CpE 213
Digital Systems Design

Lecture 25

Thursday 11/20/2003

UMR UNIVERSITY OF MISSOURI-ROLLA
The Name. The Degree. The Difference.

Announcements

- HW 8 has been posted.
  - Due date: Tuesday Nov. 25.
- Project peer review has been posted.
  - Please keep the ratings in mind while working on the project.
- Project demos are due on Monday 12/1.
  - Signup sheet is on my door. The spokesperson for each group should sign up for a slot on the schedule.
  - The sheet is posted on Blackboard for your reference.
  - All group members must be present at demonstration.
  - Deadline for signup is Tuesday 11/25.
  - Send me email by 11/21 if you have a valid schedule conflict all day on Monday 12/1.

Outline

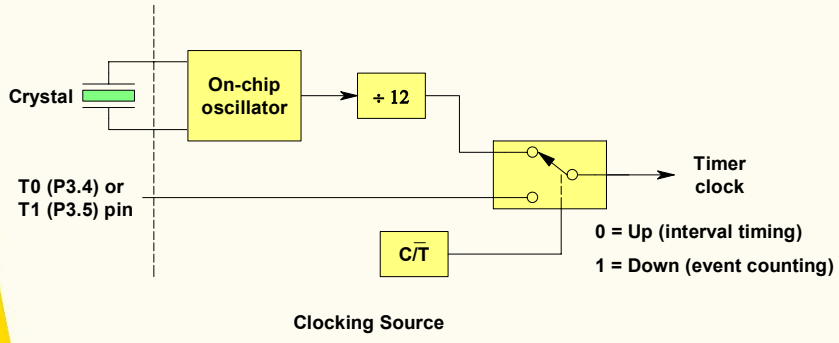
- Timers and Counters for the 8051
  - Review
  - Modes
  - Programming to generate time delays
  - Programming as event counters

Some slides adapted from Mr. K. T. Ng

Review
What Is a Timer/Counter ?

- A counter is a device that generates binary numbers in a specified count sequence when triggered by an incoming clock pulse.
- Timers are counters that count pulses. If the pulses are “clock” pulses, then the timers count time.

Timer/Counter Clocking Source



Clocking Source

Timer Special Function Registers (SFRs)

- As mentioned before, the 8051 has two timers that essentially function the same way.
- One timer is called TIMER0 and the other is TIMER1.
- The two timers share two SFRs (TMOD and TCON) that control the timers.
- Each timer also has two SFRs dedicated solely to itself (TH0/TL0 and TH1/TL1).

Timer Control Register (TCON)

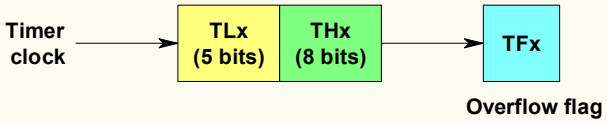
- TCON has the following functions:
  - Turns Timer 0 and Timer 1 on or off.
  - Sets trigger type (edge or level) for interrupt 0 and interrupt 1.
  - Sets flags when timer 0 or timer 1 overflow.

Timer Mode Control (TMOD) Register

- TMOD controls timer 0 and timer 1 functions.
- The register's upper nibble controls Timer 1 and the lower nibble controls timer 0.
- The four bits for each timer control a gate function, selection of counter or timer, and timer mode.
- The timer or counter function is selected by control bits C/T; these two timers/counters have 4 operating modes, selected by appropriate M0 and M1 bits in the TMOD register.
- TMOD is not a bit-addressable register. It has to be manipulated as a byte.

Mode 0 - 13 Bit Counter

- TH, TL = XXXXXXXX, 000XXXXX
- 8 bits from TH
- 5 bits from LSB of TL
- For compatibility with older microcontroller (8048), not very useful now.
- Can hold values between 0000 and 01FF.
- When timer reaches 1FFF, it rolls over to 0000 and raises TF.
- Range = 0 to 8191.

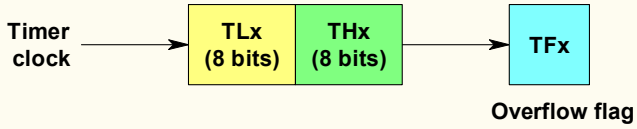


8051 Timer Mode 0

Timer Mode 1 - 16 Bit Counter

- 8 bits from TH, another 8 bits from TL.
- Each byte loaded separately
- Counts UP, so initialize with negative value.
- The counters start counting when enabled
  - SETB TR0 for Timer 0 and SETB TR1 for Timer 1
- Counts time (when acting as timer) until a preset number as specified by the THx and TLx registers is reached.
- Generates an overflow (interrupt) when timed out.
- In order to repeat the process the registers TH and TL must be reloaded with the original value and the timer overflow flag must be reset under the control of the program.

Timer Mode 1 (Cont.)



8051 Timer Mode 1

- Example

```
TH0= -1000 >> 8; //-1000. = 0xFC18
TL0= -1000 & 0xFF;
```

- Why doesn't -1000/256 work?

Finding Values to be Loaded into the Timer

- Assume that we know the desired amount of timer delay, t_{delay}. To calculate the values to be loaded into the TL and TH registers, using crystal frequency of $X_{crystal}$ MHz, the following steps are needed:
  - Step 1: find n, where n = $\frac{t_{delay} \bullet X_{crystal}}{12}$
  - Step 2: Perform 65536 - n
  - Step 3: Convert step 2 result to hex, assume this result is yyxx
  - Step 4: Set TL = xx and TH = yy.

Group Exercise

- Assuming a desired time delay of 5 ms and Xcrystal = 11.0592 MHz, determine the values of TL and TH in hex.

A Better Way

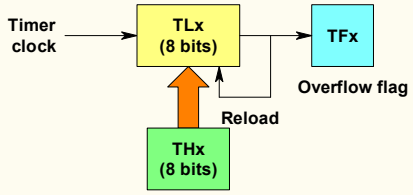
- A better way is to let the assembler do the arithmetic.
- Suppose the required delay count number, n, is 4608. When using timer0, we can load this value into TH0 and TL0 as follows:

```
MOV TH0,#-4068 SHR 8    ;MSB of -4068
MOV TL0,#-4068          ;LSB of -4068
```

- What does SHR 8 mean ?
  - It means the assembler would take the 16-bit number and shift it right by 8 bits. This gives us the MSB of -4068.
- The assembler automatically selects only the bottom 8 bits of the value, so the LSB of -4068 goes into TL0.

Mode 2 - 8 Bit Auto-Reload

- TL0 or TL1 is an 8 bit counter/timer
- TH0 or TH1 holds an initialization value
- TL reloaded from TH on 255 to 0 transition
- Sequence is: `TH, TH+1, TH+2,…, 255, TH,…`
- The reload leaves TH unchanged.
- Used to generate UART baud rate clock; a periodic flag or interrupt.

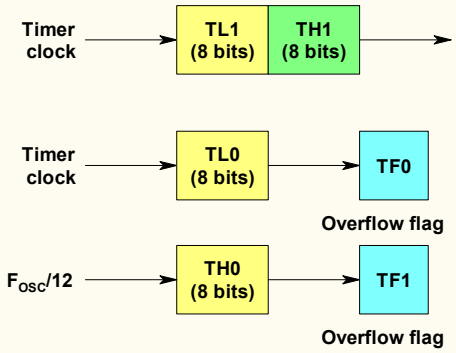


8051 Timer Mode 2

Timer Mode 3 - Three from Two

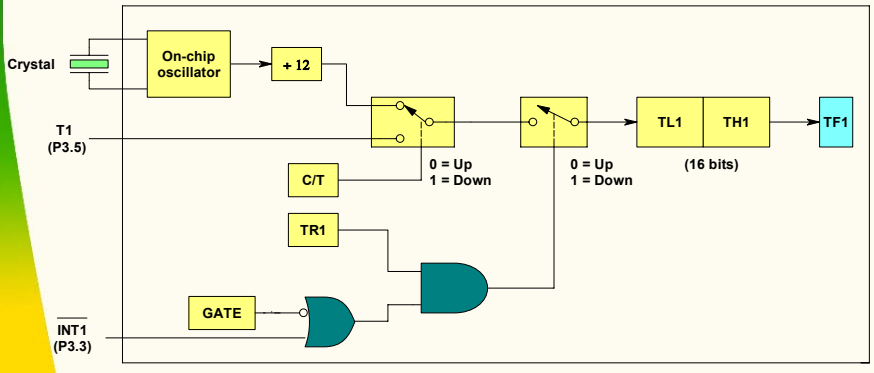
- T0 is split into two 8-bit counter/timers.
- First counter (TL0) acts like mode 0.
- Second counter (TH0) counts CPU cycles.
  - Uses TR1 (timer 1 run bit) as enable.
  - Uses TF1 (timer 1 overflow bit) as flag.
  - The clock frequency is $F_{osc}/12$
- T1 becomes a 16-bit timer that can be started and stopped at any time by the mode operating bits M0 & M1, but cannot cause an interrupt.
- T1 is normally in mode 2 when T0 in mode 3
- T1 is used for baud rate clock while T0 provides two 8-bit counters.

Timer Mode 3 (Cont.)



8051 Timer Mode 3

C/T' and GATE Bits



Timer 1 operating in mode 1

C/T' and GATE Bits (Cont.)

- C/T' is used as the timer or counter select bit. It is cleared for timer operation (input from internal system clock), and set for counter operation (input from Tn input pin).
- When the Gate control is set, Timer/Counter n is enabled only while INTn pin is high and TRn control pin is set. When cleared, Timer n is enabled whenever TRn control pin is set.
- Setting Gate = 1 allows us a way of controlling the stop and start of the timer externally at any time via a simple switch.

Summary

- Timers/counter implemented in HW
- Counts instruction cycles (timer) or
- External events (counter)
- 8, 13, or 16 bit counters available
- All count UP
- Overflow (interrupt) when MAXCOUNT reached
- Gated and auto-reload variations

Other variations

- Three counters on 8052 (T0,T1,T2)
- Watchdog timer
- Capcon (Capture/Comparator) registers
- PWM

Programming in Mode 1

- To generate a time delay using timer mode 1, the following steps are taken.
  1. Load the TMOD value register indicating which timer (0 or 1) is to be used and which timer mode is selected.
  2. Load registers TL and TH with initial count values.
  3. Start the timer by the instruction "`SETB TR0`" for timer 0 and "`SETB TR1`" for timer 1.
  4. Keep monitoring the timer flag (TF) with the "`JNB TFx,target`" instruction to see if it is raised. Get out of the loop when TF becomes high.

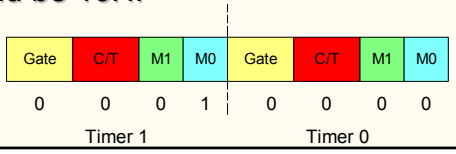
Programming in Mode 1 (Cont.)

1. Stop the timer with the instructions "`CLR TR0`" or "`CLR TR1`", for timer 0 and timer 1, respectively.
2. Clear the TF flag for the next round with the instruction "`CLR TF0`" or "`CLR TF1`", for timer 0 and timer 1, respectively.
3. Go back to step 2 to load TH and TL again.

- The programming techniques mentioned here are also applicable to counter/timer mode 0. The only difference is in the number of bits of the initialization value.

Example of Time Delay Generation Using Mode 1

- Generate a time delay of 1 ms Timer 1 in mode 1. Assume crystal frequency to be 12 MHz.
- Solution:
  - First calculate the value n:
  - n = (1 × 10^3 × 12 × 106) / 12 =1000
  - 65536 - 1000 = 64536 = FC18H.
- The initial values to be loaded into TH1 and TL1 are FC and 18, respectively.
- For timer 1 to be set to mode 1, the configuration of the TMOD register should be 10H.



Time Delay Using Mode 1 (Cont.)

The time delay program segment is as follows:

```
       MOV   TMOD,#10H    ;Timer 1, mode 1 (16-bit mode)
HERE:  MOV   TL1,#18H     ;TL1 = 18H
       MOV   TH1,#0FCH    ;TH1 = FCH
       SETB  TR1          ;Start timer 1
AGAIN: JNB   TF1,AGAIN    ;Monitor timer flag 1
                          ;until it rolls over
      CLR    TR1          ;Stop timer 1
      CLR    TF1          ;Clear timer 1 flag
```

Generating a Square Wave Using Timer 0

- Write a program fragment to continuously generate a square wave of 2 kHz frequency on pin P1.5 using timer 0. Assume the crystal oscillator frequency to be 12 MHz.
- Solution: Initial calculations are as follows.
- The period of the square wave is T = 1/(2 kHz) = 500 μs. Each half = 250 μs.
- The value n for 250 μs is:
  - (250 × 10^(-6) × 12 × 10^6) / 12 = 250
- 65536 - 250 = FF06H.
- TL = 06H and TH = 0FFH.

Square Wave Generation on Pin P1.5 (Cont.)

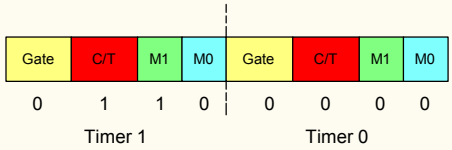
```
            MOV   TMOD,#01    ;Timer 0, mode 1
AGAIN:      MOV   TL0,#06H    ;TL0 = 06H
            MOV   TH0,#0FFH   ;TH0 = FFH
            SETB  TR0         ;Start timer 0
BACK:       JNB   TF0,BACK    ;Stay until timer
                              ;rolls over
            CLR   TR0         ;Stop timer 0
            CPL   P1.5        ;Complement P1.5 to
                              ;get Hi, Lo
            CLR   TF0         ;Clear timer flag 0
            SJMP  AGAIN       ;Reload timer since
                              ;mode 1 is not
                              ;auto-reload
```

Steps to Program in Mode 2

- To generate a time delay using the timer mode 2, the following steps are taken:
  1. Load the TMOD value register indicating which timer (0 or 1) is to be used; select timer mode 2.
  2. Load TH register with the initial count value. As it is an 8-bit timer, the valid range is from 00 to FFH.
  3. Start the timer.
  4. Keep monitoring the timer flag (TFx) with the "`JNB TFx,target`" instruction to see if it is raised. Get out of the loop when TFx goes high.
  5. Clear the TFx flag.
  6. Go back to step 4, since mode 2 is auto-reload.

Program Example for Mode 2

- Write a program segment that uses timer 1 in mode 2 to toggle P1.0 once whenever the counter reaches a count of 100. Assume the timer clock is taken from external source P3.5 (T1).
- Solution:



- The TMOD value is 60H
- The initialization value to be loaded into TH1 is
- 256 - 100 = 156 = 9CH

Program Example in Mode 2 (Cont.)

```
      MOV   TMOD,#60h    ;Counter1, mode 2, C/T'= 1
                         ;external pulse
      MOV   TH1,#9Ch     ;Counting 100 pulses
      SETB  P3.5         ;Make T1 input
      SETB  TR1          ;Start timer 1
BACK: JNB   TF1,BACK     ;Keep doing it if TF = 0
      CPL   P1.0         ;Toggle port bit
      CLR   TF1          ;Clear timer overflow flag
      SJMP  BACK         ;Keep doing it
```

- Note in the above program the role of the instruction "`SETB P3.5`". Since ports are set up for output when the 8051 is powered up, we make P3.5 an input port by making it high.

Reading the Value of a Timer

- If the timer is in an 8-bit mode - that is, either 8-bit auto-reload mode (mode 2) or in split timer mode (mode 3), then reading the value of the timer is simple. Simply read the 1-byte value of the timer and that's it.
- For reading the value of a 16-bit timer, two common ways are:
  - Read the actual value of the timer as a 16-bit number.
  - Detect when the timer has overflowed.
- When reading a 13-bit or 16-bit timer, there is a potential problem of "phase error" if the low-byte overflows into the high byte between the two read operations

Reading the Value of a Timer (Cont.)

- Consider the case that the timer value was 14/255 (high byte 14, low byte 255) but you read 15/255. Why ?
- The solution to this phase error problem is:
  - Read the high byte of the timer first.
  - Then read the low byte.
  - Then read the high byte again.
  - If the high byte read the second time is not the same as the high byte read the first time, we read both bytes again.
- The code that appears in here is known as reading a timer "on the fly."

Reading the Value of a Timer (Cont.)

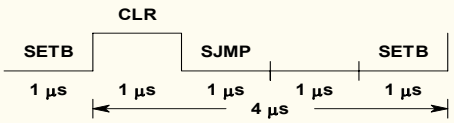
```
REPEAT:   MOV  A,TH0
          MOV  R0,TL0
          CJNE A,TH0,REPEAT
          MOV  R7,A
```

- When the code terminates we will have the low byte of the timer in R0 and the high byte in R7.
- A much simpler alternative is to turn off the timer run bit (i.e. `CLR TRx`), read the timer value, and then turn on the timer run bit (i.e. `SETB TRx`).
- The drawback of this simple approach is that the timer will be stopped for a few machine cycles, which may not be tolerable in some cases.

Short Timing Interval Using Software Loops

- Very short intervals can not be generated using timers because of the overhead needed to start and stop the timers.
- Such short timing intervals are usually generated using tight software loops.
- The following code segment generates a 250 KHz wave with 25% duty cycle on P1.0. Assume XTAL = 12 MHz.

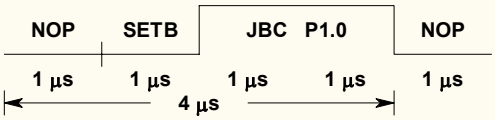
```
LOOP:     SETB  P1.0  ;1 machine cycle
          CLR   P1.0  ;1 machine cycle
          SJMP  LOOP  ;2 machine cycles
```



Short Timing Interval (Cont.)

- The following code segment generates a 250 KHz square wave on P1.0. Assume XTAL = 12 MHz.

```
LOOP:       NOP              ;1 machine cycle
            SETB  P1.0       ;1 machine cycle
            JBC   P1.0,LOOP  ;2 machine cycles
```



Summary of Techniques

Time Interval	Technique
No limit	Software loops
No limit	16-bit plus software loops
65536 machine cycles	16-bit timer
256 machine cycles	Auto-reload (8-bit)
<~ 10 machine cycles	Tight software tuning

CpE 213
Example ISM81 – Timers and counters in C

Purpose

This example supplements the material in Chapter 4 of ISM 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.

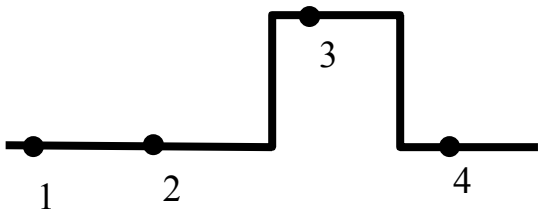


Figure 1 Sampling a square wave

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.

CpE 213
Example ISM92 - Using Timer 0 Mode 1

Purpose

Use Timer 0 in 16 bit mode 1 to generate a 1 kHz square wave on P1.0. Develop a C function to provide the appropriate time delay. Adjust timing parameters to account for added delay due to instruction execution.

Timer 0 Mode 1

The tricky part of using 16-bit mode on an 8-bit processor is getting at the two halves of the 16-bit initial counter value. You can always figure this out manually but it's better practice to let the compiler do it for you. It's easier to change, easier to read, and there will be fewer mistakes made. There are right ways and wrong ways to do this. We'll look at several such.

Rather than use inline code, we'll put our timing code into a C function and call it from main. The function will use timer 0 to provide accurate delays of n μS. Main will call the delay routine to delay half a cycle, toggle the port bit, and repeat indefinitely. We will use dscope to determine the added delay due to instruction execution and we'll use that to adjust the timing parameter accordingly. We'll put the timer function into a separate file so that we can use it in other programs.

Listing 1 shows our first solution. Several features are worth mentioning. First, all of the timer specific material has been placed in the function 'delay'. Furthermore the delay parameter is not specific to the 8051 but simply specifies a delay in microseconds. The fact that the 8051's timers count up and require a negative value is hidden in the function. That makes our main program a bit more portable at the cost of a few more bytes (and microseconds) of instructions. It's a common tradeoff and one that requires a bit of decision making but that's why engineers get paid the big bucks.

Listing 1

```
#include <reg51.h>
void delay(int n){
 int t;
 t= -n;       //timers take a negative value
 TMOD= 0x01;  //16-bit mode 1
 TH0= t>>8;   //high half of n
 TL0= t&0xff; //low half of n
 TR0=1;       //start timer 0
 while(!TF0); //wait for timer flag
 TR0=TF0=0;   //stop timer and clear flag
}             //return

sbit SQRWV=P1^0;
void main(void){
 while(1){
  delay(-500);
  SQRWV= ~SQRWV;
  }
}
```

This 'module', which consists of both main() and delay(), takes 43 bytes of code. If we use the debugger to time our program, we find that the delay is actually 533 states, not 500 like we thought. That's because of the instruction execution overhead. We can add 33 to the call to delay (delay -500+33) and make the timing exact at the risk of lack of portability. Another approach would be to define a symbol for the -500, #define SW500 -500+33 for example, and do the tweaking in the symbol definition statement. At least that will tend to put all machine specific stuff into a single location where it's easier to find.

There is another common optimization that we can make. If we examine the code produced for delay(), we find that the compiler does some strange things. That's mainly because we are using a signed integer and the 8051 can only handle unsigned variables. Listing 2 shows the assembly code for delay.

Listing 2 Delay assembly code using an int for t.

```
0000 C3                 CLR     C
0001 E4                 CLR     A
0002 9F                 SUBB    A,R7
0003 FF                 MOV     R7,A
0004 E4                 CLR     A
0005 9E                 SUBB    A,R6
;---- Variable 't' assigned to Register 'R4/R5' ----
0006 AD07               MOV     R5,AR7
                                            ; SOURCE LINE # 5
0008 758901             MOV     TMOD,#01H
                                            ; SOURCE LINE # 6
000B FF                 MOV     R7,A
000C 33                 RLC     A
000D 95E0               SUBB    A,ACC
000F 8F8C               MOV     TH0,R7
                                            ; SOURCE LINE # 7
0011 AF05               MOV     R7,AR5
0013 EF                 MOV     A,R7
0014 F58A               MOV     TL0,A
                                            ; SOURCE LINE # 8
0016 D28C               SETB    TR0
0018           ?C0001:
                                            ; SOURCE LINE # 9
0018 308DFD             JNB     TF0,?C0001
001B           ?C0002:
                                            ; SOURCE LINE # 10
001B C28D               CLR     TF0
001D C28C               CLR     TR0
                                            ; SOURCE LINE # 11
001F 22                 RET
```

If we simply change t to an unsigned int, we end up with a module that only takes 35 bytes of code and a total delay time of 525 states. Listing 3 shows why. The first six lines take the two's complement of n, which is held in the register pair R6,R7. This is accomplished by subtracting n from 0 using double precision arithmetic. Since we don't do anything else with t, the compiler leaves the high half of -n in A and uses it to load TH0. About the only really dumb thing the compiler does here is to move R7 through the ACC instead of loading it directly into TL0.

Listing 3 Assembly code for delay using an unsigned int.

```
0000 C3                 CLR     C
0001 E4                 CLR     A
0002 9F                 SUBB    A,R7
0003 FF                 MOV     R7,A
0004 E4                 CLR     A
0005 9E                 SUBB    A,R6
0006 758901             MOV     TMOD,#01H
0009 F58C               MOV     TH0,A
000B EF                 MOV     A,R7
000C F58A               MOV     TL0,A
```

The shift and bitwise and operators were used so that the compiler would do these operations inline. Another approach would be to use the expressions TH0=t/256; and TL0=t % 256; however the compiler generates a call to a fairly large library routine in this case. Using the more direct TH0=-n>>8 actually uses

a bit more code that using the temporary variable t. You need to be careful with operator precedence. The expression TH0=-n>>8 generates a correct result while TH0=-(n>>8) does not unless the low half of n is equal to zero. This is due to the way the two's complement works.

Summary

This example has shown:

- How to use timer 0 in mode 1 to generate time delays
- How to adjust delay values to take into account instruction execution overhead
- How to extract the high and low bytes of a 16-bit integer to load timer registers.
- How use of unsigned variables are more efficient for the 8051

Questions

Answer the following questions.

1. Calculate the values of -500>>8 and -(500>>8). Are they the same? If not, why not? Be specific.
2. Calculate the values of -512>>8 and -(512>>8). Are they the same?