**Laboratory 5: Interrupts**

**Introduction and objectives:**

This assignment deals with the topic of interrupts. Often a CPU needs to pause the sequential execution of a running program to respond to some urgent external signal it has received. Most CPU’s have a mechanism for interrupting program execution using dedicated signal lines called ‘interrupts’. On receiving an interrupt, the current state of the system is temporarily stored somewhere, execution jumps to a special predefined location in memory where the interrupt service routine (ISR) resides. After successfully completing the ISR, the previous program state is restored and normal execution resumes.

Interrupts can be triggered by some event within the microcontroller system, like a countdown timer or they can be sent in as an electronic signal on one of the I/O lines.

In this assignment you will learn how to write interrupt service routines to handle both types of interrupts and setup hardware to test interrupt generation.

**Procedure:**

Solve the question asked in Part A first independently and get your answer sheet examined before beginning your experimental work.

Part B consists of two related exercises that you work on with your group partners on the Arduino board. The solution to the first exercise is needed for the second, so you must do them in order.

 /7

(Pt. C

Freq. + Phase meter

 /3

(Pt. A)

Question

/15

 /5

 (Pt. B)

 Freq. Meter

**Part A: Solve this exercise first independently, and get it examined.**

Examine and debug the following program with an interrupt service routine that a novice programmer wrote.

*(If you have forgotten what we talked about in class, or missed the lecture please refer to the quick reference sheet on interrupts at the end of this assignment)*

// This program counts the number of pulses coming in on digital pin #2

#include <avr/interrupt.h>

int inputpin = 2; // expect pulses to arrive on pin 2

volatile int count = 0; // counts the number of pulses

void setup()

{

 Serial.begin(9600);

 pinMode(inputpin,INPUT);

 attachInterrupt(0, pulse, HIGH); // Digital pin 2 corresponds to Interrupt 0
 // Call ISR function ‘pulse’ every time

 // pin 2 goes HIGH
}

void loop()

{

 count=count-1;

 delay(100); // and do something else here…

 if(count>100)

 {

 Serial.println(“Congratulations! Count>100”);

 }

}

void pulse()

{

 count++; // increment count every time a rising edge is detected on pin 2

}

**Questions:**

With a fast series of pulses arriving on pin 2, will the count ever *reliably* get up to 100?

YES\_\_\_\_\_\_\_
NO\_\_\_\_\_\_\_\_
MAYBE\_\_\_\_
Describe the problem:
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Suggest a solution or guideline for our novice programmer

**Part B: Frequency counter or speed meter for a motor**Your objective is to measure the average speed of a motor. Assume that a sensor mounted on the motor is putting out a more or less continuous series of TTL pulses i.e. these pulses go from a low of 0V to high of 5V so they can be fed straight into the digital inputs of the Arduino.

Use interrupts to measure the average frequency of these pulses.

Use the logic pulse generator provided to input 1 Hz/1kHz pulses and check the measured frequency. You can then also use the function generator on your work-bench to measure other arbitrary frequencies.

Here are a few tips:

1. The Arduino has an internal timer TimerOne that can be used in the mode of a ‘stopwatch’. You load the timer with a number, it counts down per clock cycle and when the countdown is complete it overflows and generates an interrupt. The TimerOne function library is available on your desktop to make these timers easy to use.
2. Note that it is perfectly fine to have a program with two (or more) interrupts – you need to define the ISR’s associated with each interrupt. When an interrupt occurs, the ISR attached to it is run.
3. *The most important rule on writing code for an ISR is that it must be as small as possible (for example perform simple math on some variable). Any operation inside an ISR that costs a lot of time can lead to a ‘race condition’ – this happens when the ISR takes too long to execute and interrupts keep piling up one behind another so that the main program loop() is never executed. With this in mind,* ***never ever*** *use Serial.println(…) type function calls from within an ISR*

Novice programmer track time in a program using *micros()* and *millis()* which give you the current time since the Arduino started. The counter behind the function *micros()* will overflow after every 106 counts – so if two events occur at δt= 1sec+1μs, putting timestamps on the two events with *micros()* is likely to give wrong δt= 1μs !
It is much simpler and reliable to *independently* keep track of time using the TimerOne timer (which acts as a count-down clock) serviced by one ISR. Use a second ISR to count the number of pulses arriving on a digital pin.

**Draw your program algorithm/flowchart here:**

 **/2 (algorithm) + /3 (Demo)**

**Part C) Double speed meter that also measures phase.**

Assume you have two motors running at different speeds like the one above, or equivalently one motor driving two gears at different speeds.

Each gear produces a digital pulse train whose frequency corresponds to their speeds. Normally if the gears are perfectly meshed, the phase difference between these two pulse trains should be constant – phase difference here is measured as the time difference *Δt* between two successive positive edges. Write a program and set up the associated hardware connections to continuously monitor this phase difference.

You need to:

1. Measure the average speed of each pulse train.
2. Measure the phase difference between them

(phase is simply defined here as the *Δt* between two successive positive edges – even if the frequencies of the two pulse trains are different)

To test your program, you can use an RC filter to add phase to a square wave (1 Hz or 1 kHz) generated by the logic pulse display unit – one signal goes straight into one digital pin on the Arduino, and the other, delayed signal goes into a second digital pin. Make sure the RC filter’s signal attenuation does not reduce the signal level too much – the logic 1 signal must remain at V>3V.
The program should verify the pre-calculated RC time constant of your filter.

**Draw your algorithm flowchart here:**

 **/3 (algorithm) + /4 (Demo)**

**Some notes on coding for interrupts**

**(1)**

The simplest function to setup an Interrupt Service Routine (ISR) is *attachInterrupt(…)*

Here’s a self-explanatory example of its usage:

#include <avr/interrupt.h>// type definitions of all interrupt

 // related things, must have this

volatile int GlobalSharedVariable1 = 0; // variables shared between loop() and the ISR()

volatile int GlobalSharedVariable2 = 0; // must be declared as volatile

void setup()

{

attachInterrupt(0, myISR1, CHANGE); // call function myISR1()

 // when digital pin 2 changes

attachInterrupt(1, myISR2, CHANGE); // call function myISR2()

 // when digital pin 3 changes

}

void loop()

{

 // You can ‘peek’ at the values of GlobalSharedVariable1 & 2

 // eg. print them. But it’s not a good idea to change their values here

}

void myISR1()

{

 GlobalSharedVariable1++;

}

void myISR2()

{

 GlobalSharedVariable2++;

}

**(2)**

The TimerOne library has been placed on your desktop with many functions that let you manipulate Timer1. Look at the documentation files in the folder, and the source code of the library functions to understand how it works and how to use it most efficiently.

It should already be set up to be used readily from the Arduino IDE. If not, make sure the folder ‘TimerOne’ containing TimerOne.cpp and TimerOne.h has been copied to ${HOME}/SystemSetup/arduino-00XX/libraries where all the other libraries reside. arduino-00XX is the version of the Arduino IDE installed on your desktop (XX=18 or 21)

Look in the documentation for details on syntax of the TimerOne.Initialize(…) and other functions provided by the library.