**DUE: At beginning of lab 6 session**

Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Section: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Names: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ and \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |
| --- | --- |
| LABORATORY CHECK OFFS | |
| **Section 1 Configuring a simple interrupt to count the number of switch presses**  INT0 Counts up. Observe occasional repeated print of same pulseCount value. Give example showing where the SAME value of pulseCount is printed twice.  Switch was pressed \_\_\_\_\_\_\_ times.  Switch was pressed \_\_\_\_\_\_\_ times.  Switch was pressed \_\_\_\_\_\_\_ times.  Switch was pressed \_\_\_\_\_\_\_ times.  After cli(); added, do you ever observe the SAME value of pulse count being printed?  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  What defect in the counting still remains?  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  **Section 2 Using a second interrupt to count down**  Counter counts up and down, but with glitches  **Section 3 Using the pin change interrupts to allow other pins to trigger interrupts.**  PCINT1 triggered by both PC0 and PC5 using volatile Boolean flag.  Any pending interrupt flags are cleared before enabling interrupt  **Section 4 Counting up and down using PCINT1**  Count up, count down using pin change interrupt only and changeMap code method.  **Section 5 Using a Timer to Call an Interrupt Service Routine.**  Timer 1 configured properly and pulseCount increments once per second.  “TIM1\_COMPA\_vect” results observed | \_\_\_\_  \_\_\_\_  \_\_\_\_  \_\_\_\_  \_\_\_\_  \_\_\_\_ |
| Points | \_\_\_\_ |

Prelab:

Read the AVR Atmel 328 microcontroller datasheet sections on Interrupts (chapter 12), External Interrupts (chapter 13) and I/O ports (chapter 14). pages 57-93. The pages numbers refer to Revision G of the ATMEGA datasheet as posted on mycourses. You may also find this page helpful once you start the lab: <http://gammon.com.au/interrupts>

Learning Outcomes:

By the end of this lab you will be able to:

1. Configure registers to setup an interrupt service routine (ISR) which will jump to a new code section at any time, immediately, based on an event e.g. input pin level changing or timer value being reached.
2. Explain why the volatile declaration should be used by variables modified by an interrupt service routine and used outside the interrupt service routine.
3. Configure the registers to setup a pin change interrupt (PCINT0, PCINT1, or PCINT2)
4. Use the PINC command to read the state of each pin (HIGH or LOW) on an entire port regardless of whether the pin is an input or an output.
5. Create a change map to show which pins on a port have changed.
6. Use an exclusive OR function (“^”) at a bitwise level.
7. Declare and use static variables to keep a variable’s state (its previous value) after a function or ISR has been called.
8. Decode which pin changed on a PORT using the exclusive OR function to create a pin change map.
9. Use a timer to create an interrupt at a regular time interval and call a short piece of code (an ISR).

Overview:

In the last lab, hardware timers were introduced as a way allow a microcontroller to perform certain tasks without requiring the attention of the main software loop. For example, the hardware timer could blink the LED on and off directly without having to run code in the main software loop or the timer could generate pulsewidth modulated signals at a controlled frequency. In both cases, the hardware timer was only able to control the state of a digital pin (e.g. set the pin high or low).

It is possible to use the hardware timer to not only control the state of pin but to also run a short piece of code immediately when the timer changes state. The timer is configured so that it interrupts the main software loop, jumps to a new section of code, runs it, and then quickly returns to running the code in the main software loop. This interruption of the mainline code is referred to as an “*interrupt*”. Often, an interrupt is a hardware signal (usually external to the µC) that causes the mainline code of the microprocessor (inside the microcontroller) to stop and execute the interrupt code then resume running the mainline code.

Interrupts allow the microcontroller to handle multiple tasks, apparently at the same time. For example, if the code in the interrupt executes in less than 20 microseconds, then it may be unnoticeable in terms of its impact on the speed of execution of the main software loop.

The technique of interrupting the code to run another piece of code is call **interrupt handling**. The phrase **Interrupt Service Routine** or **ISR** refers to the short piece of code that is executed when an interrupt event occurs. Very often the code in the interrupt routine has to be run urgently and without delay, that is why it is given a higher priority than the main software loop.

**What can trigger an interrupt routine to run?**

In the Atmel 328 microcontroller, the jump to the interrupt routine can be triggered by a timer event, such as a timer compare match or timer overflow, or the interrupt can be triggered by a change in the state of an external pin e.g. a digital signal on a dedicated hardware pin INT0 (Arduino d2) or INT1 (Arduino d3), or a change in the state of any pin by using a port change interrupt, e.g PCINT0 (many different pins). Port change interrupts are a little more complex to deal with because you have to write code to determine which of the pins on the port has changed.

**What happens to the code in the main loop while the interrupt routine runs?**

When the interrupt event occurs, the processor resources such as the stack and ALU must be freed up to run the code in the interrupt routine. Variables from the main loop are set aside in temporary memory locations while the processor runs the interrupt service routine. When the interrupt routine is done, the main code continues from where it left off.

**What if the interrupt routine changed one of the variables that the main code is using?** How will the processor know that its value has changed and needs to be refreshed in the temporary registers? The solution comes by telling the compiler that certain variables are volatile and may be changed by an interrupt service routine. Any variable that is modified in an interrupt service routine and later used outside the interrupt routine must be declared as **volatile**. For example : volatile int pulseCount = 0;

When the compiler sees the keyword volatile, it knows to force the code to go back into the main memory and refresh the value of the variable, and to not rely on the temporary copy that was set aside when the ISR occurred. Volatile variables are a bit slower to use, but absolutely essential to avoiding trouble. If you forget to declare a variable volatile, the program’s operation can appear erratic and it can be very hard to debug what is going on.

In later labs, you will be running a PWM timer, responding to external inputs and running an overall state machine. For that system, you will need to use interrupt routines to allow the microcontroller to handle all of these tasks at the same time.

How to Succeed With This Lab:

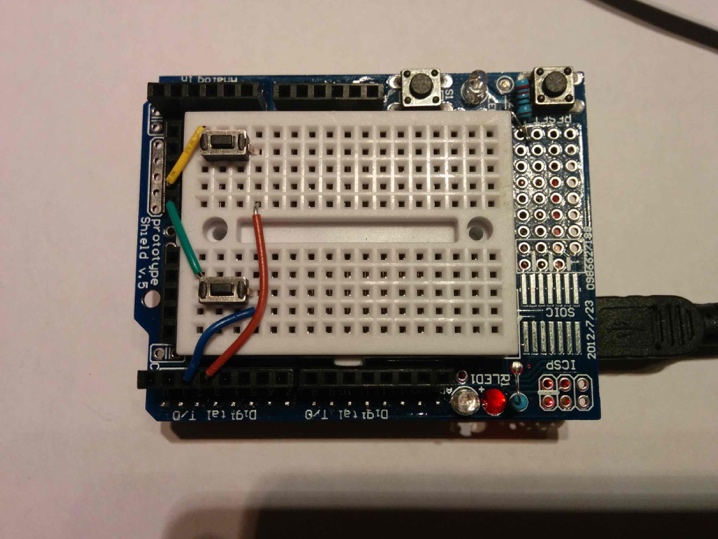
Check the register functions, names, and settings by reading the datasheet for the Atmel 328 microcontroller.

Troubleshooting skills used:

1) Using a boolean flag to indicate when an interrupt service routine has been called.

**SECTION 1 – Configuring a simple interrupt to count the number of switch presses**

**Procedure:**

1. Review the code Lab5\_ISR\_INT0, in the text box below.
2. Examine the code and determine where you need to connect a switch ground to generate the switch input. Build the switch circuit on your protoshield as shown below (note: both the switches for this section and the next section are shown in the figure)
3. 
4. Go through the code and learn its operation. Specifically, understand:
   1. How registers are used to configure the interrupts.
   2. The use of the function cli().
   3. Individual interrupts and how they are enabled.
   4. Use of sei().
   5. Note that the ISR routine looks somewhat like a function call, but that the syntax is slightly different i.e. the name of the routine has to exactly match the interrupt vector that is triggering the ISR e.g. ISR(INT0\_vect) {} and the interrupt receives no parameters, and returns no values.

// Lab5\_ISR\_INT0

// Written by: Nov 01, 2014, JTSchueckler

// revised: Mar 05, 2017, Clark Hochgraf

// Desc: ++ PLEASE READ ALL COMMENTS ++

// This program demonstrates a simple use of the INTO functionality.

// Review the initialization of the interrupt system.

// Review the setup of the ISR.

// Define volatile global variables for ISR system interface.

volatile long pulseCount = 0, prevpulseCount = -1;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void setup() {

Serial.begin(9600); Serial.println("Lab 5 ISR INT0 counter");

configurePins();

// Display the bootup values of EICRA, EIFR and EIMSK

Serial.print("EIFR \t"); printlnBinaryByte(EIFR);

Serial.print("EICRA \t"); printlnBinaryByte(EICRA);

Serial.print("EIMSK \t"); printlnBinaryByte(EIMSK);

Serial.println();

initInterrupts();

// Display the programmed values of EICRA, EIFR and EIMSK

Serial.print("EIFR \t"); printlnBinaryByte(EIFR);

Serial.print("EICRA \t"); printlnBinaryByte(EICRA);

Serial.print("EIMSK \t"); printlnBinaryByte(EIMSK);

Serial.println();

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void loop() {

if (pulseCount != prevpulseCount) { // only print the pulse count if it has changed

prevpulseCount = pulseCount;

Serial.print("Switch was pressed ");

Serial.print(pulseCount);

Serial.println(" times.");

}

} // Arduino loop()

//===================================================================================

void configurePins(void) {

pinMode(2,INPUT\_PULLUP); //Set up PD2 (INT0) as an INPUT w/Pullup;

}

//===================================================================================

void initInterrupts(void) {

/\* disable interrupts using cli(); and enable interrupts using sei(); with Arduino IDE

Typical Atmel C code would use \_\_disable\_interrupt();

to clear the global interrupt bit in the SREG.

and \_\_enable\_interrupt(); to set the global interrupt enable in the SREG.

OR by direct manipulation of the "I" bit in the SREG.

NOTE: The Serial Monitor and other Arduino functions will be directly affected

if you try to manipulate the SREG in the Arduino operation. \*/

cli(); // Clear the Global INT bit to disable ALL interrrupts

EICRA |=(1<<1); // Configure EICRA to detect falling edge for INT0

// ISC0 = 0b10 refers to Interrupt Sense Control for INT0

// the two bits {1 and 0} program the hardware to respond to a falling edge

// at the input pin INT0 (PD2)

EIMSK |= 0x01; // Unmask INT0 in EIMSK so that it is enabled

EIFR |= 0x01; // Clear any pending interrupt flags for INTO by writing a 1 to bit 0

sei(); // Sets the Global INT bit to enable ALL interrrupts

}

//===================================================================================

void printlnBinaryByte(byte value) { // prints 8-bit data in binary with leading 0's

Serial.print("B");

for (byte bitmask = 0x80; bitmask; bitmask >>= 1) {

Serial.print((bitmask & value) ? '1' : '0');

}

Serial.println();

}

//===================================================================================

ISR(INT0\_vect){ // the ISR name must match the interrupt (e.g. INT0\_vect)

pulseCount++;

// NOTE: NO Extraneous instructions in the ISR

}

1. Create a sketch and load the code. Press the switch a few times and observe the value of pulseCount. Does the counter value accurately count each switch press?
2. Press the switch (50 times) and look carefully for unexpected errors in what is printed for the pulseCount. Specifically, what error do you see, what is the pattern? In this case, we are not just interested in the fact that pressing the switch may result in multiple count value changes. There is another error. Record this error on your signoff sheet.

Refer to the datasheet for information about the registers for interrupts.

1. Modify the main loop code as shown in the textbox below by disabling interrupts before checking the value of pulseCount and reenabling interrupts after checking pulseCount.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void loop() {

**cli();**

if (pulseCount != prevpulseCount) { // only print the pulse count if it has changed

prevpulseCount = pulseCount;

Serial.print("Switch was pressed ");

Serial.print(pulseCount);

Serial.println(" times.");

}

**sei();**

// other code that you want to run in the main loop

} // Arduino loop()

1. Load and run the code. Examine the behavior of the pulseCount printing as you press the switch 50 times. How has the behavior changed? What defect in the counting still remains?
2. Complete the coversheet and **obtain a signoff.**

**Explanation of the strange pulseCount values:**

You might be wondering why the pulseCount values printed are not as expected. The reason is related to the behavior of interrupts. An interrupt can occur at any time, even when the code is only half way through running an instruction, such as checking the condition inside the if statement. It is possible for the pulseCount to change from when it is checked in the condition of the if statement (if (pulseCount != prevpulseCount) ) to when it is printed inside the if statement (Serial.print(pulseCount);); Even worse, it is possible that the pulseCount interrupt can occur halfway through the value of pulseCount being read. Keep in mind that the microcontroller is an 8-bit microcontroller with 8 bit instructions. PulseCount is an integer (16 bits) that takes two operations to read into the stack. It is possible that the interrupt occurs right in the middle of getting pulseCount. These types of problems are very hard to diagnose.

The solution is to use **Atomic operations** that can not be interrupted. In this case, we disabled interrupts (cli();) before accessing and working with pulseCount and then re-enabled interrupts as soon as we were done with pulseCount. Atomic operations must be kept very short/quick or interrupts will be missed.

**SECTION 2 – Using a second interrupt to count down**

The code from section 1 only counts up when the interrupt is triggered. In this section, you will modify the code so that the pulse counter is decremented when an interrupt occurs on the pin corresponding to INT1.

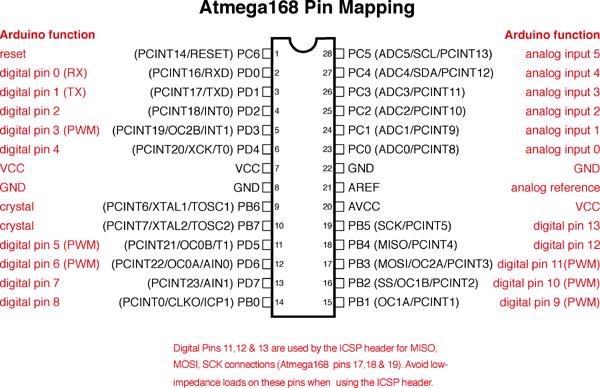
**Procedure:**

1. Modify the code used in SECTION 1 to respond to a second interrupt. Use the INT1 interrupt. This ISR will decrement pulseCounter when INT1 is triggered. Connect a second pushbutton switch to the appropriate pin and configure the pin in software.
2. The pulseCount should only be printed when the value of it changes.
3. Test your pulseCount up and pulseCount down operation. Demonstrate the system to your instructor? Is the operation error free?
4. **Obtain a signoff.**

HINT: You will have to fully setup and enage the new interrupt and will have to add code for the new ISR. You will have a total of two separate ISR routines.

**SECTION 3 – Using the pin change interrupts to allow other pins to trigger interrupts.**

In the Atmel 328 microcontroller, there are only two dedicated external interrupts (INT0, INT1) tied directly to pins (INT0 to pin 2, INT1 to pin 3). However, it is possible to generate interrupts from other pins, on a port wide basis (a port consists of a group of 8 pins). There are only of three of these port wide interrupt vectors (PCINT0, PCINT1, PCINT2) {See Table 12-6 Pg 65}. Each of these interrupts is triggered by a change in the state of a pin on one of the pins in a port. For example, the interrupt service routine for PCINT2 can be triggered by a change in state on pins from 16 to 23. In the pin diagram, the pin named PCINT16 is also labeled as PD0 (port D0) while the pin named PCINT23 is also labeled as PD7 (port D7). See Below:



Pin change interrupts are triggered when any unmasked pin on the port changes state. The PCINT’s are different from INT0 or INT1 in that PCINTs do not tell you which of the pins caused the interrupt. However, you can figure that out by keeping track of the previous state of each pin value (was it previously high or low when the interrupt was last called). To read the value of a pin, without changing the pin mode, you can read the value from a register called PINx. PIN standard for PORT INPUT. The register PINx gives you an 8 bit representation of the state of whole 8-bit port. So, you are reading in all the pins on a port at once, rather than reading one pin at a time.

The PCINT interrupts are configured in a similar way to INT0. Refer to the datasheet for all the relevant register settings. Look up registers PCICR and PCMSK0. (Pin Change Interrupt Control Register, Pin Change MaSK register)

**Overview:**

In the following procedure, you will modify your code to so that when pin PC0 is switched LOW, the main loop is interrupted and when PC5 is switched LOW, the main loop is interrupted.

**Procedure:**

1. Disable and remove the INT0 and INT1 interrupts and ISR code as these pins will not be used.
2. Replace these interrupts by a single pin change interrupt service routine, ‘PCINT1\_vect’ that covers PCINT pins 14 through 8 [14:8] (PORT C).
3. See the Atmel datasheet section 13.2 that describes the pin change mask PCMSK1. The naming is somewhat confusing because the interrupt name is PCINT1, but the pin names also include pins names like PCINT8, PCINT9, PCINT10, etc.
4. Setting bit 0 to HIGH(1) in PCMSK1 would enable pin PCINT8 to trigger the interrupt service routine PCINT1. Note that the pin named PCINT8 is also named adc0 or A0 or PC0 (PORT C0). Setting bit 5 to HIGH (1) would allow the pin named PCINT13 to trigger the interrupt service routine as well as pin PCINT8.
5. Configure the register PCICR to enable the interrupt.
6. **Reconfigure the hardware so that the switches SW1 and SW2 are connected to the correct pins for PC0 and PC5. You will, of course, have to set the pins as inputs with internal pullup resistors.**
7. Test that your interrupt service routine is called when either the PC0 or PC5 pin is switched low by printing the message “PCINT was triggered” inside the ISR.

Use this code for your ISR, while you are building and testing it:

{ delay(100);

// THIS IS FOR TESTING ONLY – Remove from final code -

// This is slow code in an ISR – a very bad idea

Serial.println("PCINT1 was triggered");

}

1. Normally you would not put a print statement or a delay() inside the ISR, because these statements are very slow to execute which could interfere with the operation of other interrupts. In fact, your code may have crashed because of this. A much better method is to set a Boolean flag in the ISR and then print in the main loop:

Modify your ISR code to be the following

{

PCINT1\_was\_triggered=true;

}

Declare the flag as a volatile Boolean

volatile boolean PCINT1\_was\_triggered=false;

In the main loop, put the following code to read the flag and print only if the ISR was triggered:

if(PCINT1\_was\_triggered==true) {

Serial.println("PCINT1 was triggered");

PCINT1\_was\_triggered=false;

};

1. Test your code and notice that the print statement is triggered when either the PC0 or PC5 pin goes HIGH (rising) or goes LOW (falling). Using SW1 or SW2 will generate two print statements because; A) there is a pin change when pressing either SWn as (PC0/5) goes from HIGH to LOW and B) there is a pin change when releasing either SWx as (PC0/5) goes from LOW to HIGH. This is because the interrupt is triggered based on the change in state of a pin NOT its steady state value.
2. Demonstrate your code to your instructor and **obtain a signoff.**

**SECTION 4 – Counting up and down using PCINT1**

Now that you have verified that the PCINT1 ISR can be triggered by change in pin state on PC0 or PC5, the next step is to figure which pin (PC0 or PC5) caused the interrupt. If it was pin PC0, increment pulseCount, otherwise decrement the pulseCounter.

Because this code will be in an interrupt service routine, it is important that it runs as quickly is possible, so you should avoid using digitalWrite(), or digitalRead() and instead use the faster PORT commands with direct bit manipulation.

A good way to read the status (HIGH or LOW) of several digital pins at once is to read the status of PORTC using the PINC register. PINC is a byte that provides a reading of the input pins of port C as shown in the register description below.



If you want to read the status of PC0 (port C, bit 0) then you can take PINC and mask it to show just bit 0.

PC0\_status=PINC & 0x01

To read the status of Arduino pin A5 (port C, bit 5) then you can take PINC and mask it to show just bit 5.

PC5\_status=PINC & 0x20

Note that using PINC, you can read whether the pin is HIGH or LOW, without changing the direction of the pin (output or input). So, you can read the state of an output pin, without making the pin into an input. (e.g. not having to use DDR registers).

**ChangeMaps**

Rather than decoding each bit of PINC, another approach is to create a change map that shows which pins have changed since the last time PINC was read.

First, the ISR needs to keep track of what the previous byte value of PINC was. For this, a static byte is declared. **The keyword static means that the value of prevPINC is not forgotten when the ISR routine is left.** When the ISR is called again, the past value of prevPINC is remembered.

static byte prevPINC;

byte newPINC;

byte changeMap;

Notice that newPINC and changeMap don’t have to be static variables.

Next, the current value of PINC is read and stored as a byte.

newPINC = PINC;

Then the newPINC is compared to prevPINC to see which bits have changed. This comparison for change can be done using an exclusive OR function (“^”) at a bitwise level to produce a new variable called changeMap.

changeMap = newPINC ^ prevPINC;

changeMap indicates which bits changed since the last time the ISR was called. Based on the changeMap, you can decide if the ISR was called due to a change in PC0 or PC5 Of course this can be generalized to even more pins, as each pin maps to a specific bit.

If pin PC0 cause the interrupt, then masking change map with 0b0000 0001 will result in a non-zero value, and the pulseCount should be incremented.

if (changeMap & 0x01) pulseCount++;

If pin PC5 cause the interrupt, then masking change map with 0b0010 0000 will result in a non-zero value, and the pulseCount should be decremented.

if (changeMap & 0x20) pulseCount--;

Using these methods, a single interrupt servive routine (e.g. PCINT1) can perform many functions.

When writing the ISR, remember that you need to assign a value to prevPINC after calculating the changeMap.

**Procedure:**

1. Modify your code to use the changeMap method to increment or decrement pulseCount.
2. Demonstrate and explain the operation of your code to your instructor. **Obtain a signoff.**

**SECTION 5 – Using a Timer to Call an Interrupt Service Routine.**

So far in this lab, hardware pins have been used to call interrupts. It is also possible to call interrupts from software or from timers. In this section, you will write code to increment pulseCount when Timer1 has a Compare Match.

**Procedure:**

1. Create a new sketch to demonstrate the use of Timer1 interrupts.
2. In setup section, make your code print the following: ”Timer1 program was started”
3. Setup Timer 1 to run in mode 15, fast PWM with OCR1A as TOP. Set the clock prescale to 1024 and choose OCR1A so that the Timer period is 1 second.
4. Configure COM1A and COM1B for normal port operation, i.e. neither pin PB1 or PB2 is affected by timer compare match. This leaves the pins OC1A and OC1B free for other purposes. Name your ISR “TIMER1\_COMPA\_vect”
5. Run your code and verify that the pulse counter increments and prints out once per second.
6. Demonstrate this to your instructor for a signoff.
7. Now change the name of your ISR to “TIM1\_COMPA\_vect” and run the code again. What happens? Demonstrate this and **obtain a signoff.**

**Troubleshooting Tip:** If an interrupt occurs and there is no named section of code for the program to jump to (i.e. the ISR name doesn’t exist or is spelled wrong) then the code still jumps to a certain location in memory but what happens next is undefined and can be very bad. This why earlier, you started by printing “PCINT was triggered” before writing the rest of the interrupt, this was to make sure that the interrupt was at least jumping to the correct section of code. This sort of incremental build/test approach is a very helpful skill to develop.

Hints for writing ISRs (adapted from Nick Gammon)

In brief, keep them short! While an ISR is executing other interrupts cannot be processed. So you could easily miss button presses, or incoming serial communications, if you try to do too much. In particular, you should not try to do debugging "prints" inside an ISR. The time taken to do those is likely to cause more problems than they solve.  
  
A reasonable thing to do is set a single-byte flag, and then test that flag in the main loop function. Remember, inside an ISR interrupts are disabled. Thus hoping that the time returned by millis() function calls will change, will lead to disappointment. It is valid to **obtain** the time that way, just be aware that the timer is not incrementing. And if you spend too long in the ISR then the timer may miss an overflow event, leading to the time returned by millis() becoming incorrect. Recording (saving) the current timer value is a reasonable thing to do in an ISR.

**Warning:** Since interrupts are disabled inside an ISR, and since the latest version of the Arduino IDE uses interrupts for Serial reading and writing, and also for incrementing the counter used by "millis" and "delay" you should not attempt to use those functions inside an ISR. To put it another way:

* Don't attempt to delay, eg: delay (100);
* You can get the time from a call to millis(), however it won't increment, so don't attempt to delay by waiting for it to increase.
* Don't do serial prints (eg. Serial.println ("ISR entered"); )
* Don't try to do serial reading.