are equivalent, but in general bang changes any non-zero value to zero and a zero value to a 1 (true to false and false to true) whereas squiggle takes the bitwise 1s' complement of a value. Finally, you may wonder why the SQRWV=1 statement is necessary. Remember that the 8051's ports are mainly psuedo bidirectional. Port 1 bits have a pullup resistor and pulldown transistor. If P1.0=0 that means the pulldown is 'on' and there is no way short of burning out the chip that we are ever going to pull it up externally. On the other hand, if the port bit is a '1', then externally pulling it up again doesn't hurt and since the internal pullup is a high impedance resistor we can externally pull it down easily. That's why if you write a '1' to an 8051 port bit, you don't necessarily read back a '1'.

Listing 4

```
SQRWV=1;
 sigcur= SQRWV;
 do {sigprev=sigcur; sigcur=SQRWV;} while(!(~sigprev&sigcur));
```

Figure 1 shows what is happening and why the code is written the way it is. It shows a pulse on SQRWV with four evenly spaced samples. If we measure the execution time of the do-while loop, we'll find that it takes 11 states so the samples will be spaced 11 µS apart when using a 12 Mhz clock. We want to detect the edge lying between samples 2 and 3. The first assignment merely 'primes the pump' so to speak. Once we get into the loop, we'll take sample 1, put it into sigprev, and read sample #2 as sigcur. Both of these values are 0 so the while condition evaluates to !(~0&0) which is !(1&0) equal to !(false) or true and the loop continues. The next time through the loop we take sample #2 (which is in sigcur), save it as sigprev, and read the new sample #3 into sigcur. This time the condition evaluates to false and we fall through the loop, ready to do something based on the occurence of the event (a rising edge on SQRWV). The four possibilities of sigcur and sigprev are 00, 01, 10, and 11. Only the 10 case drops us out of the loop. Why do we only sample once per loop iteration? Let's see what happens if we make this common mistake. Assume that the two samples we take are samples 1 and 2. No edge is detected so the loop continues. The next two samples are 3 and 4 and we've missed the edge between 2 and 3! We need to compare every sample with every previous value, not just in pairs.

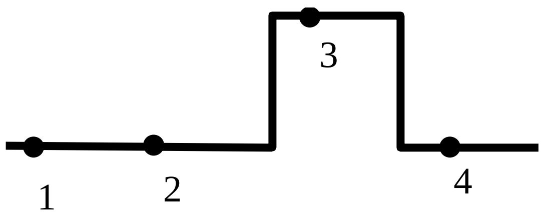

Figure 1 Sampling a square wave

Ok. So we can detect edges. Now what? It's a simple matter to add a counter to listing 4 to count rising edges. That's what the 8051 literature call 'a counter'. Note that both the delay loop and the edge counter are 'counters' in a sense. It's just that the former counts time while the latter counts events. If we combine these two ideas, we will have a useful application: an interval measuring device. In other words, we could measure the interval between rising edges and thus the period of the square wave, or we could measure the interval between successive edges and thus the duty cycle of the square wave. We could also count the number of edges during a period of time and get a measure of the signal's average frequency.

Listing 5 shows one variation on this theme. Just like before we are using a for loop to provide some time delay. Inside the for loop, instead of doing nothing like we did before, we now have an edge detector and a counter. During our time delay we can count up to ten edges. The number of edges occuring during this period of time is a measure of the signal's frequency.

Listing 5

```
#include <reg51.h>
#define uchar unsigned char
#define DELAY 10
sbit SQRWV= P1^0;
main(){
 bit sigcur, sigprev;
 uchar i,cnt;
 SQRWV=1;
 while(1){
  cnt=0;
  sigcur= SQRWV;
  for (i=DELAY;i>0;i--) {
   sigprev=sigcur; sigcur=SQRWV;
   if (~sigprev&sigcur) cnt+=1;
  };
  /* do something with cnt here */
 };
}
```

It takes roughly 136 states or 136 µS to go from the first sample of SQRWV through the end of the for loop with a DELAY of 10. If we count, say, four rising edges during that time, it means that our square wave's frequency is roughly 4/136 Mhz or about 30 kHz. Actually that could be off by plus or minus a cycle. We counted four edges but that might be because we're just over three complete cycles or we may be just under five cycles. So our square wave is anything between 22 kHz and 36.8 kHz. The error is 1/136 Mhz or about plus or minus 7 kHz. Not terribly accurate but not bad for 'free' software.

So, now we can not only count time, we can count events, and measure events per interval or time per event (frequency, periods, pulse widths, duty cycles, etc). There are several problems though. The first is that the resolution and ranges aren't very good and are limited by software. We can produce really slow square waves but nothing faster than about 1/6 Mhz and we can detect events but only if their arrival rate isn't too high. The other problem is that our solutions hog the whole processor. It's possible to do multiple things at once like generate several square waves of arbitrary frequency and measure the pulse widths of several arbitrary signals, but it's cumbersome. The solution is to use some hardware assists. Short of dedicating a processor to each simple task, there are some simple circuits that will help considerably. Not unexpectedly these circuits are just counters.

The 8051 has two counters built in. The 8052 and most new variants of the 8051 have three so-called *counter/timers*. These can generally be configured to count either time (a timer) or events (a counter). Some can automatically do simple tasks like measure a pulse width (capture register) or generate a pulse width modulated waveform (comparator register or pwm output device). See the data sheets for the Philips 89C51Rx2 and 87C752 or Infineon C500 family for examples. We'll limit this example to the generic 8051 T0 and T1 counter/timers.

Not counting the startup code, the program in Listing 6 takes only 17 bytes and generates a square wave with a period of 100 µS and 50% duty cycle. It's exactly the same code as example 4-3 on page 91 of the text. The observant reader might say 'now wait a minute. Doesn't this program take all of the cpu time like before?' and that would be correct. The big difference is that before the while(!TF0) we could do some other small task and as long as it took less than 50 µS the timing would all be the same. In the previous software only solution, we had to be very careful about how many instructions were executed or we'd spoil the timing. In fact the very observant reader may have already noticed the 'other' problem in listing 5. That problem is the different amount of time that is taken depending on whether we count an event in the loop or not. It's a fixed amount of time per event and the error could be calibrated out but in general, tuning software loops like this is a tricky and tedious business. The timer hardware makes it much simpler. We could have an arbitrarily complex task before the while loop taking anywhere from 0 to 49 µS and the timing is all the same since the timer runs independent of the software.

Listing 6

```
#include <reg51.h>
#define uchar unsigned char
#define PERIOD 100 //period in microsec
sbit SQRWV= P1^0;
main(){
 TMOD=0x02;
 TH0= -PERIOD/2;
 TR0=1;
 while(1){
  while(!TF0);  //wait for timer countdown
  TF0=0;
  SQRWV=~SQRWV;
  }
}
```

There are several features of listing 6 worth pointing out. First, note that the period is defined symbolically instead of using a 'naked constant' like the example in the text. This is good coding practice. Second, the period is defined and the half period calculated using a static expression. Let the compiler do the arithmetic for you, it costs nothing. Third, always remember that the 8051 timer/counters count UP, not down. If you want a delay of x you have to initialize the counter to –x. Here we initialize the high half of counter 0 (TH0) with –50. After starting the counter (TR0=1) TL0 will count up to 0, taking 50 µS to do so (assuming a 12 Mhz clock) and set TF0, exiting the while loop. Once TF0 is set, it stays set until cleared. When TF0 sets, TL0 is reloaded from TH0 and keeps on counting in mode 2. If you forget and initialize the counter with x instead of -x (and most will including me at times), the counter will blindly count from x all the way up to 0 so your delay will be N-x instead of x. In this case, since we're using an 8 bit counter, the difference is 256-100 versus 100, not big. With a more common 16 bit counter, the difference can be substantial; 65436 versus 100 in this case. If you are using a simulator instead of a real computer, 65436 µS can seem like an eternity. If the mistake is inside a loop to give a much longer delay, even a real processor could wait for a long long time. Notice that even the author forgot this time. He initialized TH0 but not TL0, so the first time through TL0 will count up from whatever value it happens to have (0 after a reset) all the way to 0. That means the first half cycle of the square wave will last 256 µS instead of 50 µS. Not a big deal but bugs like this can be difficult to find. Fourth and finally, the program can be divided into two parts: an initialization phase (TMOD= to TR0=) and a run phase (the outer while loop). This is typical. Certain tasks only need to be done once and others need to be done repeatedly. It is common to use functions for this to make the code easier to understand. We didn't do it here since this is such a short piece of code but in general it's good practice. Listing 7 is the general idea.

Listing 7

```
#include <reg51.h>
```

```
#define uchar unsigned char
#define PERIOD 100 //period in microsec
sbit SQRWV= P1^0;

void init(void){
 TMOD=0x02;
 TH0= -PERIOD/2;
 TR0=1;
 }

void gensqwv(){
  while(!TF0);  //wait for timer countdown
  TF0=0;
  SQRWV=~SQRWV;
  }

main(){
 init();
 while(1) gensqwv();
}
```

But what if our task involves computing a fast fourier transform or other suitably complex task taking much longer than 50 µS? There are a couple of solutions to this dilema. You can either break the long task up into several smaller pieces and ‘schedule’ each of them in turn (complicated, ugh) or you can let TF0 ‘interrupt’ the processor when it gets set. In effect you can set things up so that you can focus on the complex task and let the timer take care of itself. It’s not exactly that simple but interrupts do make things easier.

Summary

This example has shown:

- The difference between a timer and a counter: timers count time and counters count events. Both are counters.
- A software only timer and a software only counter
- Control of the 8051’s internal timer with C.

Questions

Answer the following questions.

1. Show how the code in Listing 4 can be shortened by 1 cycle and decreased in size by two bytes by rewriting the C code. Can it be made even smaller and faster without changing the function by rewriting in assembly code?
2. What is the smallest pulse width that can be reliably detected with the code in Listing 4?
3. Listing 5 can be used as the basis for measuring the approximate frequency of a square wave connected to P1.0. Describe a simple change to Listing 5 that will result in a smaller error in the frequency measurement. Few things in life are free and making this change will result in making something else worse. What is the result of decreasing the error?
4. Use microvision to run the program in Listing 6. Open the Timer/Counter 0 and Port 1 windows under Peripherals and watch what happens as you step through the code. Fix the program so that TL0 is properly initialized.