EE 2361 - Introduction to Microcontrollers

Laboratory #4

Input Capture and Output Compare Peripherals

Table of Contents

Background

Input capture (IC) and output compare (OC) are two common peripherals in microcontrollers. The input capture peripheral is capable of automatically detecting and storing the time of a change in state of an input pin. The output compare peripheral can automatically switch an output at a specific time in the future. Together, they allow for automated initialization and recording of I/O events. When certain external devices are utilized this can greatly reduce the instructions cycles required of the MCU to control these devices.

Both input capture and output compare require the use of a Timer (either one of Timer 2 or Timer 3 can be used). As you already know, a timer simply counts from zero up to its maximum value (PRx) and then resets itself back to zero. Once initialized and turned on, the Timer circuit does not require any intervention on the MCU's part.

Purpose

This lab will walk-through the basics of input capture and output compare in the Pre-Lab Reading. You will then build a two-player tug-of-war game using two push button switches, a servo mounted flag, and an LED. Each player is assigned a push-button, and whoever clicks his/her button faster, the flag would turn towards that player. If a player manages to keep the flag towards himself/herself for at least 5 seconds, the LED will turn on, announcing the end of the game (sorry no bullhorns!)

Lab 4 is part of the intermediate labs, it will summarize a substantial portion of the datasheet and family reference manual information and present it in a more specific and digestible fashion. You will greatly benefit if you follow along in the datasheet and FRM (Family Reference Manual) sections mentioned in the text, especially because you probably have not covered IC or OC in class yet. When you reach lab 6 and the final project, you will need to be able to extend beyond what is discussed here and independently find peripheral operation descriptions and configuration registers definitions to implement a task.

Supplemental Resources

PIC24FJ64GAxxx Reference Manual

- Timers - Section 11 and 12
- Input Capture - Section 13
- Output Compare - Section 14

Microchip PIC24 Family Reference Manual - Timers
Microchip PIC24 Family Reference Manual - Input Capture
Microchip PIC24 Family Reference Manual - Output Compare

SparkFun Micro Servo
SparkFun Sonic Rangefinder

Required Components

Standard Components to use and support the PIC24FJ64GA002 (pull-up, caps, etc.)
USB to Breadboard Adapter
Two push buttons
2x Red LED T1 3/4 Diffused
2x 220 Ohm Resistor

Pre-Lab Reading

In this pre-lab reading you will learn how to setup the registers and basic operation of the timer, input capture, and output compare peripherals. The pre-lab procedure will ask you to apply this knowledge.

MCC (advanced and optional)

MPLAB Code Configurator (or MCC) is a IDE plugin that assists in creating boilerplate code. MCC has the ability to auto generate configuration code for inputs and outputs, clock/PLL configuration, as well as various peripherals on the PIC24.

While this tool can be very useful for creating working samples of code very quickly, it can be both a hindrance and asset. The programmer that uses MCC blindly is susceptible to bugs and errors in the MCC tool itself (not as uncommon as one might hope.) It is important for a programmer to be able to create configuration register settings by hand and with MCC (you need to learn manual configuration of timers/IC/OC in class and discussion sections). Use of MCC is encouraged but should always be double checked.

While this lab's examples will not explicitly use MCC, you are welcome to use it at this point. However, be aware that doing so blindly (without carefully reviewing the register map and building your own set of configuration commands) *will* cause you great headaches. Read the Manual!

If you so choose, you can learn more about MCC in the Quick Lesson - MPLAB Code Configurator.

Timers

The TMRx peripherals are automated counters (they require no or little CPU intervention to operate.) Every N clock cycles each timer will count up by one. When it reaches a predetermined value it will reset, set a flag, and optionally trigger an interrupt.

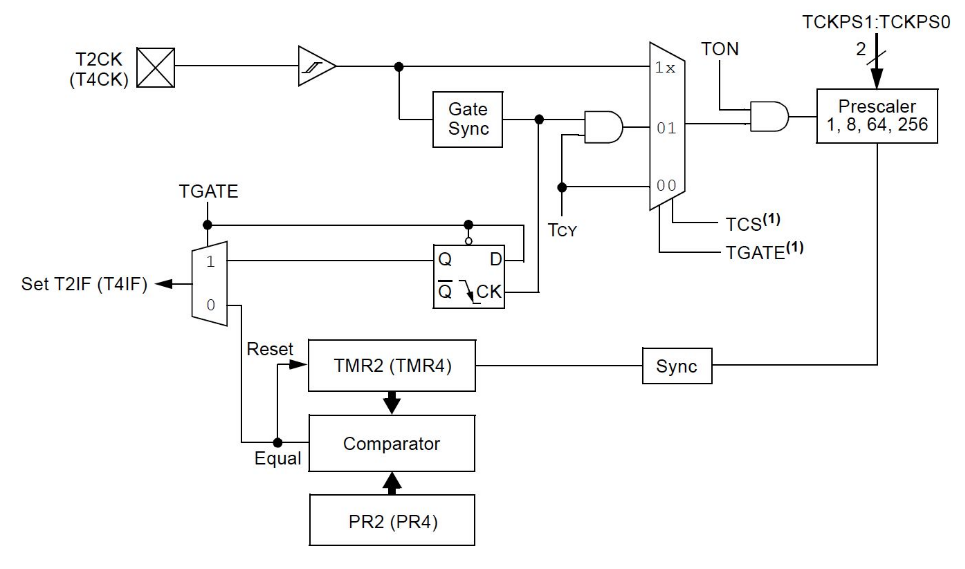
There are three types of Timers in PIC24's

- Timer 1 (Type A) - Main 16-bit timer
- Timer 2 and 4 (Type B) - 16-bit timer, can be combined with Type C for a 32-bit timer
- Timer 3 and 5 (Type C) - 16 bit timer, can trigger ADC (N/A when Type B 32-bit mode)

These timers can utilize the Fcy clock, peripheral clock, or an external pin as the "time base" (input clock to the counter.)

For this lab we'll concentrate on Timer 2, as this timer is tied to the input capture peripheral. It will be operating in 16-bit mode with the Fcy (ie., Tcy) clock as the input. Below is the block diagram for Timer 2:

FIGURE 12-2: TIMER2 AND TIMER4 (16-BIT SYNCHRONOUS) BLOCK DIAGRAM



Note 1: This peripheral's inputs must be assigned to an available RPn pin before use. Please see **Section 10.4 "Peripheral Pin Select"** for more information.

Timer 2/4 from PIC24FJGAxxx Reference Manual

TMR2 Registers

- TMR2 : Contains the current count of the timer (readable *and* writable!)
- PR2 : Highest value the timer register TMR2 will count to before resetting back to 0.

- T2CON : General Control Register for Timer 2
    - TON : Enables Timer
    - TCKPS<1:0> : Prescaler 1,8,64,256 (divides input clock to counter)
    - TCS = 0 and TGATE = 0 : Select Tcy as input

In the figure above, one can see that TMR2 and PR2 get compared every cycle for a match. When this match is detected, the *following* cycle TMR2 is reset and the T1IF flag is set (and possibly interrupts triggered). To be absolutely clear: The timer resets one cycle after TMR2=PR2, thus all periods are actually PR2+1 cycles. This is shown in PIC24 FRM - Timers in figure 14-5 replicated below:

Figure 14-5: Interrupt Timing for Timer Period Match

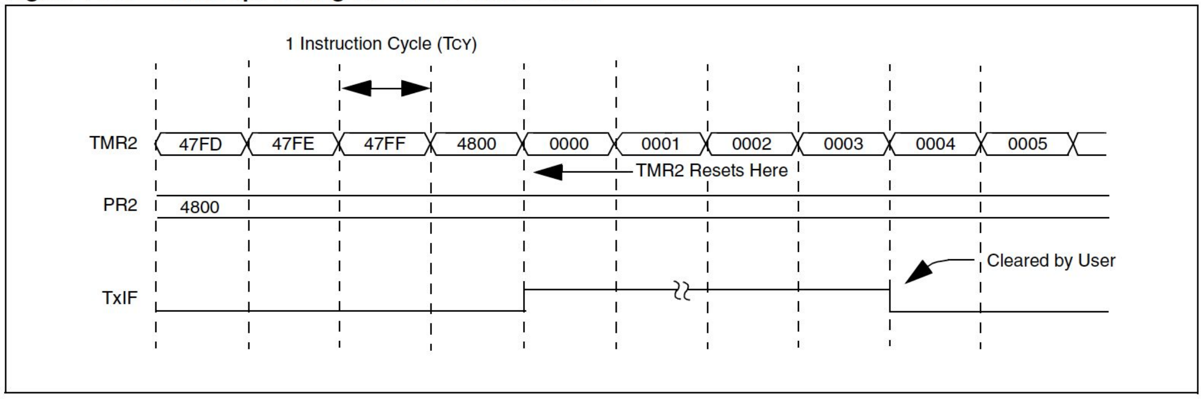


Diagram showing T2IF is set 1 cycle after TMR2 = PR2

You can read more in the Microchip PIC24 Family Reference Manual - Timers and PIC24FJ64GAxxx Reference Manual Section 11 and 12

Setting Prescaler

In order to correctly setup the duration of a timer, a programmer must calculate the duration desired in Fcy periods (Tcy = 1/Fcy = 1/16M = 62.5ns).

$$Counts = \frac{Duration\ (ns)}{62.5\ ns}$$

For example, if we wanted to measure an event of variable length between 0.1ms and 1ms, we need to count to between 1,600 and 16,000 cycles. This number *will* fit in a 16-bit counter, thus simply load the PR2 register with a value greater than or equal to the maximum value needed (15,999, which is 0x3E7F). In practice, you want to set it up to a value that is comfortably larger than the target (e.g., 20,000)

If we wanted to measure a period between 1ms to 25ms (16,000 to 400,000 clock cycles.) The count will no longer fit in a 16-bit counter. We must choose a pre-scalar (clock divider) that will prevent overflow and still have sufficient resolution for our needs. In this case, a prescaler of 8 will be give the highest resolution possible of 500ns (TCKPS<1:0> = 01). Note that as discussed in class, when you use a prescaler of 8, then a value of, say, 100 in TMR2 means it has been 100x8=800 Tcy cycles from the last reset of TMR.

Once the prescaler is determined, setting up the timer is simply a matter of setting each of the registers mentioned above and enabling the timer. It is a good practice to disable the timer during setup and enable as the last step. An example is shown below in pseudocode:

```
 //Configure Timer 2 (500ns/count, 25ms max).
// note that resolution = 500ns = 8 x 62.5ns, max period = 25ms = Tcy*8*50,000

 T2CON = 0x0010; //Stop Timer, Tcy clk source, PRE 1:8

 TMR2 = 0;     // Initialize to zero (also best practice)
 PR2 = 60000; // Set period to be larger than max external sig duration

 T2CONbits.TON = 1; // Restart 16-bit Timer2
```

Code for Timer2 setup of 500ns/count, > 25ms max timer.

Input Capture

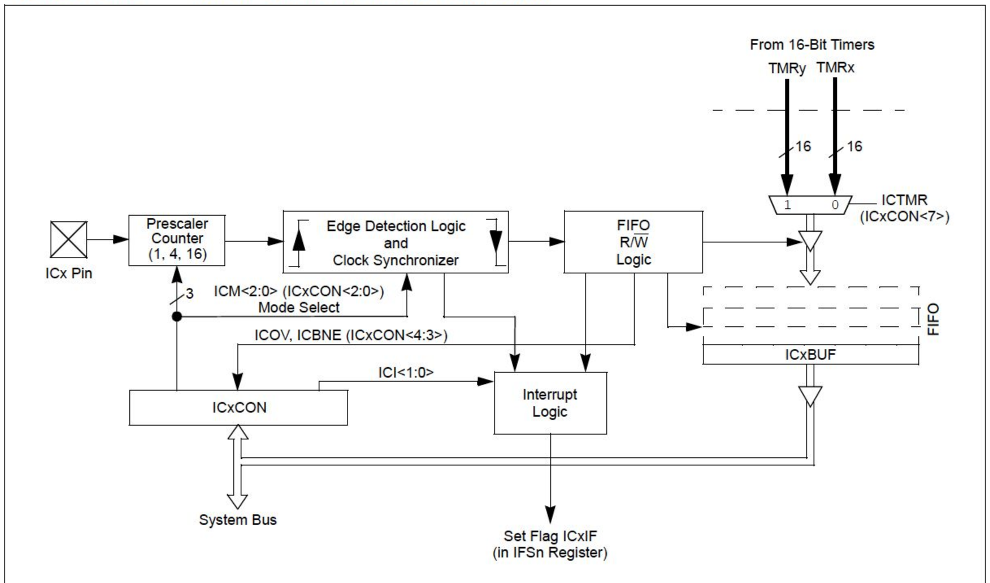
The input capture (IC) peripheral is a dedicated circuit that operates in conjunction with a timer and records the timestamp on transitions on an external pin with minimal CPU intervention. In other words, the IC hardware will monitor the pin (input), and automatically save (capture) the timer (TMR) value when an edge appears on the pin. This can be very useful for external circuits that have unknown (both fast and slow) response times. For example, a motor *encoder* may send a pulse every time the motor turns 1 degree. If your vehicle is traveling at 80 mph, then the pulses come roughly every 30us. If your vehicle is stopped, the time to the next pulse is infinite and you wouldn't want your MCU to wait for that next pulse, when it could be doing more important things like adjusting the volume on your stereo.

IC peripheral operates in conjunction with a timer and is usually set up as an interrupt (although it can work with polling too.) It will store the current timer value in a 4-word FIFO buffer when the input signal changes state. There are options for responding to rising, falling or both edge types. There are even options to count sets of multiple edges before recording a timer value (IC prescaler, which works independently of the timer prescaler).

In this lab we will be using the Input Capture circuitry to detect falling edges from push button switch actions controlled by the players. The time difference between the falling edges would determine the rate of clicks. The player who clicks his/her button faster will win.

There are five input capture modules in your PIC chip, and you will be using IC1 and IC2 in this lab. The input capture circuit has the following architecture, described in section 13 of the PIC24FJ64GAxxx Family Datasheet and PIC24 FRM - Input Capture. The figure refers to ICx, where x can be 1, 2, …, 5. TMRx and TMRy refer to Timer 3 and Timer 2 respectively. Note that the default timer assigned to IC modules is Timer 3 (you will later see that the default timer for OC is Timer 2). The assigned timer is controlled using the bit ICTMR.

FIGURE 13-1: INPUT CAPTURE BLOCK DIAGRAM


From 16-Bit Timers
TMRy TMRx
16
16
1
0
ICTMR (ICxCON<7>)
ICx Pin
Prescaler Counter (1, 4, 16)
Edge Detection Logic and Clock Synchronizer
FIFO R/W Logic
3
ICM<2:0> (ICxCON<2:0>) Mode Select
ICOV, ICBNE (ICxCON<4:3>)
FIFO
ICxBUF
ICI<1:0>
ICxCON
Interrupt Logic
System Bus
Set Flag ICxIF (in IFSn Register)

Note 1: An 'x' in a signal, register or bit name denotes the number of the capture channel.
2: This peripheral's inputs must be assigned to an available RPn pin before use. Please see **Section 10.4 "Peripheral Pin Select"** for more information.

Figure 13-1 from PIC24FJ64GAxxx Datasheet

ICx Core Registers

- ICxBUF : FIFO containing timer values for (at most the last 4) edge events
- ICxCON : General Control Register for Input Capture x
  - ICM<2:0> : Mode Select. A selection of the modes is shown below:
    - 011 = Capture mode, every rising edge
    - 001 = Capture mode, every edge (rising and falling)
    - 000 = Input capture module turned off
  - ICTMR : Input Capture x Timer Select bit
    - 1 = TMR2 contents are captured on capture event
    - 0 = TMR3 contents are captured on capture event

- ICBNE : Buffer not empty flag (1 = FIFO has data)

In addition to setting up the above registers, Timer 2/3 will have to be configured similar to that described in the Timers section shown above. The *peripheral select bits* will also need to be configured to route the correct RPx pin to the IC1 input pin as described in the Peripheral Select section of this manual.

During an edge event the Input Capture peripheral will store the current TMRx value into the FIFO (technically 1 cycle after the event occurs.) This is shown in the diagram below:

Figure 5-3: Edge Detection Mode Timing Diagram

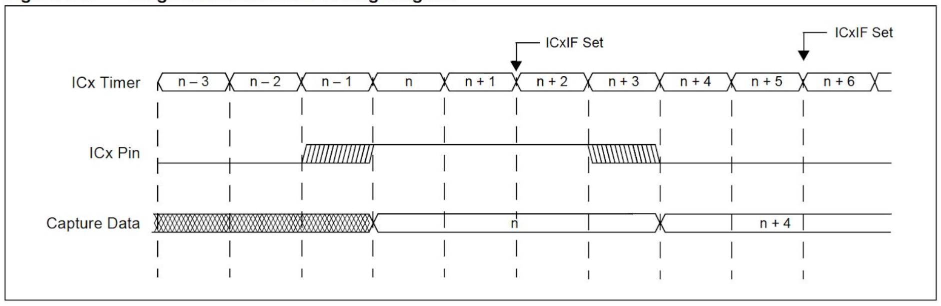


Figure 5-3 from PIC24 Family Reference Manual - Input Compare

Note: The line "capture data" (above) is the value added to the FIFO (ICxBUF). Also note the ICxIF events.

Once the programmer has determined the timer settings, edge detection method (rise/fall), and any needed prescalers, the following 2-line code describes how to setup the input capture peripheral.

```
IC1CON = 0; // Turn off and reset internal state of IC1
IC1CONbits.ICTMR = 1; // Use Timer 2 for capture source
IC1CONbits.ICM = 0b011; // Turn on and capture every rising edge
```

Setup Code for Input Capture 1 in Rise Mode

The following pseudocode shows a typical input capture reset (note that this is a pseudocode and will not compile as is):

```
Timer_y_TON = 0; //Stops 16-bit Timer y
ICxCON_ICM = 0x000; // Capture off
PR_y = 0xffff;  // minimize rollovers
```

```
    TMRy = 0;      // Initialize timer y to zero
    Timery_TON = 1; //Starts 16-bit Timer y
    ICxCON_ICM = 0x001; // Capture rising and falling edges
```

Resetting IC and its associated timer.

After the pulse is recorded it is simply a matter of reading from the FIFO and then operating on the result.

The method to recover the rising and falling edge times is shown in the following pseudocode:

```
volatile long int curPeriod=0;  // we probably need a buffer, and
                                // not a single int variable.

IC1Interrupt() { // NOTE THAT THIS IS A PSEUDO-CODE and misses the
                 // interrupt attributes
   static unsigned int prevEdge=0;
   int curEdge;

   _IC1IF = 0;
   curEdge = IC1BUF;
   curPeriod = curEdge - prevEdge;
   prevEdge = curEdge;
   // Note that we have not considered timer overflow, assuming
   // signal periods are smaller than PRx. You have to consider
   // overflows in your implementation
}
```

Example pseudocode to calculate the period of the signal using Input Capture

How to Handle Push-Buttons and Debouncing

If you remember from the previous lab(s), we need to debounce switches to make sure we do not treat transient pulses as legitimate input intended as new clicks by the user. The common method of debouncing switches is to wait for the first falling edge, and then ignore activity on the pin for a period of time, say, 2ms. This can be done easily when handling switch input using polling loops. The same can be done when implementing switch handling using interrupts -- which is what we will do in this lab -- but we can instead use a simpler method to handling debouncing.

We suggest using this simple method to handle debouncing: since we know that the source of the input capture pin is a push-button controlled by a user, any activity that is faster than 2ms has to be due to switch imperfections and hence has to be ignored.

Buffers and Averaging

Since users cannot maintain exactly the same rate of clicking a button, we have to take the average of the periods of the most recent 4 clicks and use that as the rate of clicks. You can use a circular software buffer (not to be confused by the hardware buffer used in the IC module), and store the periods calculated in the IC ISR (the above code). We can then quickly calculate the average of the values in the buffer. All this can be done inside the IC ISR.

Output Compare

Output compare is a peripheral that can maintain a constant switching signal on an output pin with no CPU intervention. This is extremely useful to control a variety of devices for example:

- Servos with pulse-width modulated (PWM) inputs
- Mixing and Dimming LEDs
- Motor Drivers - Warning: NEVER connect a motor directly to your PIC24! Use a power FET (uni-directional) or an H-bridge like DRV8835 (bi-directional)
- Reduced MCU overhead when bit banging slow serial connections
- Driving speakers with simple beeps

Output compare can drive an output pin with a number of different waveforms, including:

- Toggle high/low once at a specific time in the future
- Toggle high+low continuously at a specific frequency
- Output a specific pulse width at a given frequency

This lab will concentrate on the last of these, PWM mode. Note: There is an option for a hardware fault control in this peripheral. This is designed to stop all motor drives (not PWM servos) during detection of an over-current or over-temperature fault. This lab will not operate with fault support.

OC1 Core Registers

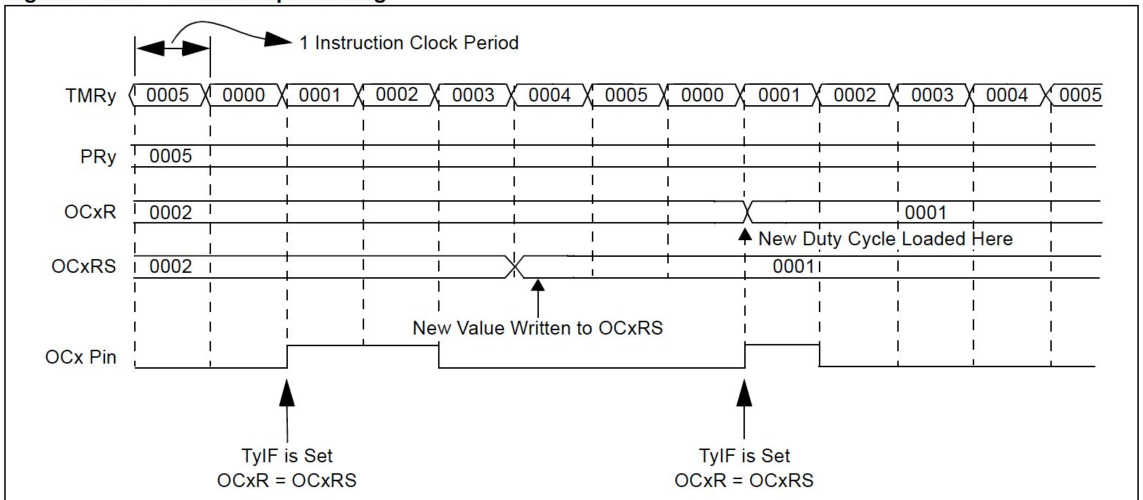
- OC1R : Count when output state is changed (in PWM mode pin is forced LOW)
- OC1RS : Count when output state is changed (in PWM mode acts as a shadow reg)
- OC1CON : General Control Register for Output Compare 1
  - OCM<2:0> : Mode Select
    - 000 = Output compare channel is disabled
    - 110 = PWM mode on OC1, Fault pin, OCFx, disabled
  - OCTSEL : Timer Select
    - 1 = Timer3 is the clock source
    - 0 = Timer2 is the clock source

In addition to setting up the above registers, Timer 2/3 will have to be configured similar to that described in the Timers section shown above. The peripheral select bits will also need to be configured to route the correct RPx pin to the OC1 input pin as described in the Peripheral Select section of this manual.

Let us assume we have chosen Timer 3 to be paired with the output compare module. The output compare circuit in PWM mode will set the OCx pin high when TMR3=0. The pin will remain high until TMR3=OC1R, after which the pin will go low and remain low until TMR3 goes all the way up to PR3. With the normal reset of Timer 3 (TMR3 will go from PR3 → 0), a new cycle starts. The Output Compare PWM mode utilizes OC1RS as a shadow register to prevent "glitches" in the output waveform during asynchronous register writes. During normal operation the programmer should only write to OC1RS (OC1R should be set to the desired initial duty-cycle only right before turning the OC module on, i.e., setting OCM=xxx).

The default behavior and asynchronous write behavior are illustrated in the diagram below:

Figure 4-18: PWM Output Timing(1,2)



Note 1: An 'x' represents the Output Compare channel number. A 'y' represents the time base number.
2: OCxR = Output Compare x register, OCxRS = Output Compare x Secondary register.

Example of PWM register shadowing

The duty cycle is defined as the time the waveform is HIGH. It can be measured in either percent (%) or time (sec). Duty cycle is simply calculated with the timer time base as follows:

$$PWMperiod\,(sec) \;=\; (PRy + 1) * 62.5ns \; * TMRxPreScaler$$

$$DutyCycle\,(sec) = OCxR * 62.5ns \; * TMRxPreScaler$$

Once the programmer has determined the correct pins, timer settings, and duty cycle. The output compare peripheral can be setup for PWM mode with the following pseudocode:

```
    // Timer 3 setup should happen before this line
    OC1CON = 0;     // turn off OC1 for now
    OC1R = 1234;    // servo start position. We won't touch OC1R again
    OC1RS = 1234;   // We will only change this once PWM is turned on
    OC1CONbits.OCTSEL = 1; // Use Timer 3 for compare source
    OC1CONbits.OCM = 0b110; // Output compare PWM w/o faults
```

Pseudocode for the setup of Output Compare 1 in PWM mode

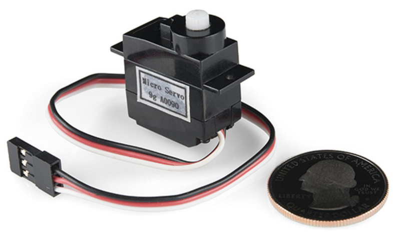
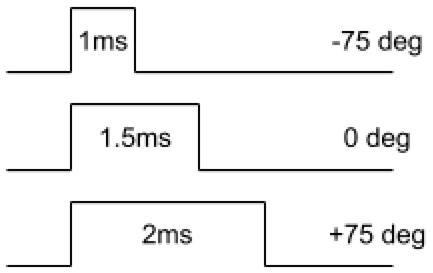
And modified during operation with the single instruction:

```
        OC1RS = 4321;
```

Pseudocode for modifying Output Compare 1 duty cycle during normal operation

Standard Servos

Standard "servos" available to hobbyists contain a motor, encoder, and control circuit. They operate based on a single input command signal. This signal represents the desired angle that the user wants the servo to point towards. The internal control circuit uses the encoder in a tight local feedback loop to force the servo to rotate to specific angle and then holds that position (even when the servo arm is pushed.) This makes servos extremely useful for steering R/C models or control of robotic components (Alpha 1S).

Typical PWM controlled "analog" hobby servo with signalling diagram

Most servo's have a maximum range of motion of between 150° to 180°. The exact relationship between angle and input pulse width is not usually defined. If very high precision is needed, servos are usually calibrated in software or a fully custom motor control system is employed.

Peripheral Pin Select (PPS)

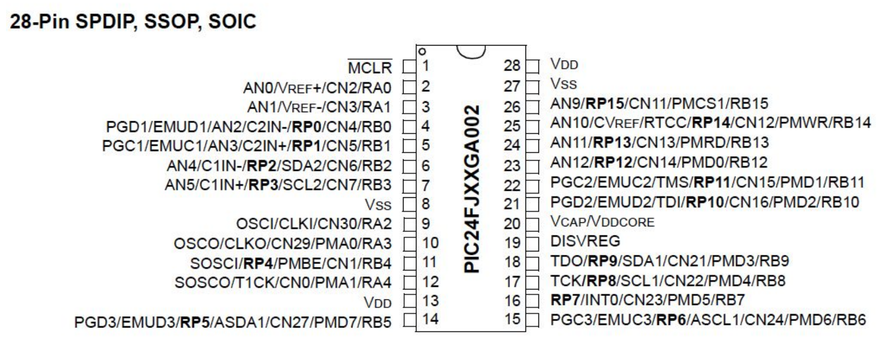
Many of the peripherals in the PIC24 have re-configurable inputs and outputs. This is a feature largely available in low pin count MCU's and CPU's. For example, there may be significantly more internal functions than available pins. In addition, one device may be used in a variety of hardware settings and the pins which are attached to each external device may change or be significantly restricted.

In the PIC24FJ series the Peripheral Pin Select functionality is controlled by two sets of registers one for the output pins/peripherals and one for the input pins/peripherals. These control registers can *not* be changed arbitrarily as it can adversely impact the functionality of some peripherals. Read this entire section, especially the section at the end “Unlocking Peripheral Select”.

Warning!!! If your Peripheral Select Registers are not changing, you must unlock them first!

From the PIC24FJ64GAxxx Datasheet :
“Under normal operation, writes to the RPINRx and RPORx registers are not allowed. Attempted writes will appear to execute normally, but the contents of the registers will remain unchanged.”

The diagram on page 4 of the PIC24FJ64GAxxx Family Datasheet is a nice reference to determine which pins are available for various peripherals. Pay special attention to the RPx designations.


28-Pin SPDIP, SSOP, SOIC
PIC24FJXXGA002
MCLR 1
AN0/VREF+/CN2/RA0 2
AN1/VREF-/CN3/RA1 3
PGD1/EMUD1/AN2/C2IN-/RP0/CN4/RB0 4
PGC1/EMUC1/AN3/C2IN+/RP1/CN5/RB1 5
AN4/C1IN-/RP2/SDA2/CN6/RB2 6
AN5/C1IN+/RP3/SCL2/CN7/RB3 7
VSS 8
OSCI/CLKI/CN30/RA2 9
OSCO/CLKO/CN29/PMA0/RA3 10
SOSCI/RP4/PMBE/CN1/RB4 11
SOSCO/T1CK/CN0/PMA1/RA4 12
VDD 13
PGD3/EMUD3/RP5/ASDA1/CN27/PMD7/RB5 14
28 VDD
27 VSS
26 AN9/RP15/CN11/PMCS1/RB15
25 AN10/CVREF/RTCC/RP14/CN12/PMWR/RB14
24 AN11/RP13/CN13/PMRD/RB13
23 AN12/RP12/CN14/PMD0/RB12
22 PGC2/EMUC2/TMS/RP11/CN15/PMD1/RB11
21 PGD2/EMUD2/TDI/RP10/CN16/PMD2/RB10
20 VCAP/VDDCORE
19 DISVREG
18 TDO/RP9/SDA1/CN21/PMD3/RB9
17 TCK/RP8/SCL1/CN22/PMD4/RB8
16 RP7/INT0/CN23/PMD5/RB7
15 PGC3/EMUC3/RP6/ASCL1/CN24/PMD6/RB6

PIC24FJ64GAxxx Family Datasheet: Pin Diagram for our 28-pin SPDIP package

Input Pin Selection

For input pins each individual peripheral has a register for each of its input pins. The programmer sets this register to the value of the RPx pin shown in the diagram above. Note: if multiple peripherals designate the same RPx pin, they will both use that pin (for example to share a clock line).

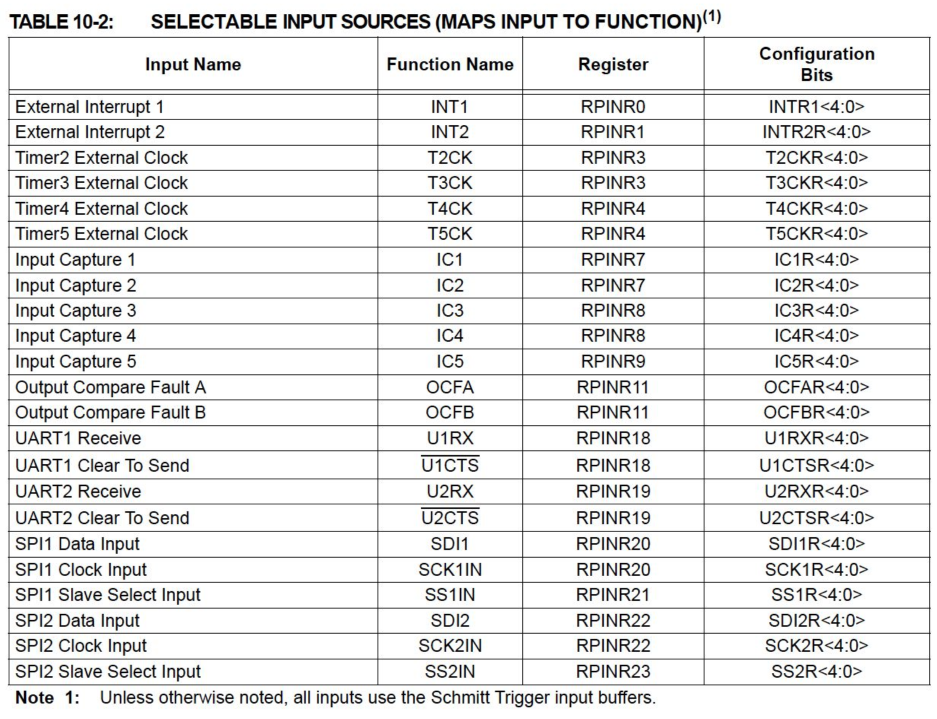


TABLE 10-2: SELECTABLE INPUT SOURCES (MAPS INPUT TO FUNCTION)[1]

Input Name	Function Name	Register	Configuration Bits
External Interrupt 1	INT1	RPINR0	INTR1<4:0>
External Interrupt 2	INT2	RPINR1	INTR2R<4:0>
Timer2 External Clock	T2CK	RPINR3	T2CKR<4:0>
Timer3 External Clock	T3CK	RPINR3	T3CKR<4:0>
Timer4 External Clock	T4CK	RPINR4	T4CKR<4:0>
Timer5 External Clock	T5CK	RPINR4	T5CKR<4:0>
Input Capture 1	IC1	RPINR7	IC1R<4:0>
Input Capture 2	IC2	RPINR7	IC2R<4:0>
Input Capture 3	IC3	RPINR8	IC3R<4:0>
Input Capture 4	IC4	RPINR8	IC4R<4:0>
Input Capture 5	IC5	RPINR9	IC5R<4:0>
Output Compare Fault A	OCFA	RPINR11	OCFAR<4:0>
Output Compare Fault B	OCFB	RPINR11	OCFBR<4:0>
UART1 Receive	U1RX	RPINR18	U1RXR<4:0>
UART1 Clear To Send	$\overline{\text{U1CTS}}$	RPINR18	U1CTSR<4:0>
UART2 Receive	U2RX	RPINR19	U2RXR<4:0>
UART2 Clear To Send	$\overline{\text{U2CTS}}$	RPINR19	U2CTSR<4:0>
SPI1 Data Input	SDI1	RPINR20	SDI1R<4:0>
SPI1 Clock Input	SCK1IN	RPINR20	SCK1R<4:0>
SPI1 Slave Select Input	SS1IN	RPINR21	SS1R<4:0>
SPI2 Data Input	SDI2	RPINR22	SDI2R<4:0>
SPI2 Clock Input	SCK2IN	RPINR22	SCK2R<4:0>
SPI2 Slave Select Input	SS2IN	RPINR23	SS2R<4:0>

Note 1: Unless otherwise noted, all inputs use the Schmitt Trigger input buffers.

Table 10-2 from PIC24FJ64GAxxx Family Datasheet

TABLE 10-2 in the PIC24FJ64GAxxx Family Datasheet shows the peripheral input name and the corresponding register name. To set the input selection to RPx simply load the number "x" into this register to set the input to RPx.

For example, to configure the Input Compare 1 peripheral the IC1R<4:0> register needs to be set to 9 in order to connect it to RP9 (also known as RB9).

Output Pin Selection

For output pins there is one register for each RPx pin called RPxR. This register is set to an ID number between 0 and 31 that represents an output from the peripherals on the PIC24. Note: It is not possible to share an output between two peripherals.

TABLE 10-3 in the PIC24FJ64GAxxx Family Datasheet shows the peripheral output name and the a corresponding ID number (Output function number). To select an output pin for a peripheral simply load the appropriate RPxR register with the ID number.

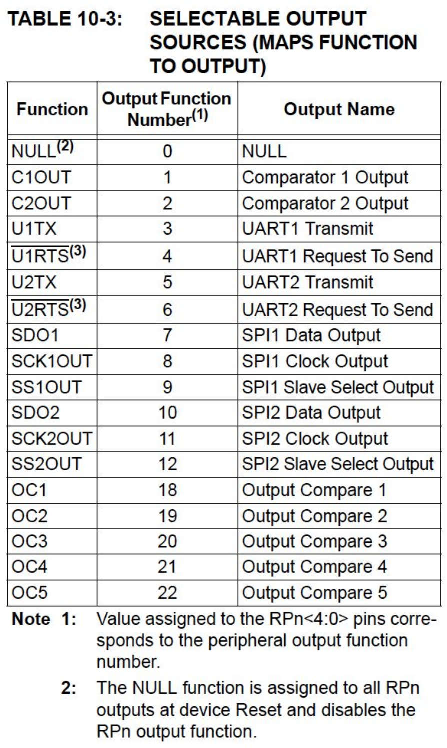


TABLE 10-3: SELECTABLE OUTPUT SOURCES (MAPS FUNCTION TO OUTPUT)

Function	Output Function Number[1]	Output Name
NULL[2]	0	NULL
C1OUT	1	Comparator 1 Output
C2OUT	2	Comparator 2 Output
U1TX	3	UART1 Transmit
$\overline{\text{U1RTS}}$[3]	4	UART1 Request To Send
U2TX	5	UART2 Transmit
$\overline{\text{U2RTS}}$[3]	6	UART2 Request To Send
SDO1	7	SPI1 Data Output
SCK1OUT	8	SPI1 Clock Output
SS1OUT	9	SPI1 Slave Select Output
SDO2	10	SPI2 Data Output
SCK2OUT	11	SPI2 Clock Output
SS2OUT	12	SPI2 Slave Select Output
OC1	18	Output Compare 1
OC2	19	Output Compare 2
OC3	20	Output Compare 3
OC4	21	Output Compare 4
OC5	22	Output Compare 5

Note 1: Value assigned to the RPn<4:0> pins corresponds to the peripheral output function number.

2: The NULL function is assigned to all RPn outputs at device Reset and disables the RPn output function.

For example, to configure the Output Compare 1 peripheral the RP6R<4:0> register needs to be set to 18 in order to connect it to RP6.

Unlocking Peripheral Pin Select (PPS)

During normal operation changes to the RPxR and xxxR<4:0> registers are not allowed. These registers must be unlocked, changes made, and then relocked to be successfully written. In order to unlock PPS the programmer must write the a specific value to the OSCCON register as described section 10.4 in the datasheet. A full-code snippet is provided below for illustration:

```
    __builtin_write_OSCCONL(OSCCON & 0xbf); // unlock PPS
    RPINR7bits.IC1R = 9;  // Use Pin RP9 = "9", for Input Capture 1 (Table 10-2)
    RPOR3bits.RP6R = 18;  // Use Pin RP6 for Output Compare 1 = "18" (Table 10-3)
    __builtin_write_OSCCONL(OSCCON | 0x40); // lock   PPS
```

Example of how to unlock PPS

While these registers can be changed in the middle of normal operation (with a unlock/relock sequence) it is recommended to only change PPS immediately after RESET. For example in setup(). Peripherals that are connected/disconnect in the middle of an operation may require extra work to reset correctly.

USB power connector (5V)

Provided in your lab kit is a USB breakout board (you should have soldered pins to this board in the first week). We will use this to provide the 5V that the servo requires. Simply connect GND to the joint ground on your breadboard. Be very careful to only supply 5V to the HRxx and Servo Motor. Applying 5V to your PIC24 will release the magic smoke!

It is recommended that you supply the USB connector through a “power brick”, a usb-based battery, or your PC USB port. You can use the micro-USB cable that came in your EE 1301 lab kit.

It is possible to operate your full breadboard just on the USB power using the supplied regulator. Please see the Quick Lesson - Regulator Power if you’d like to take this step. Note, you will still need to program your PIC24 through the chipKIT PGM.

Putting the Software Together

The following tasks must be handled in this lab:

- IC events have to be captured and the delay between consecutive falling edges have to be calculated. This "period" is discarded if <2ms, otherwise stored in a software buffer of size 4. Note that we have two IC inputs each from a user.
- Timer overflows have to be considered when calculating the period of clicks.
- The average of the four buffer values has to be calculated and stored in a global variable to be used by the code that calculates the servo position.
- The difference between the two average click delays should determine which way the servo will point to. If User 1 is clicking way faster than User 2, then the servo should be tilted all the way towards User 1. Otherwise, if User 1 is barely clicking faster than User 2, the servo should be tilted only slightly towards User 1. You should experimentally find the right equation for translating the average delay difference to the angle of the servo.

You are supposed to use interrupts to handle IC events, storing the 4 most recent click periods in a buffer, and calculating the average value (all of these operations inside the ICx ISR). A volatile global variable will store the average period of a user's clicks (two such variables because we have two players). You can choose to handle OC operations either using polling in main, or as interrupts, but we suggest using polling.

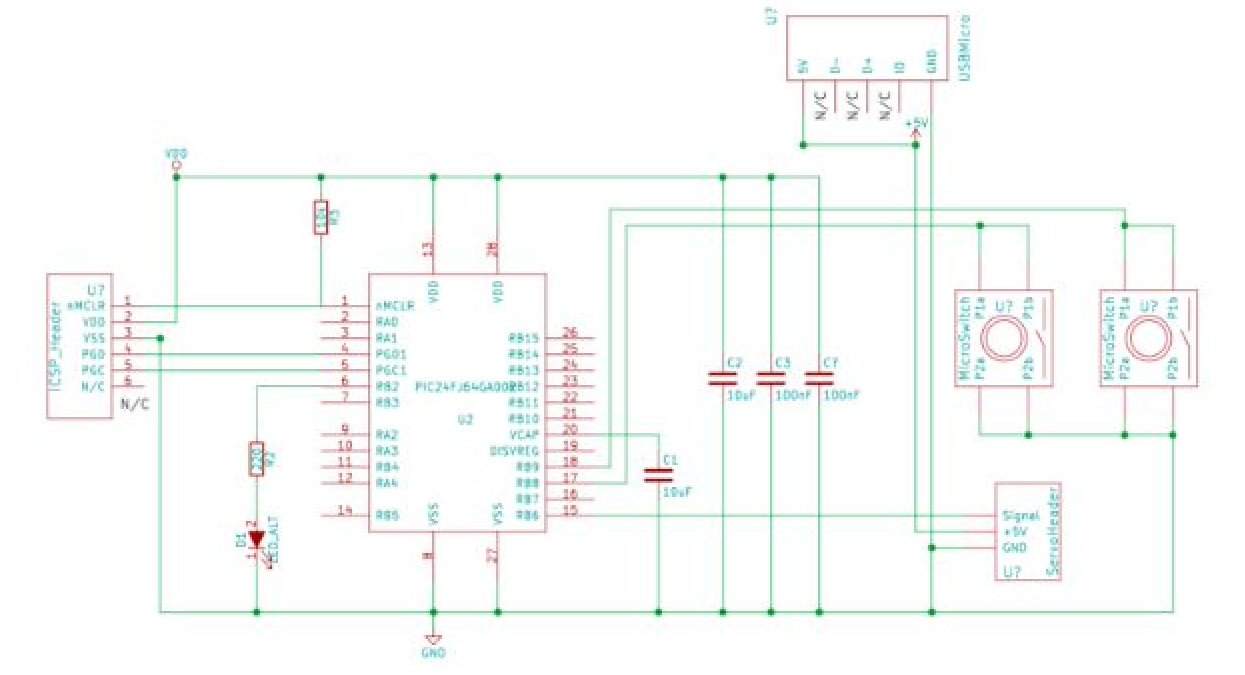
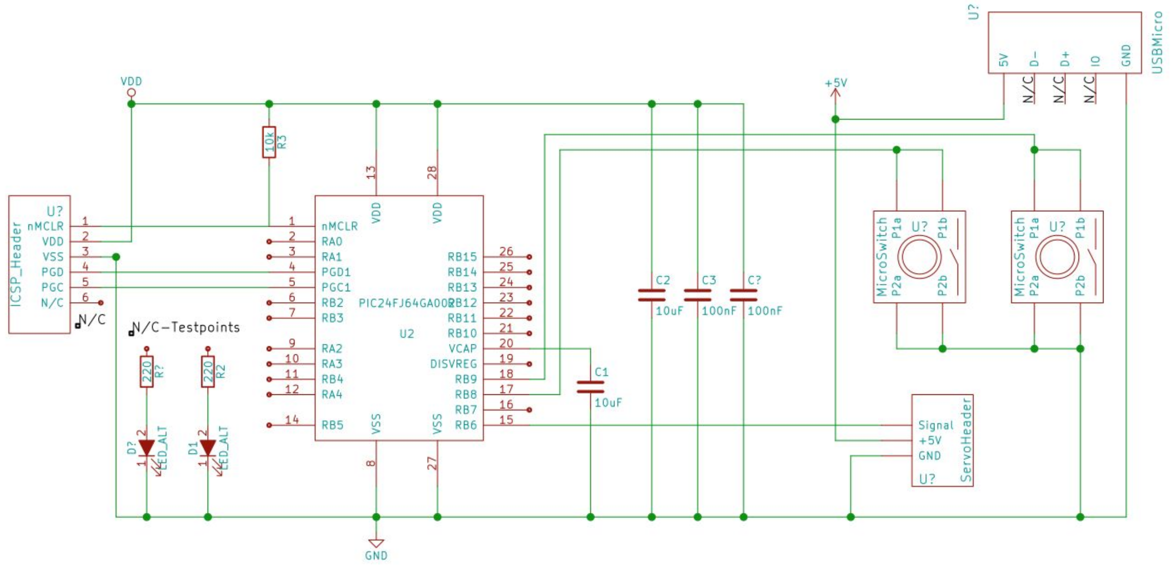
Pre-Lab Procedure

The procedure below outlines the new basic functionality required for the final components of lab 3. Completing the following is a good solid start on the lab, but it can be helpful to continue into the main lab procedure if you have time.

Wiring

1.) Set up your Push-button switches and USB Power adapter using the schematic below. (In addition, if you want to debug the output compare module, you may want to add a resistor/LED between the line for the servo signal (RB6) and ground.)

Note how +5V is separate from the rest of the circuitry and only routed to the servo. Until now we have used 3.3V to power the microcontroller. Sometimes it can be ok to route 3.3V to a 5V part. It is never ok to route 5V to a 3.3V (magic smoke will be released.)

Schematic for push-buttons and LED monitored Output Compare

Warning DO NOT connect the USB power (5V) to the Breadboard "rails". Keep VCC (5V) completely separate from VDD (3.3V) supplied to the rest of your board.

Setup Output Compare (w/ Timer 3)

As the PWM-based servo code is the easier code to get operational, you will want to begin by setting up this device first. A servo is not provided in your lab kit (one will be provided for your use during the lab.) However, you can test this code simply by providing the signal to an LED test point.

2.) Create an initialization function for the Servo

```
void initServo(void);
```

This function utilizes PPS to bind Output Compare 1 to RP6/RB6, and standard register writes to initialize Timer 3 to have a period of 20ms (you will need to calculate values for TCKPS and PR3 as well as setup the rest of the OC1 registers.) You will also need to set AD1PCFG and TRIS bits as appropriate.

3.) Create a "set output" function for the Servo

```
void setServo(int Val);
```

Hint: This is the easiest function you will ever write!

4.) Test your output compare by outputting alternating values of 25% duty cycle (5ms HIGH) and 75% duty cycle (15ms HIGH) every 2 seconds (this will show up nicely on your test LED.)

NOTE: Do not drive a servo with these pulse widths!!! The servo requires a duty cycle of 0.5 to 2ms on a 20ms period.

Be prepared to demo your servo control to your lab TA at the beginning of the lab.

Setup Input Capture (w/ Timer 2)

5.) Next, you will need to setup an initialization function for the push-buttons.

```
void initTwoPushButtons(void);
```

This function utilizes PPS to bind Input Capture 1 to RP8/RB8, and bind IC2 to RP9/RB9. It also uses standard register writes to initialize Timer 2 to have a period > 1s to avoid overflows as much as possible. Both input capture modules should be setup and configured to use Timer 2 and capture falling edges (although capturing both falling and rising edges is OK too, as long as you use the same standard for both users). You will also need to set AD1PCFG and TRIS bits as appropriate.

6.) Implement the ISR for both the IC modules.

You definitely need to test these ISRs in the simulator and provide different stimuli (e.g., simulating fast transitions as might happen in debouncing to see if your ISR ignores such fast events).

For the purposes of the pre-lab, you only need to show that you calculate the period (with or without overflow), as this would be the preliminary version of the ISR that you will develop during the whole lab period.

As for the final version of the ISR, you should definitely write the code in stages and thoroughly test it before making additions. For example, first implement the code to calculate signal periods without overflow, store the long ones in the circular buffer and discard the fast events. Once you adequately test this function, then more on to include overflow as well. Once you test it with the overflow, add the averaging, and so on.

Once you have tested the code in the simulator, you can use the hardware-based “Debug Main Project” mode with break points inside the ISRs. This way you can investigate the recorded timer values, step through the ISR and check calculations.

Pre-Lab Checklist

- ❏ Complete Pre-Lab reading
- ❏ Working code to initialize Timer1, Input Capture 1, and Output Compare 1
    - ❏ `void initServo(void);`
    - ❏ `void setServo(int Val);`
    - ❏ `void initTwoPushButtons(void);`
    - ❏ `IC1 and IC2 ISRs`
- ❏ A demo of the PWM output compare driving the LED (2 brightnesses)
- ❏ A demo of a very basic functionality of both the IC ISRs. You should show that you can trigger and capture a variable pulse width (breakpoint inside the two IC ISRs and showing to the TA that your code does stop there when you push each button, and it calculates the period without overflow considerations for now).

Lab Procedure

In the main procedure of this lab you will create a two-player tug-of-war game. Each user will control a push-button. Whoever clicks his/her button at a faster rate, the servo will tilt towards that player. You will need a Servo (provided by your lab TA) and push buttons. You will put these together with your code from the pre-lab to create the game.

Servo setup and debug

1.) Adjust the output compare to output a pulse width of 1.5ms with a period of 20ms.
2.) Obtain a servo from your lab TA (these devices are to remain in the lab.) Connect it to your output compare pin. Your servo should activate, move to 0 degrees (middle) and actively hold there (try pushing on it lightly.)
3.) Change the pulse width to 1.8ms and 1.2ms each, your servo should switch to (roughly) +- 75 degrees.

Putting it all together

4.) Complete the code by adding averaging, delay difference calculations and make a complete working circuit and code. The game should behave as follows:
   a.) Start the game when each button has been pressed at least once.
   b.) For each player, record the average period between button presses using the 4 most recent measurements within the last second.
   c.) The player with the lower period should have the servo directed toward him/her. If both players are tied, the servo should not point in either direction.
   d.) If a player has the servo directed toward him/her for at least five seconds, the LED connected to RB2 should light up to signal the end of the game.

Post-Lab Checklist
- ❏ Show your IC ISR code to your lab TA.
  - ❏ Show how you handle overflow events and use them in your period calculations.
  - ❏ Show how the circular buffer and averaging code works.
- ❏ Show your code handling the delay difference and controlling the servo to your TA.
- ❏ Demo your circuit to your TA

Lab Report

Put together a brief report on the work in this lab. It should include the following:

- Documentation for the functions you created
- The stages in which you wrote the code, and how you verified the correctness of that stage.
  - Examples of “stages” would be: 1-enabled IC interrupts and calculated the period without overflow. 2- Added overflow calculations, 3-Added the code to ignore pulse widths less than 2ms, 4-Added the circular buffer, …
  - List the test cases you used in each stage in a table.
- Describe your observations of the performance of the game and if you observe any errors or anomalies.

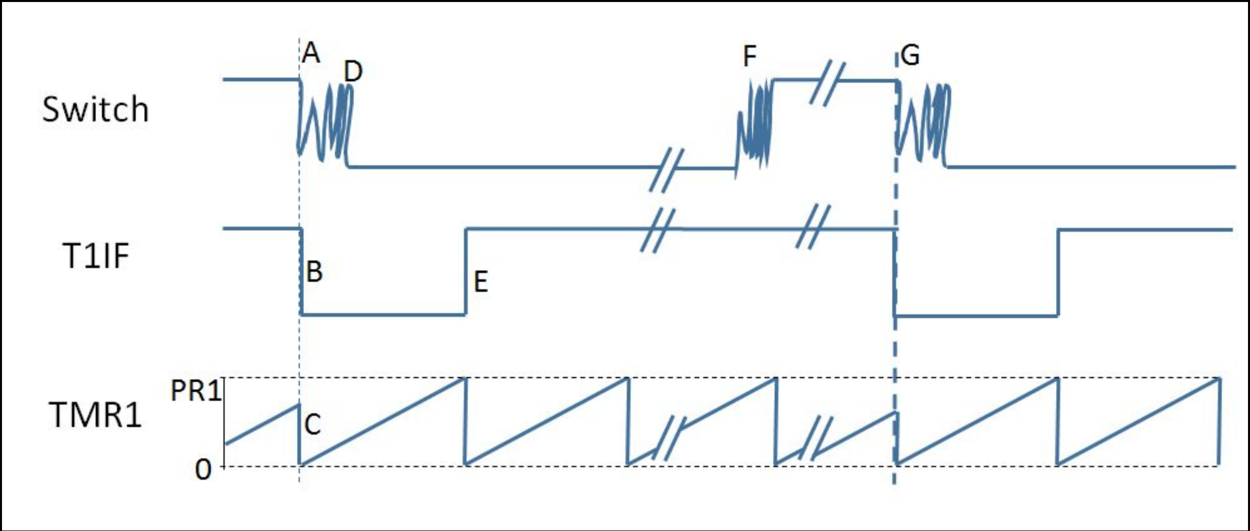
Reminder: The Lab TA must see a demo of your circuit.

Going Further (Optional)

If you like a programming challenge, you can use a more complicated method for handling debouncing. You can use one of the following methods that rely on a timer (e.g., Timer 1) to keep track of 2ms.

Method 1: continually keep track of 2ms intervals

In this method, Timer 1 will always be on and active. If T1IF is 1, it means at some point in the past we have passed the 2ms mark (maybe many times over, but at least once). When we see a falling edge, we first check to see if T1IF is 1. If it is, then it is safe to consider the edge as a legitimate event. Before returning from the ISR, we reset TMR1 to 0 and reset T1IF to 0. If another quick falling edge happens due to imperfections in the switch, we know we have to ignore it because T1IF is not 1 yet. The following figure shows how this method works:



Method 1 for Debouncing in an ISR

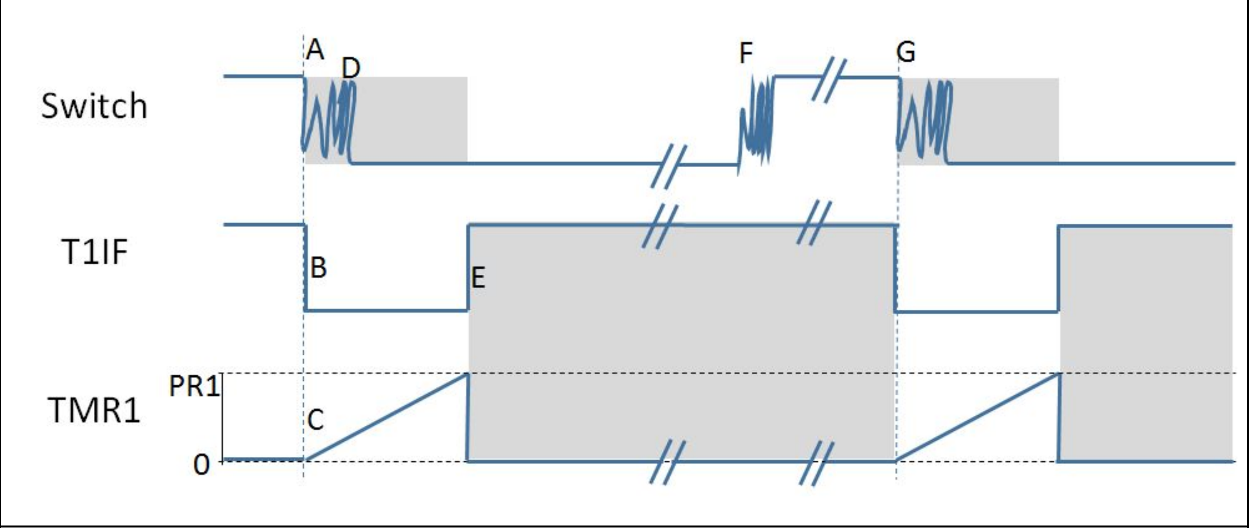
The TMR graph shows the value of TMR1 changing linearly from 0 to PR1, and then resetting back to zero by the timer hardware. That is why it looks like a sawtooth figure.
The breakage in all three graphs show potentially long periods between the user depresses the button or releases the button.

In the figure above, the following events have been labeled using letters:

A) The user pushes the button
B) In the IC ISR, we reset T1IF
C) In the IC ISR, we reset TMR1=0 to mark the beginning of a 2ms period.
D) Activity here does cause the processor to go to the IC ISR, but the ISR checks T1IF first, and recognizes these events as noise. It reads them from the IC1BUF and discard them.
E) The end of the 2ms period is reached. Any activity after this point will be considered by the IC ISR.
F) You have to think of a mechanism to ignore the noise in this region (or, change your strategy to calculate both high-pulse and low-pulse durations as part of the switching speed by the user).
G) A new cycle starts and this point is similar to A).

Method 2: Enabling and Disabling IC / Timer Interrupts

In this method, the IC ISR will disable itself (set _IC1IE=0) when it first detects a legitimate edge, and before returning, sets up Timer 1 ISR, asking it to “wake IC up” after 2ms. Timer 1 ISR will turn itself off and before returning, enable the IC interrupt again. The following figure shows how this method works:



Method 2 for Handling Debouncing

The following events have been labeled as:

A) The user pushes the button. The IC ISR reads the IC1BUF, disables the IC interrupt, and turns off IC (ICM=0). The reason we turn it off is because we do not want spurious activities fill the IC1BUF to be read later as legitimate edges.
B) The IC ISR also resets T1IF and enable its interrupts T1IE.
C) The IC ISR also resets TMR1 to start a 2ms period.
D) These activities will not register with the IC unit and will be ignored.
E) The Timer ISR will disable T1IE (in the figure, it also turns off the timer, although this is really not necessary). The Timer ISR enables ICIE and also turns on the IC unit again.
F) This is not handled in the figure, but it should. Refer to the comments in Method 1.
G) This is a new cycle and similar to A).