EE 354 **Fall 2016**

Notes on 8051 Interrupts

A standard 8051 has five interrupt sources. Two of these are for external interrupts called $\overline{\text{int0}}$ and $\overline{\text{int1}}$, two more are for the timers, and the last is for the serial port. Each of the interrupts can be individually turned on or off using a mask register. There is also a global interrupt flag which can turn off all interrupts.

For interrupts which occur simultaneously some priority must be established. The standard 8051 allows for two priority banks and any of the five interrupts can be placed in either of these banks. Within each bank the priority is established by the order in which the interrupts are polled in hardware. Each of the interrupts has an interrupt vector in lower code memory. Figure 1 below shows the standard interrupt structure of an 8051.

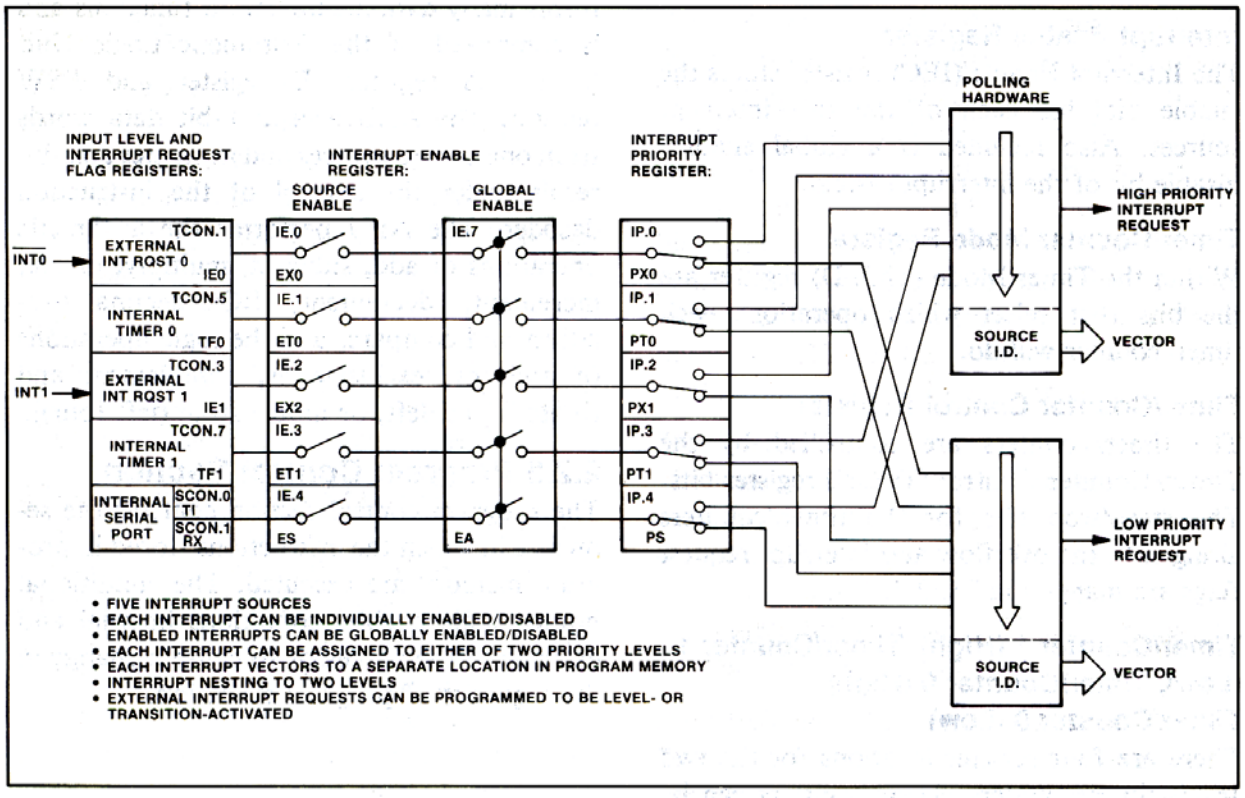

Figure 1
Interrupt structure of a classic 8051 processor

The AT89C51CC03 processor extends the standard interrupt system to include 9 interrupt vectors as shown in Figure 2 below. The interrupt priority system has also been modified. The 9 interrupt sources include: two external interrupts, three timer interrupts, a serial port interrupt, a programmable counter array (PCA) interrupt, and a timer overrun interrupt on the A to D converter.

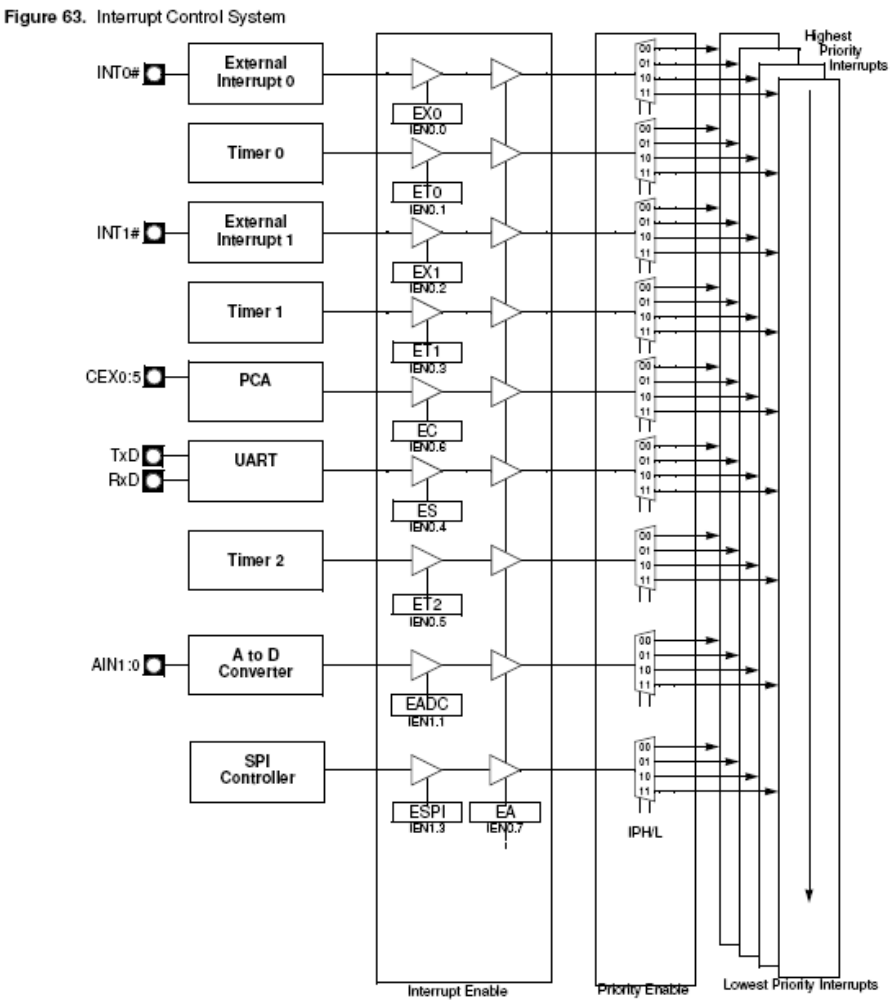

Figure 2
Interrupt system for the AT89C51CC03.

Each interrupt on the AT89C51CC03 can be enabled or disabled by setting or clearing a bit in the interrupt enable register. The interrupt enable register is actually two registers called IEN0 and IEN1. These two registers and their bit definitions are shown in Figure 3. Note that bit IEN0.7 is a global enable bit which can enable or disable all 0 interrupts simultaneously.

On interrupt systems it is necessary to determine interrupt priorities. Interrupts with a higher priority are permitted to interrupt other interrupts and get precedence when two interrupts happen simultaneously. The priority of an interrupt is determined by the interrupt priority register which can be set by the user's program. As shown in Figure 2 there are four priority levels and within each level the interrupt priority is established by the polling order. For example, if external interrupt 0 and timer interrupt 0 are both put in priority bank 0 then the external interrupt will have priority since it is polled first. The priority bank level is established by two bits called IPH.x and IPL.x where x is the bit position in the priority registers. The registers are shown in Figure 4.

Table 65. IEN0 Register

IEN0 (S:A8h)
Interrupt Enable Register

7	6	5	4	3	2	1	0
EA	EC	ET2	ES	ET1	EX1	ET0	EX0

Bit Number	Bit Mnemonic	Description
7	EA	**Enable All Interrupt bit** Clear to disable all interrupts. Set to enable all interrupts. If EA=1, each interrupt source is individually enabled or disabled by setting or clearing its interrupt enable bit.
6	EC	**PCA Interrupt Enable** Clear to disable the PCA interrupt. Set to enable the PCA interrupt.
5	ET2	**Timer 2 Overflow Interrupt Enable bit** Clear to disable Timer 2 overflow interrupt. Set to enable Timer 2 overflow interrupt.
4	ES	**Serial Port Enable bit** Clear to disable serial port interrupt. Set to enable serial port interrupt.
3	ET1	**Timer 1 Overflow Interrupt Enable bit** Clear to disable timer 1 overflow interrupt. Set to enable timer 1 overflow interrupt.
2	EX1	**External Interrupt 1 Enable bit** Clear to disable external interrupt 1. Set to enable external interrupt 1.
1	ET0	**Timer 0 Overflow Interrupt Enable bit** Clear to disable timer 0 overflow interrupt. Set to enable timer 0 overflow interrupt.
0	EX0	**External Interrupt 0 Enable bit** Clear to disable external interrupt 0. Set to enable external interrupt 0.

Reset Value = 0000 0000b
bit addressable

Table 66. IEN1 Register

IEN1 (S:E8h)
Interrupt Enable Register

7	6	5	4	3	2	1	0
-	-	-	-	ESPI	-	EADC	-

Bit Number	Bit Mnemonic	Description
7	-	**Reserved** The value read from this bit is indeterminate. Do not set this bit.
6	-	**Reserved** The value read from this bit is indeterminate. Do not set this bit.
5	-	**Reserved** The value read from this bit is indeterminate. Do not set this bit.
4	-	**Reserved** The value read from this bit is indeterminate. Do not set this bit.
3	ESPI	**SPI Interrupt Enable bit** Clear to disable the SPI interrupt. Set to enable the SPI interrupt.
2	-	**Reserved** The value read from this bit is indeterminate. Do not set this bit.
1	EADC	**ADC Interrupt Enable bit** Clear to disable the ADC interrupt. Set to enable the ADC interrupt.
0	-	**Reserved** The value read from this bit is indeterminate. Do not set this bit.

Reset Value = xxxx 0x0xb
bit addressable

Figure 3

IEN0 and IEN1 interrupt enable registers.

Table 67. IPL0 Register

IPL0 (S:B8h)
Interrupt Enable Register

7	6	5	4	3	2	1	0
-	PPC	PT2	PS	PT1	PX1	PT0	PX0

Bit Number	Bit Mnemonic	Description
7	-	**Reserved** The value read from this bit is indeterminate. Do not set this bit.
6	PPC	**PCA Interrupt Priority bit** Refer to PPCH for priority level
5	PT2	**Timer 2 Overflow Interrupt Priority bit** Refer to PT2H for priority level.
4	PS	**Serial Port Priority bit** Refer to PSH for priority level.
3	PT1	**Timer 1 Overflow Interrupt Priority bit** Refer to PT1H for priority level.
2	PX1	**External Interrupt 1 Priority bit** Refer to PX1H for priority level.
1	PT0	**Timer 0 Overflow Interrupt Priority bit** Refer to PT0H for priority level.
0	PX0	**External Interrupt 0 Priority bit** Refer to PX0H for priority level.

Reset Value = X000 0000b
bit addressable

Table 68. IPL1 Register

IPL1 (S:F8h)
Interrupt Priority Low Register 1

7	6	5	4	3	2	1	0
-	-	-	-	SPIL	-	PADCL	-

Bit Number	Bit Mnemonic	Description
7	-	**Reserved** The value read from this bit is indeterminate. Do not set this bit.
6	-	**Reserved** The value read from this bit is indeterminate. Do not set this bit.
5	-	**Reserved** The value read from this bit is indeterminate. Do not set this bit.
4	-	**Reserved** The value read from this bit is indeterminate. Do not set this bit.
3	SPIL	**SPI Interrupt Priority Level Less Significant Bit** Refer to SPIH for priority level.
2	-	**Reserved** The value read from this bit is indeterminate. Do not set this bit.
1	PADCL	**ADC Interrupt Priority Level Less Significant Bit** Refer to PSPIH for priority level.
0	-	**Reserved** The value read from this bit is indeterminate. Do not set this bit.

Reset Value = XXXX 0X0Xb
bit addressable

Figure 4

Interrupt priority registers. Note that this figure shows only the IPL register. There is also an IPH register which has similar bits in corresponding positions. To set a priority level it is necessary to set both IPH and IPL for a given interrupt as shown in Figure 5.

Table 63. Priority Level Bit Values

IPH.x	IPL.x	Interrupt Level Priority
0	0	0 (Lowest)
0	1	1
1	0	2
1	1	3 (Highest)

Figure 5

Bit values which determine interrupt priority

External interrupts and edge triggering
The two external interrupts can be set to be either edge triggered or level sensitive. This is governed by two bits in TCON called IT0 and IT1 for external interrupts 0 and 1 respectively. If IT0, for example is 0 then external interrupt 0 is triggered by a low level on the interrupt pin. If IT0 is set to 1 then the interrupt is triggered by a falling edge on the interrupt pin.

The falling edge, for the edge triggered case, sets an internal flip-flop to hold the interrupt until it is serviced. Once the interrupt is serviced the flip-flop is automatically turned off. For the level sensitive case there is no internal flip-flop and it is the responsibility of the interrupt service routine to get the external device to remove the external interrupt before returning to the main program.

Interrupt Latency
Interrupt latency is defined as the time between when an interrupt occurs and the time when the interrupt service routine begins. Instructions in a computer typically complete the following steps: 1) fetched from memory 2) decoded 3) fetch and decode the operands 4) execute the instruction and 5) check for interrupts. These steps are shown schematically in Figure 6 below.

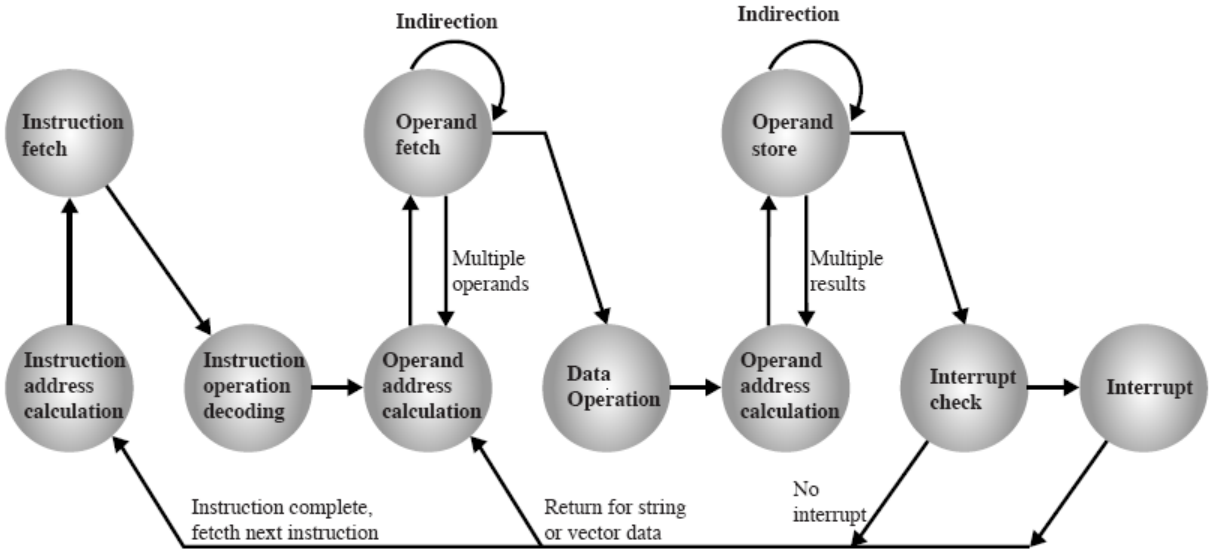

Figure 6
Flow chart for fetching and executing an instruction.

This set of operations can be done in microcode or as part of the sequential machine that runs the CPU. Note that the checking for interrupts is the very last thing that gets done. The reason for this is that the state of the machine must be saved when an interrupt occurs and this cannot be done in mid-instruction. Almost all machines have registers and flip-flops internal to the operations of the machine that are not user accessible. Saving the state of the machine in mid-instruction would mean that these registers would have to be saved as well as the location within the instruction where it was interrupted. Instead, computers check for interrupts at the end of an instruction where the state of the machine is stored in user accessible registers that can be saved.

Also the program counter keeps track of which instruction is being executed to make return from an interrupt possible.

If an interrupt occurs at the beginning of an instruction cycle it will be ignored until the end of the instruction which adds to the interrupt latency. For the AT89C51CC03 the multiply instruction takes the longest time to execute at 4 machine cycles. It requires a full machine cycle to resolve the interrupt priorities and 2 more machine cycles to push the program counter onto the stack. Thus, if an interrupt occurs immediately after a multiply instruction begins the ISR will not start until 4 + 1 + 2 = 7 machine cycles later.

But this is not quite the worst case. The 8051 has a special instruction for returning from interrupts called RETI (Return from Interrupt). This instruction pops the top of the stack into the program counter just as a normal return does but it also restores the interrupt priority system. The RETI instruction is the only instruction which is not interruptible. Thus, the microcode for this instruction does not check for the interrupt flag. The reason for this is that it guarantees that if any system becomes bogged down in interrupt requests there will be at least one instruction executed in the main program between each service routine. So the worst case interrupt latency will occur if the interrupt occurs at the beginning of the execution of a RETI instruction to return to the main program to do a multiply instruction. The RETI instruction requires 2 machines cycles so the worst case interrupt latency on the AT89C51CC03 requires 2 + 4 + 1 + 2 = 9 machine cycles.

Interrupts in C-Code

To set up an interrupt in C you need to do all of the following:

1. Set the mask bit to enable the particular interrupt you are using.
2. Set the EA (global enable) bit to 1 to enable the interrupt system
3. Set the edge/level flag for external interrupts.
4. Write the interrupt service routine in the following format:

```
void ISRName(void) interrupt int# using R#
   {
   }
```

   Where ISRName is the name you have chose for the interrupt service routine, int# is the interrupt number, and R# is the register bank number.

Note that the interrupt number is a bit confusing since it has nothing to do with the interrupt function. Thus interrupt 1 is external interrupt 0, interrupt 2 is Timer 0 interrupt, interrupt 3 is external interrupt 1, etc. To keep this straight keep in mind that the compiler uses the interrupt number to determine where to place the interrupt vector. The vector gets located at 8 x int# + 3. So, for example, interrupt 1 is at 8 x 1 + 3 = 11 which is the location of the vector for the Timer0 interrupt. See Figure 7 for the interrupt you want, locate its vector in the table, and determine the interrupt number from the equation. For example, if I want to get an interrupt for the analog to digital converter I find that it is at vector 043H in memory which is 67 decimal. So 8 x (int#) + 3 = 67 gives int# = 8.

Table 64. Interrupt priority Within level

Interrupt Name	Interrupt Address Vector	Priority Number
external interrupt (INT0)	0003h	1
Timer0 (TF0)	000Bh	2
external interrupt (INT1)	0013h	3
Timer1 (TF1)	001Bh	4
PCA (CF or CCFn)	0033h	5
UART (RI or TI)	0023h	6
Timer2 (TF2)	002Bh	7
ADC (ADCI)	0043h	8
SPI interrupt	0053h	9

Figure 7
Interrupt numbers and interrupt vector addresses.

Example 1: Use timer 0 and timer 1with interrupts to produce a square wave on P3.2. The square wave should have a 2 msec high time and a 1 msec low time.

TMOD Register. We can use a 16-bit timer so we want mode 1. We are not using the gated input and we want the timer to count FT0 and FT1 clocks so all of the other bits are 0. Set TMOD to 0001 0001B = 11H.

TCON Register. In TCON we will need to set TR0 and TR1 to 1 to get timer 0 and time 1 to run.

Timer Register values for overflow. Setting CCKON to 01h will make FTClock = 28.2076MHz/6 = 4.701267 MHz and the processor core will be double clocked. This gives a period of 0.2127086 µsecond period. To get 1 millisecond = 1000 µseconds we need

$$\frac{1\,\text{count}}{0.2127086\,\text{microseconds}} x \frac{1000\,\text{microseconds}}{\text{millisecond}} = 4701\,\text{counts}$$

To get a 2 msec high time we will need a count of 2 x 4701 = 9402 counts.

Since this is a 16-bit timer it overflows when it runs from 65,535 to 65,536. So we want to initialize the timer to 65,536 – 4701 = 60835 which corresponds to 0EDA3H. So we want to load TH1 with 0EDH and TL1 with A3H. Timer 0 will count 65,536 – 9402 = 56134 which corresponds to 0DB46H. We will make TH0 = 0DBh and TL0 = 046h.

Enabling the interrupt. Timer 0 uses interrupt 1so we need to enable the timer 0 interrupt by setting ET0 = 1. We must also set the global interrupt enable bit to 1 using EA = 1. These two enables are in the interrupt enable register IE at location 0A8H. (This is a SFR.) Interrupt 1 has a vector at location 0BH so we must place a jump to the interrupt service

routine (ISR) at this location. Finally we must write the ISR. For this application the ISR is going to toggle a bit on P3.2 so it's trivial.

```
//TimerInts.c
// This program produces a square wave with a 2 msec high time and 1 msec low
//  time on P3.2 using timer 0 and timer 1 in an interrupt mode.
#include<REG51ac2.h>

void main(void)
   {CKCON = 0x01;   // x2 mode
    TMOD = 0x11;    //Timer 0 mode = not gated, internal clock, 16-bit mode
                    //Timer 1 mode = not gated, internal clock, 16-bit mode
    //For fosc = 28.2076MHz in x2 mode timer is clocked at 28.2076Mhz/6 =
4.701MHz
    // so period is 1/4.701Mhz = .2127086 usec.  To get 1 msec we need
    // 1000/.2127086 = 4701 counts.  65536 - 4701 = 60835 = 0xEDA3.
    // For 2 msec we need 9402 counts.  65536 - 9402 = 55134 = 0xDB46.

    TH0 = 0xDB;     //Timer 0 set to DB46 -> 55134
    TL0 = 0x46;

    TH1 = 0xED;     //Timer 1 set to EDA3 -> 60835
    TL1 = 0xA3;
    TR0 = 1;
    ET0 = 1;        //Timer 0 interrupt enable
    EA = 1;         //Global interrupt enable
    while(1);
   }
//
void T0Int() interrupt 1 using 1
  {P3 = P3 | 4;  //bit P3.2 to 1
   TR0 = 0;      //Turn timer 0 off
   TH1 = 0xDB;   //Timer 1 set to DB46 = 55134
   TL1 = 0x46;
   TR1 = 1;      //Turn timer 1 on
   ET0 = 0;      //Timer 0 interrupt off
   ET1 = 1;      //Timer 1 interrupt on
  }
//
void T1Int(void) interrupt 3 using 1
  {P3 = P3 & 0xFB;   //bit P3.2 to 0
   TR1 = 0;          //Turn timer 1 off
   TH0 = 0xED;       //Timer 0 set to EDA3 = 60835
   TL0 = 0xA3;
   TR0 = 1;          //Turn timer 0 on
   ET1 = 0;          //Timer 1 interrupt off
   ET0 = 1;          //Timer 0 interrupt on
  }
```

Figure 8

The c code for running Timer 1 for 1 msec and Timer 0 for 2 millisecond and toggling P3.2. This version uses an interrupt with Timer 0 and Timer 1.

Example 2: Use timer 0 with an interrupt to produce a 50 KHz square wave on P3.2. Timer 0 is used in auto reload mode.

Timer Register values for overflow: FT0Clock is running at 28.2076MHz/6 = 4.701267 MHz which corresponds to a 0.2127086 μsecond period. A 50 KHz square wave has a 0.02

millisecond period but there are two transitions per cycle so we need a 0.01 millisecond period for the timer.
To get 0.01 millisecond = 10 μseconds we need

$$\frac{1\,\text{count}}{0.2127086\,\text{microseconds}} x \frac{10\,\text{microseconds}}{\text{millisecond}} = 47\,\text{counts}$$

An 8-bit counter can count to 255 so we can use Timer 0 in the 8-bit auto-reload mode with TH0 loaded initially with 256 – 47 = 209 counts which corresponds to 0D1h

TMOD Register.
For 8-bit auto-reload we want Timer 0 in mode 2. We are not using the gated input and we want the timer to count FT0 so all of the other bits are 0. Set TMOD to 0000 0010B = 02H.

TCON Register. In TCON we will need to set TR0 to 1 to get timer 0 to run.

Enabling the interrupt. Timer 0 uses interrupt 1so we need to enable the timer 0 interrupt by setting ET0 = 1. We must also set the global interrupt enable bit to 1 using EA = 1. These two enables are in the interrupt enable register IE at location 0A8H. (This is a SFR.) Interrupt 1 has a vector at location 0BH so we must place a jump to the interrupt service routine (ISR) at this location. Finally we must write the ISR. For this application the ISR is going to toggle a bit on P0.0 so it's trivial.

```
//TimerReload.c
// This program produces a 50KHz square wave on P3.2 using
//  Timer 0 in the 8-bit autoreload mode.
//  50KHz corresponds to a 20usec period but interrupt complements
//  P3.2 at twice that rate so we need a 10 usec period for T0.
#include<REG51ac2.h>
void SqWave();
void main(void)
   {CKCON = 0x01;   // x2 mode
    TMOD = 0x02;    //Timer 0 mode = not gated, internal clock, 8-bit, auto
reload
    //For fosc = 28.2076MHz in x2 mode timer is clocked at 28.2076Mhz/6 =
4.701MHz
    // so period is 1/4.701Mhz = .2127086 usec.  To get 10 usec we need
    // 10/.2127608 = 47 counts.  256 - 47 = 209 = 0xD1
    TH0 = 0xD1;     //Timer 0 high set to 209 for 10 micro-seconds
    TR0 = 1;        //Timer 0 run control bit in TCON
    ET0 = 1;        //Timer 0 interrupt enable
    EA = 1;         //Global interrupt enable
    while(1);
   }
//
//Timer 0 comes in on Interrupt 1.
void SqWave() interrupt 1 using 1
  {P3 = P3 ^ 4;  //Exclusive or with 00000100 for bit 3.2
  }
```

Figure 9
The c code for running Timer 0 in an 8-bit auto reload mode and toggling P3.2 every 10 microseconds.