

Microcontroller interrupts in frequency controlling

C Panoiu¹, R Rob¹ and M Panoiu¹

¹ Politehnica University of Timisoara, Department of Electrical Engineering and Industrial Informatics, 5 Revolution Street, 331128 Hunedoara, Romania

E-mail: caius.panoiu@gih.upt.ro

Abstract. Present paper represents an educational software application that permits to measure the signals frequency. The application is accomplished in MicroC compiler that also permit to user to program a microcontroller PIC18F45K22 using a developing board EasyPICv7. An important issue of present application is to measure the frequency of an external signal that is acquired by an input pin of the microcontroller. The frequency value is displayed on the seven segment display provided by the developing board.

1. Introduction

PIC microcontrollers are a very useful and versatile tool for use in many electronic projects. They are very inexpensive and easy to find. They are also very powerful and many are capable of speeds up to 64 MIPS using the internal oscillator block, about 16 times faster than most comparable AVR microcontrollers. PICs are also easy to program, however getting the project set up can often be tricky.

An interrupt is a means of interrupting the main program flow in response to an event, so that the event can be handled. The event (referred to an interrupt source) can be internal to the PIC, such as a timer overflowing, or external, such as a change on an input pin.

There are 19 registers used to control interrupt operation. In general, interrupt sources have three bits to control their operation [1], [5]. They are presented in the followings:

- Flag bit to indicate that an interrupt event occurred.
- Enable bit that allows program execution to branch to the interrupt vector address when the flag bit is set.
- Priority bit to select high priority or low priority.

Interrupt Priority

The interrupt priority feature [2] is enabled by setting the IPEN bit of the RCON register. When interrupt priority is enabled the GIE/GIEH and PEIE/GIEL global interrupt enable bits of Compatibility mode are replaced by the GIEH high priority, and GIEL low priority, global interrupt enables [6]. When set, the GIEH bit of the INTCON register enables all interrupts that have their associated IPRx register or INTCONx register priority bit set (high priority). When clear, the GIEH bit disables all interrupt sources including those selected as low priority. When clear, the GIEL bit of the INTCON register disables only the interrupts that have their associated priority bit cleared (low priority). When is set, the GIEL bit enables the low priority sources when the GIEH bit is also set. When the interrupt flag enables bit and appropriate Global Interrupt Enable (GIE) bit are all set, the interrupt vector will immediately address 0008h for high priority, or 0018h for low priority, depending



on level of the interrupting source's priority bit. Individual interrupts can be disabled through their corresponding interrupt enable bits.

Interrupt Response

When an interrupt is responded to, the Global Interrupt Enable bit is cleared to disable further interrupts. The GIE/GIEH bit is the global interrupt enable when the IPEN bit is cleared. When the IPEN bit is set, enabling interrupt priority levels, the GIEH bit is the high priority global interrupt enable and the GIEL bit is the low priority global interrupt enable. High priority interrupt sources can interrupt a low priority interrupt. Low priority interrupts are not processed while high priority interrupts are in progress. The return address is pushed onto the stack and the PC is loaded with the interrupt vector address (0008h or 0018h). Once in the Interrupt Service Routine, the source(s) of the interrupt can be determined by polling the interrupt flag bits in the INTCONx and PIRx registers. The interrupt flag bits must be cleared by software before re-enabling interrupts to avoid repeating the same interrupt. The "return from interrupt" instruction, RETFIE, exits the interrupt routine and sets the GIE/GIEH bit (GIEH or GIEL if priority levels are used), which re-enables interrupts. For external interrupt events, such as the INT pins or the PORTB interrupt-on-change, the interrupt latency will be three to four instruction cycles. The exact latency is the same for one-cycle or two-cycle instructions. Individual interrupt flag bits are set, regardless of the status of their corresponding enable bits or the Global Interrupt Enable bit.

2. Application description

The application described here is able to acquire a digital signal provided by an external source and to count its pulses. Signal is acquired by port B of PIC18F45K22 microcontroller. The value of the measured frequency is displayed on the seven segments display connected on the development board. The display uses the A and D ports of PIC18F45K22 like in Figure 1 [8].

The soft application is based on generating of an interrupt to TMR0 timer. Using INTCON register, TMR0 generates one transition on each second. In these conditions, the application counts the external signal transitions during one second.

Therefore, the application accomplished in MicroC compiler [7] declares two variables: one for incrementing the transition counter (*cnt*) and one for incrementing the overflow counter (*ovrfw*).

Table 1 presents the INTCON register [2].

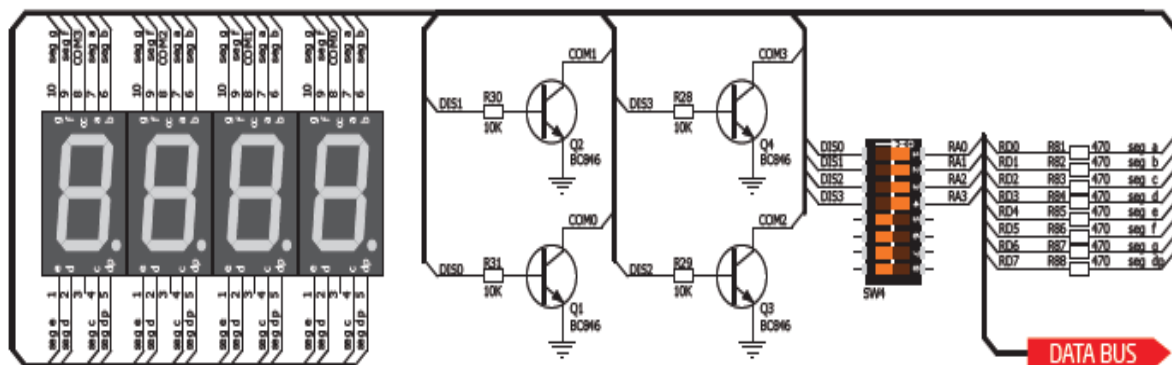


Figure 1. Connection of seven segments display.

Table 1. INTCON register configuration.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF
bit 7							bit 0

Data from Table 2 is detailed in the followings:

bit 7 **GIE/GIEH**: Global Interrupt Enable bit

When IPEN = 0:

1 = Enables all unmasked interrupts

0 = Disables all interrupts including peripherals

When IPEN = 1:

1 = Enables all high priority interrupts

0 = Disables all interrupts including low priority

bit 6 **PEIE/GIEL**: Peripheral Interrupt Enable bit

When IPEN = 0:

1 = Enables all unmasked peripheral interrupts

0 = Disables all peripheral interrupts

When IPEN = 1:

1 = Enables all low priority interrupts

0 = Disables all low priority interrupts

bit 5 **TMR0IE**: TMR0 Overflow Interrupt Enable bit

1 = Enables the TMR0 overflow interrupt

0 = Disables the TMR0 overflow interrupt

bit 4 **INT0IE**: INT0 External Interrupt Enable bit

1 = Enables the INT0 external interrupt

0 = Disables the INT0 external interrupt

bit 3 **RBIE**: Port B Interrupt-On-Change (IOCx) Interrupt Enable bit(2)

1 = Enables the IOCx port change interrupt

0 = Disables the IOCx port change interrupt

bit 2 **TMR0IF**: TMR0 Overflow Interrupt Flag bit

1 = TMR0 register has overflowed (must be cleared by software)

0 = TMR0 register did not overflow

bit 1 **INT0IF**: INT0 External Interrupt Flag bit

1 = The INT0 external interrupt occurred (must be cleared by software)

0 = The INT0 external interrupt did not occur

bit 0 **RBIF**: Port B Interrupt-On-Change (IOCx) Interrupt Flag bit(1)

1 = At least one of the IOC<3:0> (RB<7:4>) pins changed state (must be cleared by software)

0 = None of the IOC<3:0> (RB<7:4>) pins have changed state

The source code of the application is presented in the followings:

```
#include "Display_Utils.h"
```

```
unsigned short shifter, portd_index;           //Declaring variable for 7 segments display
```

```
unsigned int digit;                           //Declaring variable digit
```

```
unsigned short portd_array[4];                //Display array
```

```
    unsigned int cnt;                         // Transition counter variable
```

```
    unsigned int overflow;                    // Overflow counter variable
```

```
unsigned short mask(unsigned short num) {
```

```
    switch (num) {
```

```
        case 0 : return 0x3F;
```

```
        case 1 : return 0x06;
```

```

    case 2 : return 0x5B;
    case 3 : return 0x4F;
    case 4 : return 0x66;
    case 5 : return 0x6D;
    case 6 : return 0x7D;
    case 7 : return 0x07;
    case 8 : return 0x7F;
    case 9 : return 0x6F;
}
}

void interrupt() {                                // Interrupt function
    if (INT0IF_bit) {
        cnt++;                                    // Transition counter increment
        INT0IF_bit = 0;                          // Clear interrupt flag to enable next call
    }

    else if (TMR0IF_bit) {
        ovrfw++;                                  // Overflow counter increment
        TMR0IF_bit = 0;                          // Clear interrupt flag to enable next call on overflow
    }
}

void main() {
    ANSELA = 0;                                   // Configure port A as digital
    ANSEL_D = 0;                                  // Configure port D as digital
    TRISA = 0;                                    // Configure port A as output
    LATA = 0;                                     // Disabling port A
    TRISD = 0;                                    // Configure port D as output
    LATD = 0;                                    // Disabling port D
    TMR0L = 0;                                    // Disabling TMR0, least significant byte
    digit = 0;
    portd_index = 0;
    shifter = 1;
    GIE_bit = 1;
    TMR0IE_bit = 1;

```

```

ANSELB = 0x00;           // Configure RB0 pin as digital
TRISB = 0xFF;           // Configure RB0 pin as input
T0CON = 0xD8;           // Configure T0CON register, no prescaler
do {
    LATA = 0;           // Switch-off for 7 segment display
    LATD = portd_array[portd_index]; // Assign an adequate value for port D
    LATA = shifter;
    cnt = 0;
    ovrflw = 0;
    GIE_bit = 1;         // Enable general interrupt
    INT0IE_bit = 1;      // Enable external interrupt
    TMR0IE_bit = 1;      // Enable overflow interrupt
    while(ovrflw <= 7813);
    GIE_bit = 0;         // Disable general interrupt
    cnt = cnt % 10000;
    digit = cnt / 1000u; // extract thousands digit
    portd_array[3] = mask(digit); // and store it to PORTD array
    digit = (cnt / 100u) % 10u; // extract hundreds digit
    portd_array[2] = mask(digit); // and store it to PORTD array
    digit = (cnt / 10u) % 10u; // extract tens digit
    portd_array[1] = mask(digit); // and store it to PORTD array
    digit = cnt % 10u; // extract ones digit
    portd_array[0] = mask(digit); // and store it to PORTD array
} while(1); }

```

In Figure 2 the block scheme of the application is presented.

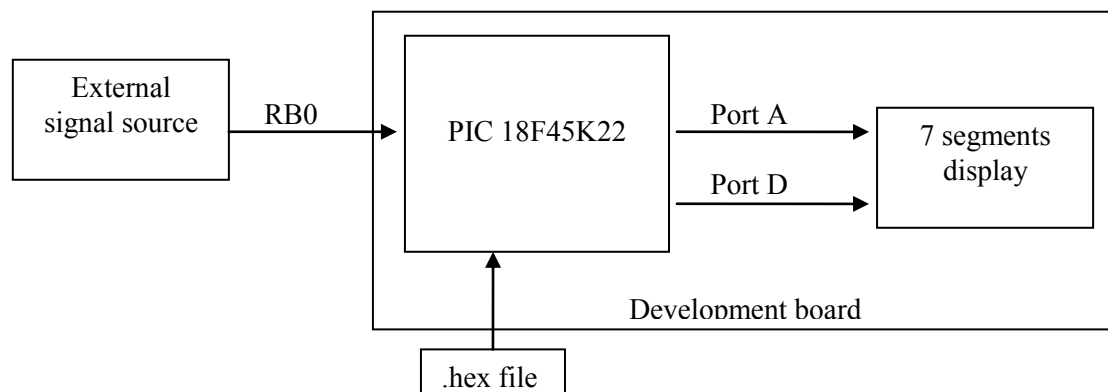


Figure 2. Block scheme of the application.

3. Conclusions

Using interrupts the following advantages can be achieved: the microcontroller can serve many devices, but not all in the same time. Each device can get the attention of the microcontroller based on the assigned priority [3], [4].

The program presented in this paper has a various applications from measuring the frequency of an input signal, to study its variation on a graphical display. The range of this frequency meter can increase with the increasing of the 7 segment digits.

References

- [1] Mitescu M and Susnea I 2005 *Microcontrollers in practice*, Springer, Berlin-Heidelberg-New York
- [2] ***<http://ww1.microchip.com/downloads/en/DeviceDoc/41412F.pdf>
- [3] ***<http://www.best-microcontroller-projects.com/hardware-interrupt.html>
- [4] ***<http://www.electronics-base.com/avr-tutorials/external-interrupts>
- [5] Popa M 2006 *Sisteme cu microcontrolere orientate pe aplicații*, Orizonturi Universitare, Timișoara, Romania
- [6] Calcutt D, Cowan F and Parchizadeh H 2004 *8051 Microcontrollers: An Applications Based Introduction*, Newnes, Amsterdam, Hollande
- [7] ***http://www.mikroe.com/pdf/mikroc/mikroc_manual.pdf
- [8] ***<http://www.mikroe.com/easypic/>