Interrupts, timers and counters

Posted on May 10, 2008, by Ibrahim KAMAL, in Micro-controllers, tagged

Most microcontrollers come with a set of 'ADD-ONs' called peripherals, to enhance the functioning of the microcontroller, to give the programmer more options, and to increase the overall performance of the controller. Those features are principally the timers, counters, interrupts, Analog to digital converters, PWM generators, and communication buses like UART, SPI or I2C. The 89S52 is not the most equipped micro-controller in terms of peripherals, but never the less, the available features are adequate to a wide range of applications, and it is one of the easiest to learn on the market.

Introduction to 89S52 Peripherals

Figure 4.1 below shows a simplified diagram of the main peripherals present in the 89S52 and their interaction with the CPU and with the external I/O pins. You can notice that there are three timers/Counters. We use the expression "Timer/Counter" because this unit can be a ***counter*** when it counts external pulses on it's corresponding pin, and it can be a ***timer*** when it counts the pulses provided by the main clock oscillator of the microcontroller. Timer/Counter two is a special counter, that does not behave like the two others, because it have a couple of extra functionality.

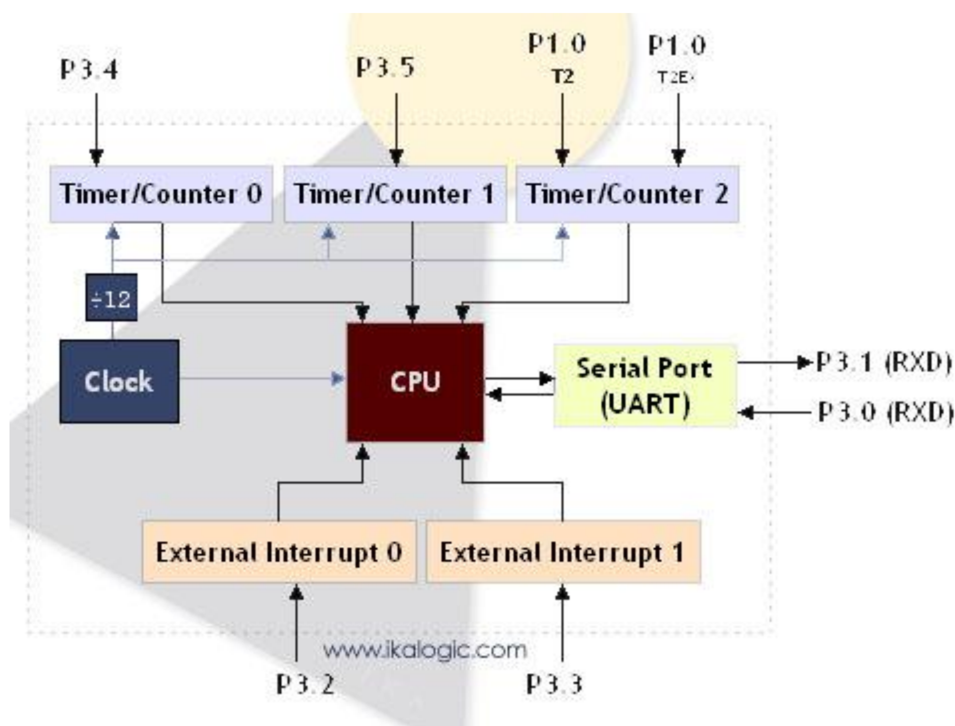

figure 4.1

The serial port, using a UART (Universal Asynchronous Receive Transmit) protocol can be used in a wide range of communication applications. With the UART provided in the 89S52 you can easily communicate with a serial port equipped computer, as well as communicate with another microcontroller. This last application, called Multi-processor communication, is quite interesting, and can be easily implemented with two 89S52 microcontrollers to build a very powerful multi-processor controllers.

If all the peripherals described above can generate interrupt signals in the CPU according to some specific events, it can be useful to generate an interrupt signal from an external device, that may be a sensor or a Digital to Analog converter. For that purpose there are two External Interrupt sources (INT0 and INT1).

This was a presentation of the available peripheral features in a 89S52 microcontroller. **Through this tutorial, we are going to study how to setup and use external interrupts and the two standard timers (T0 and T1)**. For simplicity, and to keep this tutorial a quick and straight forward one, The UART and the Timer/Counter 2 shall be discussed in separate tutorials.

External Interrupts

Let's start with the simplest peripheral which is the external interrupt, which can be used to cause interruptions on external events (a pin changing its state from 0 to 1 or vice-versa). In case you don't know, interruption is a mean of stopping the flow of a program, as a response to a certain event, to execute a small program called 'interrupt routine'.

As you noticed in ***figure 4.1***, in the 89S52, there are two external interrupt sources, one connected to the pin P3.2 and the other to P3.3. They are configured using a number of SFRs (Special Function Registers). Most of those SFRs are shared by other peripherals as you shall see in the rest of the tutorial.

- The IE register

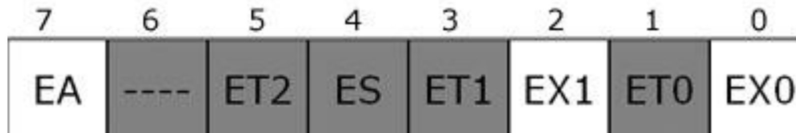

figure 4.2.A

The first register you have to configure (by turning On or Off the right bits) is the IE register, shown in ***figure 4.2.A***. IE stands for 'Interrupt Enable', and it is used to allow different peripherals to cause software interruption. To use any of the interrupts, the bit EA (Enable ALL) must be set to 1, then, you have enable each one of the interrupts to be used with its individual enable bit. For the external interrupts, the two bits EX0 and EX1 are used for External Interrupt 0 and External Interrupt 1.

Using the C programming language under KEIL, it is extremely simple to set those bits, simply by using their name as any global variables, Using the following syntax:

```
EA = 1;

EX0 = 1;

EX1 = 1;
```

The rest of the bits of IE register are used for other interrupt sources like the 3 timers overflow (ETx) and the serial interface (ES).

- The TCON register

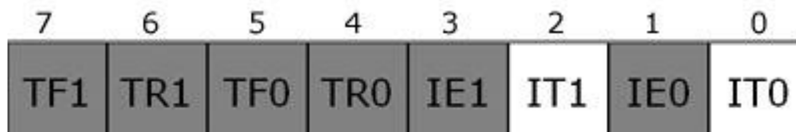

figure 4.2.B

Similarly, you have to set the bits IT0 and IT1 in the TCON register, shown in figure 4.2.B. The bits IT0/IT1 are used to configure the type of signal on the corresponding pins (P3.2/P3.3) that generated an interrupt according to the following table:

IT0/IT1 = 1	External interrupt caused by a falling edge signal on P3.2/P3.3
IT0/IT1 = 0	External interrupt caused by a low level signal on P3.2/P3.3

If IT0 or IT1 is set to 0, an interruption will keep reoccurring as long as P3.2 or P3.3 is set to 0. This mode isn't easy to mange, and most programmers tends to use external interrupts triggered by a falling edge (transition from 1 to 0).

Again, this register is ‘bit addressable’ meaning you can set or clear each bit individually using their names, like in the following example:

```
IT0 = 1;

IT1 = 1;
```

- Example Program
  Here is an example program to demonstrate the External Interrupt peripheral feature of the 89s52. **We are going to build a simple digital low pass filter**.

  External Interrupt 0 is set in ‘Falling Edge’ mode, and is used to check for noise on a signal and reset a counter in case noise is detected. Since the noise is interpreted by digital devices as a succession of high and low levels, any ‘high to low’ level transition is easily detected in the ‘Falling edge’ mode.

  If the counter reaches a pre-calibrated value, then the signal is considered to be stable, and can be sampled, otherwise, if the signal bounces between 0 and 1 before the counter reaches the pre-defined value, the external interrupt resets the counter, and the signal is not taken in account.

  Since we will be using External Interrupt 0, the signal to be checked for noise and sampled is imperatively connected to pin P3.2, and the clean, filtered output signal is to be generated on P1.0.

```
// Include standard headers

#include <REGX52.h>

#include <math.h>


unsigned int counter; //Variable used for our counter


setup_interrupts () // Function to setup the External interrupt 0 in the
required mode

{

    EA = 1;

    EX0 = 1;

    IT0 = 1;

}
```

```
filter () interrupt 0 //The function the be executed when external interrupt
occurs

{

    counter = 0; //Reset the counter to 0

}


main()

{

    time_constant = 40000; //Define the response time of our filter

    setup_interrupts (); //setup the External interrupt

    while(1)

    {

        if (counter < time_constant) // Count until the pre-defined
time_constant

        {

        counter++;

        }


        if (counter == time_constant) // if the counter was not interrupted
by any noise,

        {

        P1_0 = P3_2; // output the valid signal on P1_0

        }

    }

}
```

- Exercise:
  To make sure you've correctly assimilated the functioning of the external interrupts, try to build a program that decodes the pulses coming from an incremental encoder to determine an absolute position.

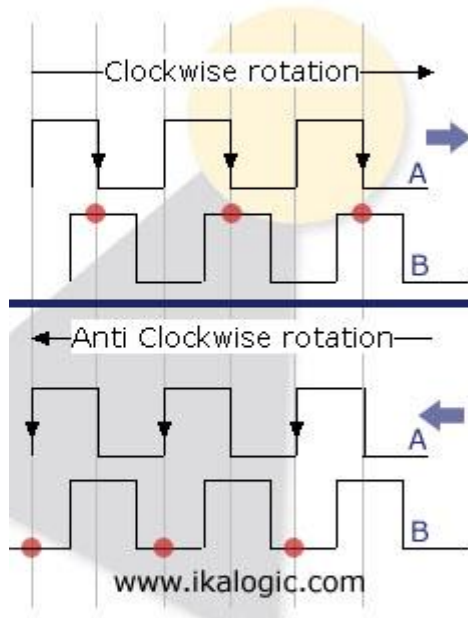

figure 4.2.C

Incremental encoder are rotational encoder that generate two square waves, shifted by 90 degrees (or by a quarter of a period), as you can see in ***figure 4.2.C***. The main idea of operation is that for a same direction of rotation, the falling edges of signal A will occur at the same time with respect to the signal B.

In other words, during clockwise rotation, the falling edge of signal will always occur while signal B is at high level. On the other hand, during counterclockwise rotation, the falling edges of signal A will always occur while signal B is at a low level.

This mechanism can be used to detect the 'quantity' of rotation in number of pulses as well as direction of the rotation

Using this method, build a program to decode the signals coming from an incremental encoder, and update the position of the encoder at each falling edge. You will need only one External interruption.

You can try your source code by simulating it in KEIL IDE, or by testing it directly on your breadboard. IF you can't find the solution, you can seek for help in the forums.

Timer/Counter

For this part, I'll often use the notation 1/0 adjacent to a register name, which means that there are two of that register, one of them for timer/counter 0, and the other for timer/counter 1, and that the description applies to both of them.

The timer is a very interesting peripheral, that is imperatively present in every microcontroller. It can be used in two distinct modes:

- **Timer:** Counting internal clock pulses, which are fixed with time, hence, we can say that it is very precise timer, whose resolution depends on the frequency of the main CPU clock (note that CPU clock equals the crystal frequency over 12).

- **Counter:** Counting external pulses (on the corresponding I/O pin), which can be provided by a rotational encoder, an IR-barrier sensor, or any device that provide pulses, whose number would be of some interest.
  Sure, the CPU of a microcontroller could provide the required timing or counting, but the timer/counter peripheral relieves the CPU from that redundant and repetitive task, allowing it to allocate maximum processing power for more complex calculations.

  So, like any other peripheral, a Timer/Counter can ask for an interruption of the program, which – if enabled – occurs when the counting registers of the Timer/Counter are full and overflow. More precisely, the interruption will occur at the same time the counting register will be reinitialized to its initial value.

  So to control the behavior of the timers/counters, a set of SFR are used, most of them have already been seen at the top of this tutorial.

- The IE register
  First, you have to Enable the corresponding interrupts, but writing 1′s to the corresponding bits in the IE register. The following table shows the names and definitions of the concerned bits of the IR register (you can always take a look at the complete IE register in ***figure 4.2.A***):

EA	Enable All interrupts
ET2	Enable Timer 2 interrupts (*will not be treated in this tutorial*)
ET1	Enable Timer 1 interrupts
ET0	Enable Timer 0 interrupts

  You can access those special bits by their names, as simply as it seems, example:

```
ET0 = 1;
```

- The TCON register
  The TCON register is also shared between more than one peripherals. It can be used to configure timers or, as you saw before, external interrupts. The following table shows the names and definitions of the concerned bits of the TCON register (available in ***figure 4.2.B***):

TF1	Overflow interrupt flag, used by the processor.
TR1	Timer/counter 1 RUN bit, set it to 1 to enable the timer to count, 0 to stop counting.
TF0	Overflow interrupt flag, used by the processor.
TR0	Timer/counter 0 RUN bit, set it to 1 to enable the timer to count, 0 to stop counting.

  As the IE register, TCON is also bit-addressable, so you can set its bit using its names, like we did before. Example:

```
TR0 = 1;
```

- The TMOD register
  Before explaining the TMOD register, let us agree and make it clear that the register IS NOT BIT-ADDRESSABLE, meaning you have to write the 8 bits of the register in a single instruction, by coding those bits into a decimal or hexadecimal number, as you shall see later.

  So, as you can see in ***figure 4.2.C***, the TMOD register can be divided into two similar set of bits, each group being used to configure the mode of operation of one of the two timers.

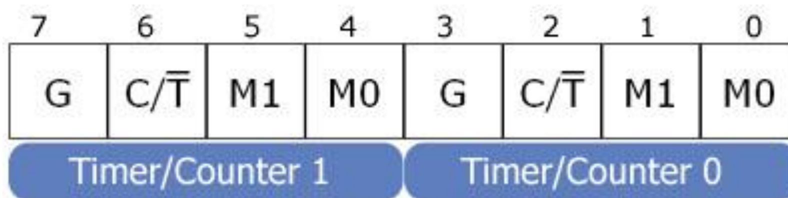

figure 4.2.C

For the a given Timer/Counter, the corresponding bits of TMOD can be defined as in the following table:

G	Gate signal. For normal operation clear this bit to 0.If you want to use the timers to capture external events's length, set it to 1, and the timer 1/0 will stop counting when External Interrupt 1/0 pin is low (set to 0 V). Note that this feature involves both a timer and an external interrupt, It you're responsibility to write the code to manage the operation of those two peripherals.
C/T'	Set to 1 to use the timer/counter 1/0 as a Counter, counting external events on P3_4/P3_5, cleared to 0 to use it as timer, counting the main oscillator frequency divided by 12.
M1 M0	Timer MODE: Those two last bits combine as 2 bit word that defines the mode of operation, defined as the table below.

- Timer/counter modes of operation
  Each timer/counter has two SFR called TL0 and TH0 (for timer/counter0) and TL1 and TH1 (for timer/counter 1). TL stands for timer LOW, and is used to store the lower bits of the number being counted by the timer/counter. TH stands for TH, and is used to store the higher bits of the number being counted by the timer/counter.

M1	M0	Mode	Description
0	0	0	Only TH0/1 is used, forming an 8bit timer/counter. Timer/counter will count up from the value initially stored in TH0/1 to 255, and then overflow back to 0. If an interrupt is enabled, an interrupt will occur upon overflow. If used as timer, pulses from the processor are divided by 32 (after being divided by 12). The result is the main oscillator frequency divided by 384. If used as counter, external pulses are only divided by 32.
0	1	1	Both TH0/1 and TL0/1 are used, forming a 16 bit timer/counter. Timer/counter will count up from the 16 bit value initially stored in TH0/1 and TL0/1 to 65535, and then

			overflow back to 0. If an interrupt is enabled, an interrupt will occur upon overflow. If used as timer, pulses from the processor are only divided by 12. If used as counter, external pulses are not divided, but the maximum frequency that can be accurately counted equals the oscillator frequency divided by 24.
1	0	2	TL0/1 is used for counting, forming an 8 bit timer/counter. TH0/1 is used to hold the value to be restored in TL upon overflow. Timer/counter will count up from the 8 bit value initially stored in TL0/1 and to 255, and then overflow, setting the value of TH0/1 in TL0/1. This is called the auto-reload function. If an interrupt is enabled, an interrupt will occur upon overflow. If used as timer, pulses from the processor are only divided by 12. If used as counter, external pulses are not divided, but the maximum frequency that can be accurately counted equals the oscillator frequency divided by 24.
1	1	3	This mode is beyond the scope of this tutorial.

Timer modes 1 and 2 are the most used in 8051 microcontroller projects, since they offer a wide range of possible customization.

Exercise project

To conclude this tutorial, a simple project is proposed, to help you assimilate the functioning of the timers, counter and interrupts.

Consider the following problem. A motor is being operated by an outdated motor controller. We want to add some security to the system, by stopping the whole system in case the motor heats up too much, or turns too fast. A temperature sensor is already set up and give a low signal 0 when the temperature is too high, and an optical encoder output a pulse for each revolution of the motor. We need to write the code to stop the motor incase it heats up or in case it reached 10 000 r.p.m.

Considering that the temperature sensor is connected to P3.2 (External interrupt 0), the encoder is connected to P3.5 (Timer/Counter 1), that the system can be stopped by a high level signal on P1_0, and that we are using a 24MHz crystal oscillator, the software for that controller would be like the following:

```
// Include standard headers

#include <REGX52.h>

#include <math.h>

#define limit 12

#define stop_signal P1_0


unsigned char sub_counter;
```

```
setup_peripherals(){

   //setup external interrupt

   EA = 1;

   EX0 = 1;

   IT0 = 1;


   TMOD = 0x52; // timer 0 mode 2, counter 1 mode 1


   //setup timer 0

   TR0 = 1;

   TH0 = 5; //makes the timer to overflow every 125 uS (divide by 250).

   ET0 = 1;


   //setup timer 1

   TR1 = 1 ;

}


timer_0 () interrupt 1{

   sub_counter++ ;

   if (sub_counter == 10){ // divide the overflow frequency further more by
10

      sub_counter = 0;

      // this part is executed every 1.25 mS

      if ((TL1 + (TH1 * 256)) > limit){ // 12/0.00125 = 9600 (~ 10000)s

         //Stop the motor

         stop_signal = 1;

         TL1 = 0;

         TH1 = 0;
```

```
      }
   }
}

over_heat_alarm () interrupt 0{
   stop_signal = 1;
}

void main(){
   stop_signal = 0; // the motor runs normally
   setup_peripherals();
   while(1){
      // Do nothing, the whole program is carried out by interrupts!
   }
}
```

You should be able to understand and calculate all the choices of timings in that source code, especially the values related with the timer 0 that have to be executed at a very precise time. For more information about RPM measurements, see this project: Contact less digital tachometer.

Source: http://www.ikalogic.com/part-4-interrupts-timers-and-counters/