

Interrupt Lab and Multi-Use Peripheral Applications

EET 250

Objectives

- Exam source code and comment (all source C and is provided)
- Learn usage of interrupts by examining Interrupt based versus non-interrupts based code examples using PIC16F917 peripherals
 - Case Studies
 - A. Timer1 overflow interrupt and non-interrupt usage
 - B. ADC conversion interrupt and non-interrupt usage
 - C. RBO button Interrupt and non-interrupt usage
- Examining use of combining non-interrupt based peripheral operations in user applications
 - I. Variable delay of LEDs using ADC value as software delay setting
 - II. Variable PWM setting using ADC value to configure pulse duty cycle in fixed period output

Comments

- This is the final lab using 44 pin demo board
- Remaining labs will require and use of student assembled 40 pin DIP PIC16F917 proto-board and external hardware

Compiler Interrupt Support

- The compiler incorporates features allowing interrupts to be handled from C code. Interrupt functions are often called interrupt service routines (ISR).
- The function qualifier **interrupt** may be applied to any number of C function definitions to allow them to be called directly from the hardware interrupts.
- An example of an interrupt function for a midrange PIC processor is shown here.

```
int tick_count;
void interrupt tc_int(void)
{
if (TOIE && TOIF) {
TOIF=0;
++tick_count;
}}
```

In general

- It is up to the user to decide when to use interrupts.
- By deciding not to use interrupts the user may run the risk of missing response to critical outside events.
 - Ie. Air-bag, Anti-lock breaking, speed control
- PIC interrupts have guaranteed latency time for response
- Interrupt increase the response time of the processor to more “real-time” versus the alternative of “polling” peripherals.
- In interrupt the peripheral tells the processor when it needs service immediately

Mid-range PIC Specifics

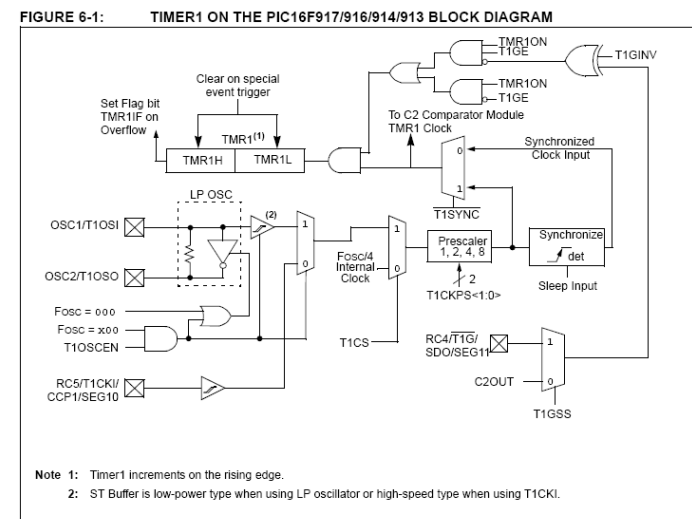
- C Code has one interrupt function that all peripherals that are interrupt enabled shared.
- Interrupt function must determine the source of interrupts (if using more then one peripheral)
 - It does this by examining all relevant interrupt flag bits XXIF to see which are set
- The interrupt function must clear the source of interrupt before exiting to re-enable that peripheral to get interrupt service
 - It does this by clearing the XXIF bit
- All processer “context saving” and “context restore” are performed automatically for W by interrupt function .

To Enable Interrupts and to make a peripheral interrupt enabled

- Must set global enable bit (GIE) in INTCON
- For peripherals referenced in INTCON
 - RB0, RB port change, Timer0 set the corresponding Interrupt enable bit and clear the corresponding XXIF bit.
 - Make sure interrupt function contains correct service for the enable peripheral
- For peripherals outside of those referenced in INTCON set bit 6 PEIE bit (general interrupt enable for other peripherals-- this is refer in book as EEIE (EEPROM enable) because PIC16F84 did not have extensive interrupt capability as PIC16F917)
 - Then go to specific interrupt enable either in PIE1 or PIE2 register
 - Interrupt function then must check pending request form PIR1 or PIR2 as needed.

Case A Timer1

- Timer 1 is a 16 bit timer
- Setting:
 - T1CON (bank 1)
 - Set up and prescaler
 - PIR1 (bank1)
 - Rollover flag for eight bits FF -> 00
 - 16 bit counter (Bank1)
 - TMR1L
 - TMR1H
- Interrupts
 - The Timer1 register pair (TMR1H:TMR1L) increments to FFFFh and rolls over to 0000h. When Timer1 rolls over the Timer1 Interrupt Flag bit (PIR1<0>) is set.
 - To enable the interrupt on rollover, you must set these bits:
 - Timer1 Interrupt Enable bit (PIE1<0>) PEIE bit (INTCON<6>) and GIE bit (INTCON<7>)
 - The interrupt is cleared by clearing the TMR1IF bit in the Interrupt Service Routine.



TIMER1-non interrupt

- Both interrupt version and non-interrupt version perform same functions.
- Open non-interrupt version TMR1.mcp
- Examine code.
- Build and download to PICKIT2
- Execute and note board LED
- This sets up timer without interrupts and then enters an While loop polling the TMR1IF bit
 - If set the PORTD LED0 is toggled.
- What are the timer1 settings?
- How often does Timer1 request service?
- Where does the TMR1IF get cleared?
- Is it cleared after every set condition?

```
/* cTMR1.c - TMR1 Demonstration

This program demonstrates the operation of the TMR1 Hardware to blink led

Tom Kibalo 10/10/07 EEt250

*/

__CONFIG(INT1C & WDTDIS & PWR1EN & MCLRDIS & UNPROTECT \
& UNPROTECT & BORDIS & IESODIS & FCMDIS);

#include <xc.h>
#include <tmr1.h>

void main()
{
    // Use TMR1 for a delay
    TRISD = 0x00;
    PORTD = 0x00;
    while(1) {
        T1CON = 0b00110001;           // TMR1 On/Internal Clock, 8x Prescaler
        TMR1H = 0x00;
        TMR1L = 0x00;
        TMR1IF = 0;
        // TMR1IE = 1;                // Turn off Pending Interrupt Requests
        // Enable TMR1 Overflow to Request Ints
        while (!TMR1IF);             // Wait for TMR1 to Overflow
        NOP();                        // Breakpoint Here - Time to Previous
        RDO = RDO ^ 1;               // Toggle RA5 to Turn ON/OFF LED
    }
}
// End cTMR1
```

TIMER1-Interrupt version

- Close non-interrupt version TMR1.mcp
- Open interrupt version Timer1Interrupt.mcp .Examine code.
- This sets up timer the same as before but with interrupt enabled.
 - PEIE, GIE, TMRG1IE are set during initialization—what is the significance of these bits?
 - Note the main routine enters a continuous loop--- do nothing loop while(1==1);
- Note the interrupt function. It uses keyword interrupt.
 - Here the TMR1IF is cleared and PORTD LED0 is toggled.
- What are the timer1 settings?
- Are the settings the same as the non-interrupt version?
- How often does Timer1 request service?
- Build code and select Simulator
- Use STOPWATCH with breakpoint set at TMR1IE=0 and another set in interrupt at TMR1IF =0.
- What is the measured time?
- Does it compare with TIMER1 configuration?
- Set debugger to PICKIT2 , download and execute.
 - The LED operation should be the same as the non-interrupt versions

```
__CONFIG(INTRC & WDTDIS & PWRTEN & MCLRDIS & UNPROTECT \
& UNPROTECT & BORDIS & IESODIS & FCMDIS);

int k = 0;

void interrupt tmrlint(void)
{
    // Respond to Timer 1 Interrupt

    TMR1IF = 0;           // Reset Timer Interrupt Request

    PORTD = PORTD+1;     // increment PORTD
} // tmrlint

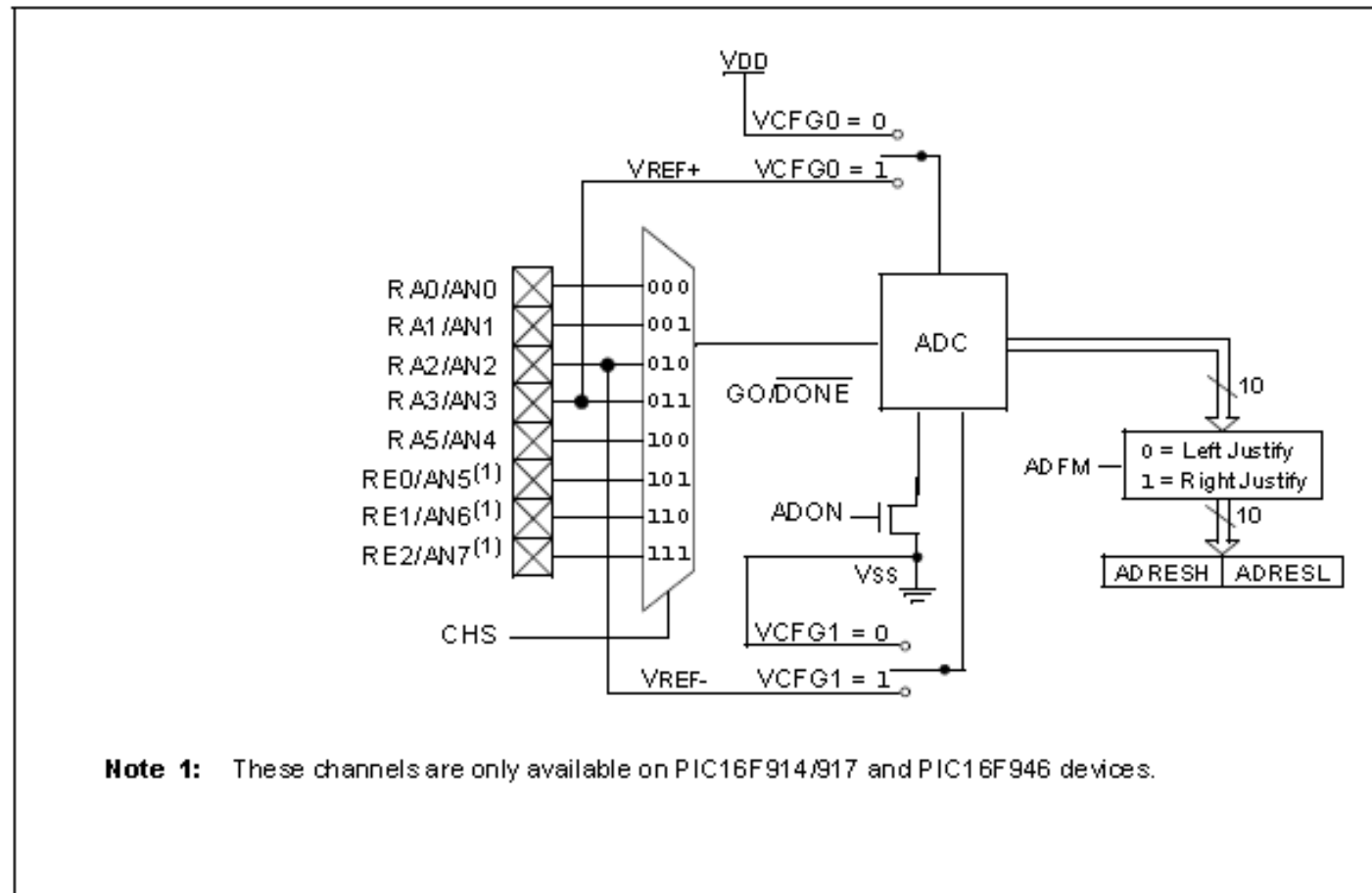
main()
{
    TRISD = 0;
    PORTD = 0;
    CMCON0 = 7;          // Turn off Comparators
    ANSEL = 0;           // Turn off ADC
    TMR1L = TMR1H = 0;   // Reset Timer 1
    T1CON = 0b00110101;  // Enable TMR1

    PEIE = 1;            // Enable Peripheral Interrupts
    GIE = 1;              // Enable Global Interrupts
    TMR1IE = 1;          // Enable Timer 1 Interrupts

    while(1 == 1);      // Loop Forever
} // End cPKLEDInt
```

Complete Block Diagram

FIGURE 12-1: ADC BLOCK DIAGRAM



ADC non-interrupt and interrupt comparisons

- Similar to previous TIMER1 comparisons
- Interrupt and non-interrupt version operation—both display contents of ADRESH to PORTD using POT as input to ADC.
- Open ADC.mcp and examine code. Build and download to PICKIT2 and test operation
- What are the ADC settings?
- Is there a continuous loop to check ADIF ?
- Close ADC.mcp and open ADCInterrupt.mcp
- Examine code.
- Are the ADC conversion settings the same?
- What bits are used to enable interrupts?
- What does the interrupt service routine do?
- Build code and download to PICKIT2
- Execute. Exercise pot changes. Does this application perform in a similar way to ADC.mcp?

Interrupts exercises so far

- Not very exciting
- Just replacing an polled non-interrupt version with an interrupt version with a “do-Nothing” continuous loop
- Interrupt really afford the user with the ability to almost do several things at one
- The next example InterruptButton.mcp will illustrate this

Two Simultaneous operations

- InterruptButton.mcp uses the RB0 interrupt to toggle RD7 LED while main loop is toggling RD0 led without interfering with each other
- How is RB0 interrupt enabled?
 - Describes bits used.
- What happens during RB0 interrupt service?
- Describe the loop being executed during main.

```
void interrupt rb0int(void)
{
    // Respond to Timer 1 Interrupt
    //wait 20 msec
    j=0;
    while(j<TwoHundredms)
    { j++; }
    INTF = 0; // Reset RB0 Interrupt Request

    RD7 = RD7 ^ 1; // Toggle RD7 to Turn ON/OFF LED under interrupt
} // rb0int

main()
{
    RD0 =0;
    RD7 =0;
    TRISB0 =1; //rb0 is an input

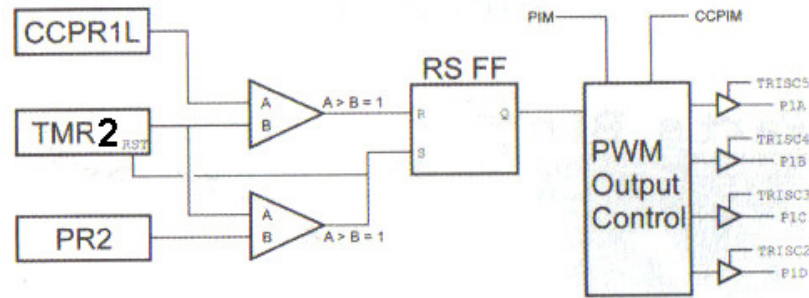
    PORTD =0x00;
    TRISD7 = 0; // MSB LED
    TRISD0 =0; // LSB LED
    GIE = 1; // Enable Global Interrupts
    INTE = 1; // Enable RB0 Interrupt

    //main loop just delay 600 ms and toggle LED 0
    while (1)
    {
        i =0;
        while(i<TwoHundredms)
        { i++; }
        RD0 = RD0 ^ 1; // Toggle RD0 to Turn ON/OFF LED
    }
} // End cDebounce
```

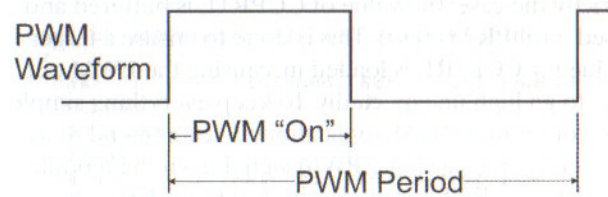
Dual Use ADC and Timer Delay

- Non-interrupt variable rate (using pot) for rotating LED
 - VrateLED.mcp –open code, examine, build, download to PICKIT2 and demo by varying pot rotation
- This code uses the output of ADC as a setting to control the outer loop (of a loop within a loop) software delay.
- How is the ADC conversion done—what C Construct is used to supervise the conversion?
- What is the control variable used by that construct?
- List the states of that variable and what happens under each state
- What variable is set using ADRESH?
- How does the overall delay change based upon the ADRESH setting?

CCP1- PWM Application Diagram



PWM block diagram



$$\text{Duty Cycle} = \frac{\text{PWM On}}{\text{PWM Period}}$$

PWM

Dual Use of PWM and ADC pot

- Similar to previous example
 - Open PWM.mcp examine, build, download to PICKIT2—execute
 - There is nothing to see –must get a lab scope and hook-up to pin RC5
 - Vary pot and note that pulse width –not period is changing
 - Mechanics
 - ADC ADRESH is used to set the CCPR1L register of the PWM of the CCP1 peripheral
 - PR2 (period register is fixed)-always same setting
 - PWM is the result of CCPR1L setting (pulse width) over the period setting (PR2)
 - Pin RC5 is configured for output
 - Note that ADC using same state engine as earlier code
 - Examine CCP1 set-up—explain SFR settings
 - Examine code for TIMER2 setting –what is the TMRIF flag period?

Exercise

- Set up WATCH for CCPR1L , PR2 and TMR2
- Using scope on RC5 and running code on PICKIT2
- Adjust pot for low PWM (<10%)
 - Measure on scope
 - Stop debug and examine WATCH. Does PWM match WATCH settings-shown calculation
- Repeat for PWM of approx 50% and approx 75%

Practical Application-LED Dimmer

- We can use the code as an LED light dimmer
- Hook a 1K resistor to RC5 in series with the anode of an LED.
- Connect cathode of LED to ground
- Run application –vary pot to illustrate LED dimming with pot adjustment
- Demo to instructor.