Infrared Decoder and Output Board

Jaren Jeffery

Capstone Project 2016-2017

Table of Contents

Proposal

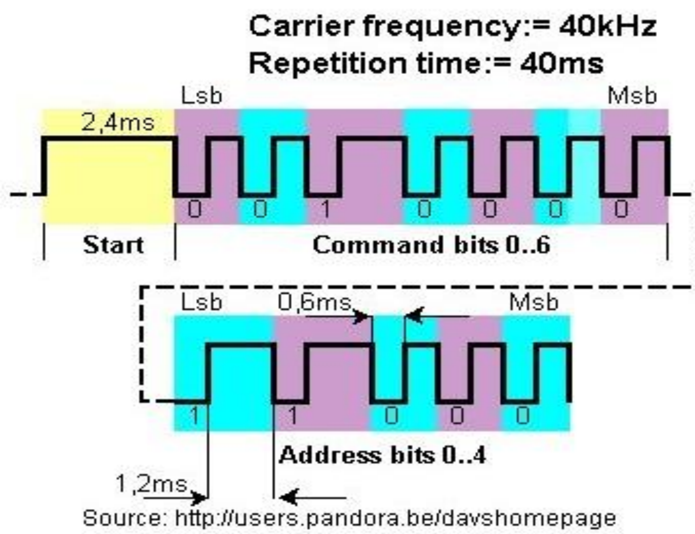
After extensive debate with myself and others in my electronics classes, I have concluded that the best utilization of my time spent towards finishing my capstone will be towards a project dedicated to infrared communications. The idea will be to allow a tank bot to be remote controlled via a universal remote that utilizes infrared signaling. The technology of transmitting data through light signals dates back to the dawn of the first shipmasters navigating treacherous seas. These shipmasters used lanterns to make different patterns of light that could be visible to other ships creating a form of communication. Obviously, the technology has made some tremendous leaps in the past century, however the concept remains the same. The transmission of data is orchestrated around the sending and receiving of ones and zeros, just like in times of old.

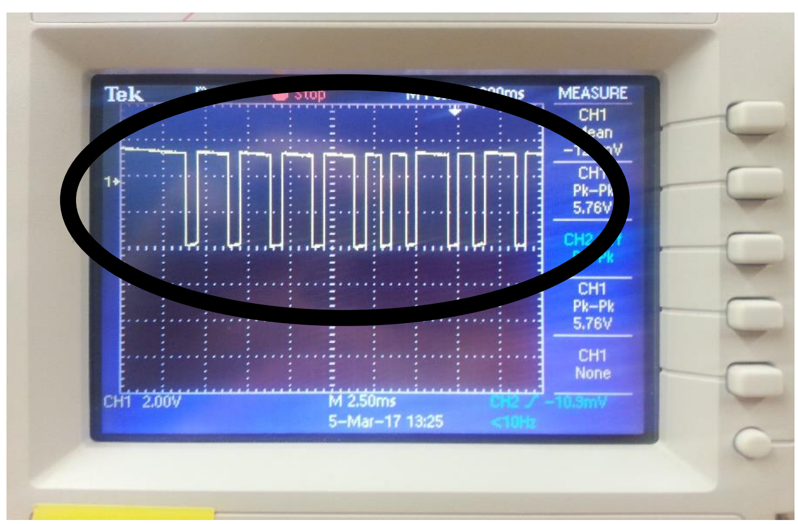
Infrared technology, of which I will refer to as simply IR, has played a pivotal role in modern society being implemented in technologies ranging from the satellites that orbit earth monitoring temperature, to the small universal remote we use to control the television. I will however, be focusing on the latter. The universal remote IR signals and communications.

What I hope to accomplish is to take a universal remote that uses IR light that sends a signal to a receiver that then will be converted to a specific input on the PIC16f886 microprocessor chip. After the data has been received, it will be used to drive a previously built tank robot (constructed in the microprocessor class EET 3780). The tank should drive both forwards and backwards as well as spin, both clockwise and counterclockwise.

The IR signal is generally broken up into three parts when transmitted: START, COMMAND and ADDRESS as illustrated in below. Infrared light has a range of wavelengths, just like visible light has wavelengths that range from red light to violet. "Near infrared" light is closest in wavelength to visible light and "far infrared" is closer to the microwave region of the electromagnetic spectrum. I relied heavily on picproject's pdf file *Sony SIRC infrared protocol* (located at the end of this proposal) to understand the behavior of the data stream being sent from the universal remote.

For the project, I will be using the GP1UW70QS series receiver to help me decode the GE Universal Remote signals being transmitted. When connected to the oscilloscope we see that there is pattern when the "channel up" button is pushed. I will be teaming up with Alec Turner taking his already constructed Makeblock IR Version Tankbot and integrating this technology into my project that should prove to be educational, challenging and entertaining.





Cost of Project / Bill of Materials

Item	Quantity	Reference	Part	Cost	Cost/unit
1	1	BT1	BATTERY	$10.00/8	$1.25
2	1	C3	.47uf	$1.00/20	$0.05
3	1	C4	22uf	*	$3.64
4	2	D1,D2	LED	13.99/100	$0.14
5	1	R2	180 Ohm Resistor	10.99/400	$0.03
6	5	R1,R3,R4,R5,R6,R7	10k Ohm Resistor	10.99/400	$0.03
7	1	SW1	SW DIP-4	6.69/10	$0.67
8	1	SW4	SW PUSHBUTTON-SPST-2	6.80/100	$0.07
9	1	S1	3sw_toggle	*	$2.00
10	1	U1	PIC16F886	*	Free Sample
11	1	U2	REG_LM7805	5.29/20	$0.26
12	1	U3	LCD_Serial	*	$4.99
13	1	U4	IR_Sensor	5.99/5	$1.20
14	1	U5	Header_10pin	5.49/10(40pin)	$0.55

Total: $14.88

Other Parts	
Breadboard	$5.00
single sided board	$10.00
universal remote	$7.00

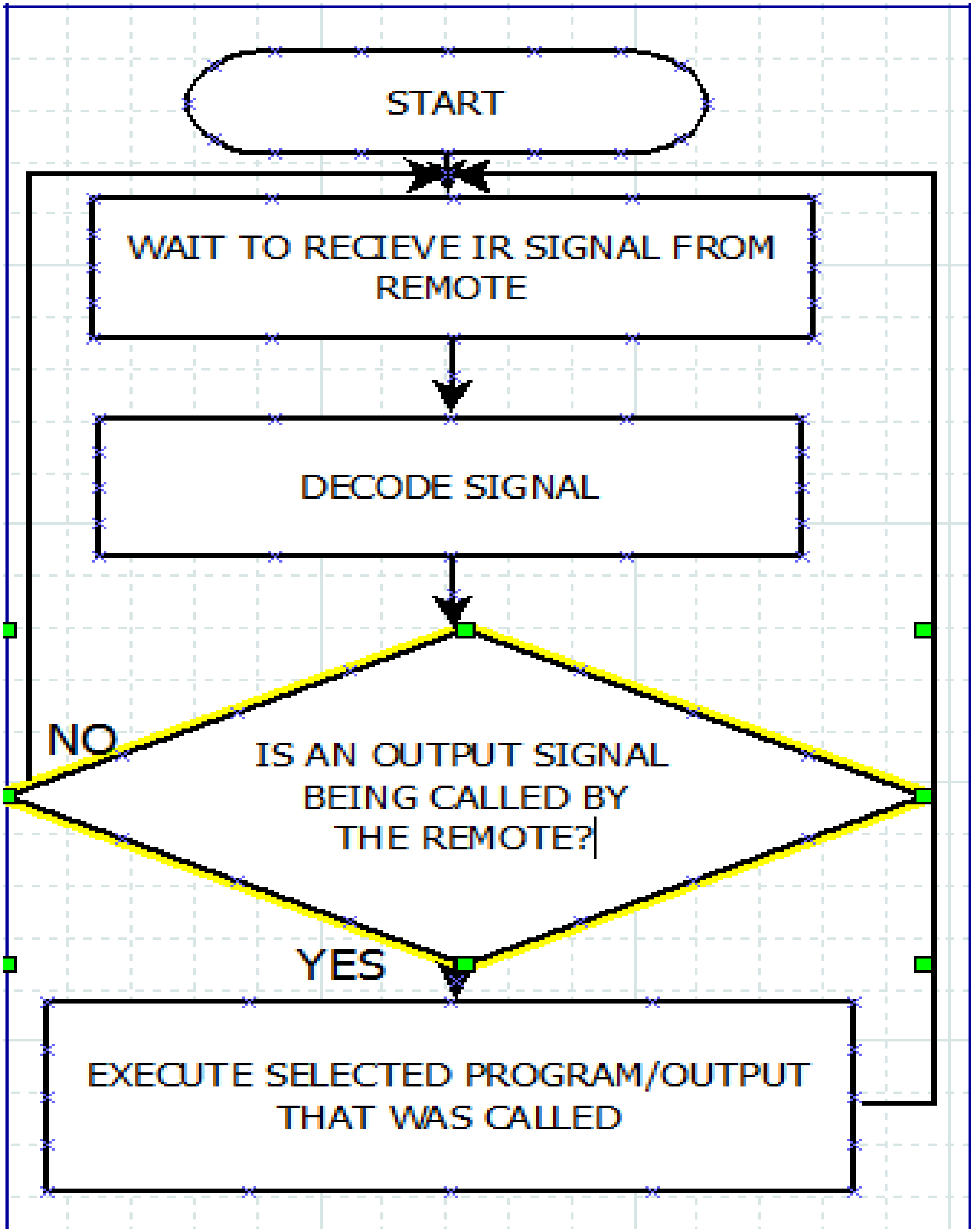
Total: $22.00

Grand Total = $36.88

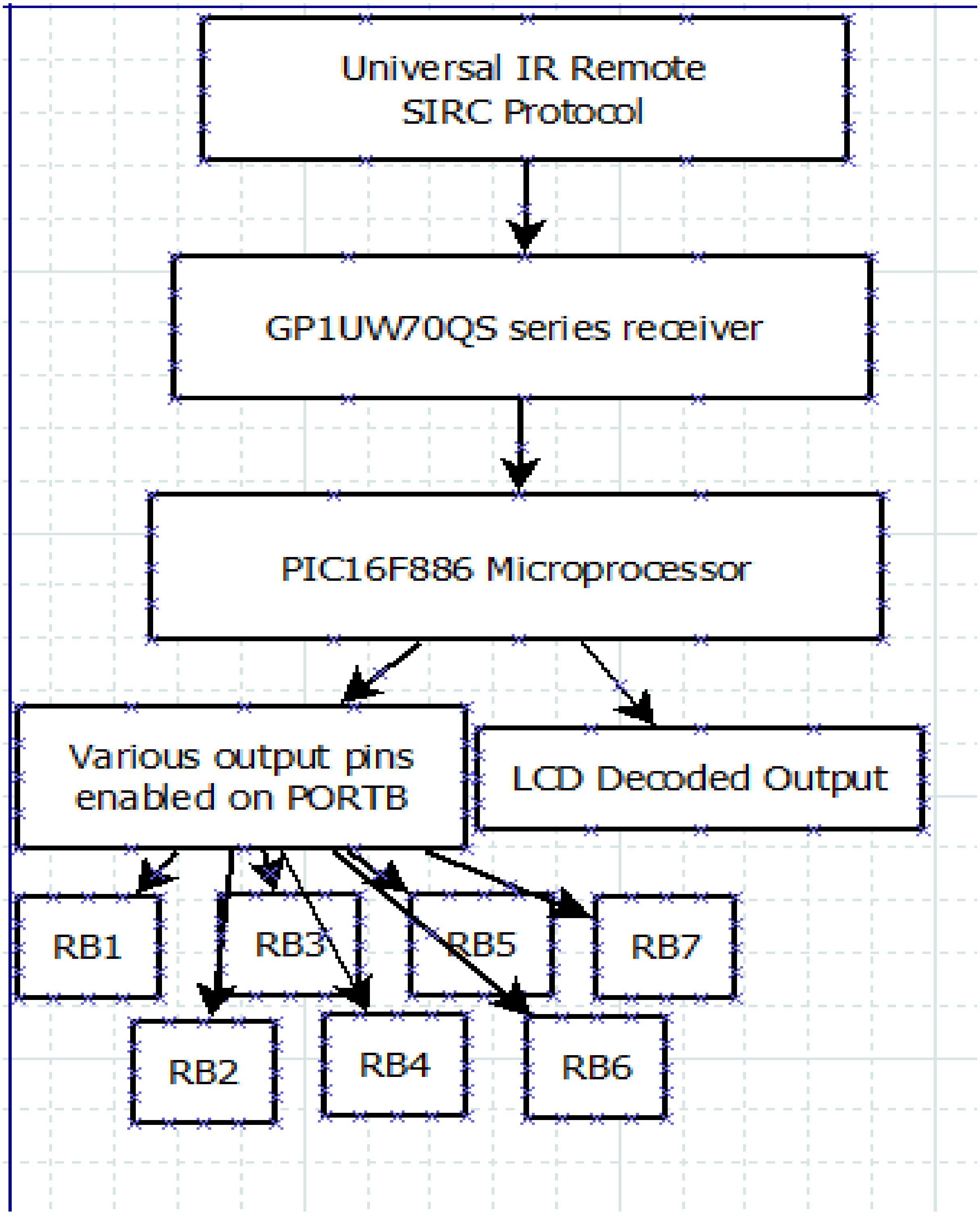
Time Line

Build Prototype board using a bread board and simple pushbuttons	January 13^{th} 2017
Decode the SIRC protocol found on my generic universal remote and map out the required push-buttons	February 3^{rd}
Write Code for the receiver to understand the input and enable the proper output channels using the XC8 compiler in MPLABX	February 15^{th}
Design a printed circuit board using OrCAD software that can be adapted to fit on a general-purpose bread board with the dimensions being approximately 2”x 2” x .25”	February 24^{th}
Attach design to toy robot tank prototype for testing and troubleshooting	March 3^{rd}
Prepare IR project for demonstration and grading	March 10^{th}
Present at Festival of Excellence	April 4^{th}

Flow Chart


START
WAIT TO RECIEVE IR SIGNAL FROM REMOTE
DECODE SIGNAL
IS AN OUTPUT SIGNAL BEING CALLED BY THE REMOTE?
NO
YES
EXECUTE SELECTED PROGRAM/OUTPUT THAT WAS CALLED

Block Diagram



Embedded C Code

This code was compiled using the XC8 Compiler using MPLABX software

```
#include <stdio.h>
#include <stdlib.h>
#include <xc.h>
#include <string.h>

#pragma config FOSC = HS // Oscillator Selection bits (RC oscillator: CLKOUT function on RA6/OSC2/CLKOUT pin, RC on RA7/OSC1/CLKIN)
#pragma config WDTE = OFF       // Watchdog Timer Enable bit (WDT disabled)
#pragma config PWRTE = OFF      // Power-up Timer Enable bit (PWRT disabled)
#pragma config MCLRE = ON       // RA5/MCLR pin function select (RA5/MCLR pin function is digital input, MCLR internally tied to VDD)
#pragma config BOREN = OFF      // Brown-out Reset Enable bit (BOD Reset disabled)
#pragma config LVP = OFF        // Low-Voltage Programming Enable bit (RB4/PGM pin has digital I/O function, HV on MCLR must be used for programming)
#pragma config CPD = OFF        // Data Code Protection bit (Data memory code protection off)
#pragma config CP = OFF         // Code Protection bits (Program memory code protection off)

//crystal frequency
#define _XTAL_FREQ 8000000
int resultH = 0;
int resultM = 0;
int resultL = 0;
int toggle = 0;

/****************************************************************************
 *                    MAIN ROUTINE
 * **************************************************************************
 * **************************************************************************
*/
void main (void){

   init ();
    __delay_ms (3500); //allow for SUU LCD to boot up with splash screen
   while(1){        //Main Home Screen
      printf("IR Decoder")    ;
      __delay_ms(200);
      clear_lcd();
   }
}
/****************************************************************************
 *                    INITIALIZATION ROUTINE
 * **************************************************************************
*/
int init()//initialize pins and ports to correct configuration
```

```
{
   ANSEL = 0x00;
   ANSELH = 0x00;

   //I/O set up for UART pins for PIC16F886 AND 887
   TRISA = 0b10100011;
   TRISB = 0b00000001;
   TRISC = 0b01101111;
   RA6 = 1;              //RED Power light
   RC4 = 0;
   GIE = 1;          //Global interrupt bit
   INTE = 1;         //enables Interrupt RB0
   INTF = 0;
   T0CS = 0;         //timer 0 clock source set to Internal instruction cycle clock (FOSC/4)
   T0SE = 0;         // TMR0 Source Edge Select bit 0 = Increment on low-to-high transition on T0CKI pin
   PSA = 1;      // Prescaler Assignment bit turned to a 1 for a 1:1 Ratio for TIMER0
   RBIE = 1;     //Port B interrupt enabled
   IOCB0 = 1;    //Interrupt on Change Bit enabled
   PEIE = 1;     //Periferial interrupt enabled on page 221 of data sheet
     //SETS THE  Internal Oscillator Frequency TO 8 MHz instead of the default 4MHz
   IRCF2 = 1;
   IRCF1 = 1;    // this is the max frequency the chip can handle without going to a external oscillator
   IRCF0 = 1;    // like a 20 KHz crystal

    //Setting up baudrate and RX configuration
   SPBRG = 12;  //9600 baud rate with 0.16% error rate
   BRGH  = 0;  //High baud rate generator
   BRG16 = 0;  //16-BIT baud rate generator
   SYNC  = 0;  //Eusart asynchronous mode
   SPEN  = 1;  //Serial port enable
   SREN  = 0;  //Single receive enable bit
   CREN  = 1;  //Continuous receive enable bit
   TXEN  = 1;
}
void detectCode(int data[])//writes decoded string onto lcd
{
   clear_lcd();
   top_line();
  for(int i = 0; i < 24; i++)
   {
      if(data[i] == 0)
         printf("0");
      else
         printf("1");
      if(i==9) next_line();
  }
```

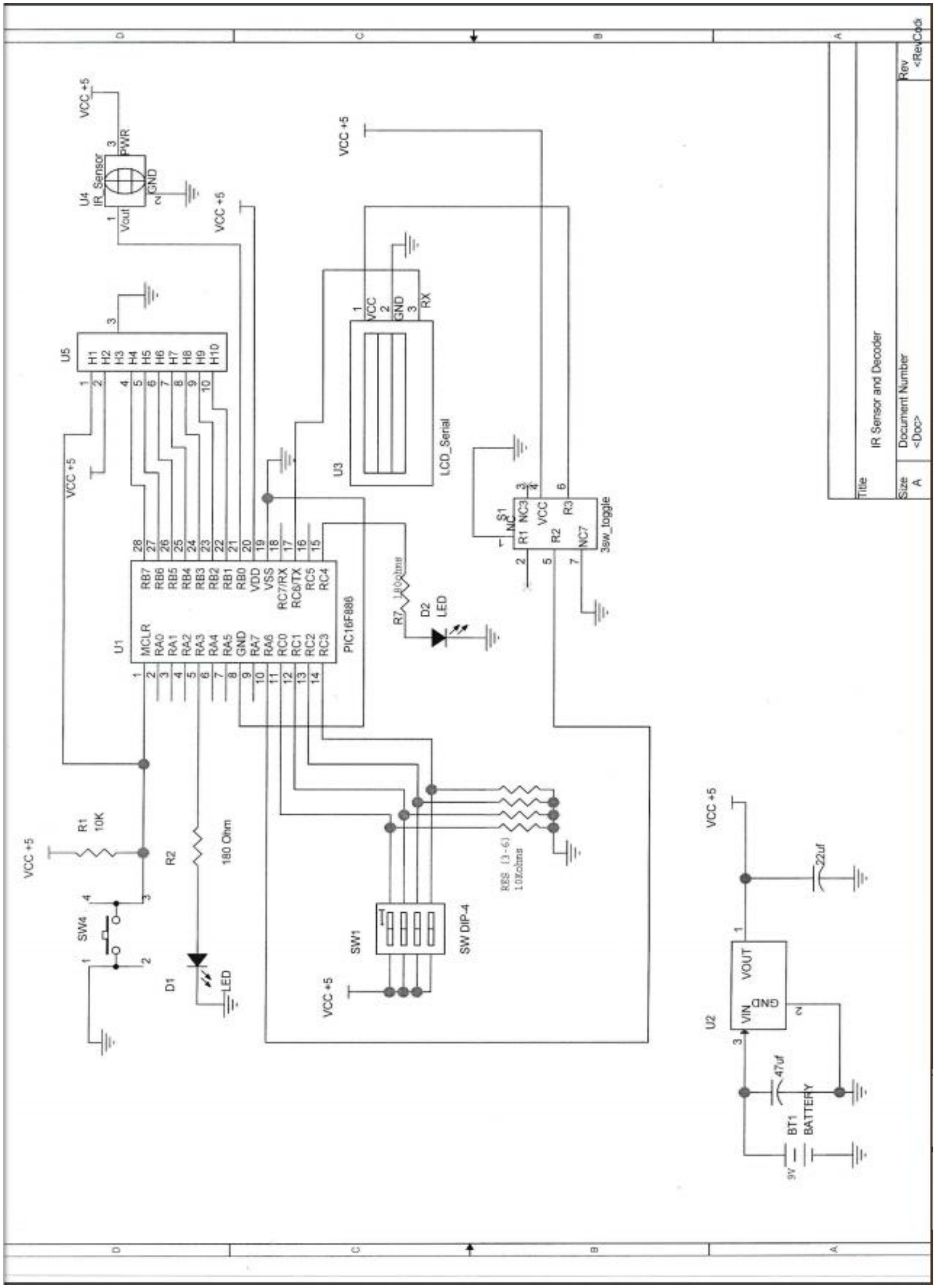
```
//*******************************************************************
            //REMOTE CONTROLS CONFIGURATION
//*******************************************************************
__delay_ms(1000);
clear_lcd();
resultH = (data[0]<<9 | data[1]<<8 | data[2]<<7 | data[3]<<6 | data[4]<<5 | data[5]<<4 | data[6]<<3 | data[7]<<2 | data[8]<<1 | data[9] );
resultM = (data[10]<<9 | data[11]<<8 | data[12]<<7 | data[13]<<6 | data[14]<<5 | data[15]<<4 | data[16]<<3 | data[17]<<2 | data[18]<<1 | data[19]<<0 | data[20] );//| data[21]<<3 | data[22]<<2 | data[23]<<1 | data[24]);
resultL = (data [20]<<3 | data[21]<<2 |data[22]<<1 | data[23]);
int addr = ((resultM <<4) | resultL);

printf("addr:%x",addr);
 next_line();
 printf("cmd:%x",resultH);

 if (resultH == 0x3cb && addr == 0x10d2){       //UP CHANNEL
    next_line();
    printf("cmd:%x    up ", resultH);
    }

 if (resultH == 0x3cb && addr == 0xd3){         //DOWN CHANNEL
    next_line();
    printf("cmd:%x    down ", resultH);
    }

 if (resultH == 0x3cb && addr == 0x30d0){       //VOLUME UP (RIGHT ARROW)
    next_line();
    printf("cmd:%x   right ", resultH);
    }
 if (resultH == 0x3cb && addr == 0x20d1){       //VOLUME DOWN (LEFT ARROW)
    next_line();
    printf("cmd:%x   left ", resultH);
    }
 if (resultH == 0x3cc && addr == 0x10de){       //BUTTON ONE
    next_line();
    printf("cmd:%x     B1 ", resultH);
    }
 if (resultH == 0x3cc && addr == 0x20dd){       //BUTTON TWO
    next_line();
    printf("cmd:%x     B2", resultH);
    }
 if (resultH == 0x3cc && addr == 0x30dc){       //BUTTON THREE
    next_line();
    printf("cmd:%x     B3", resultH);
    }
```

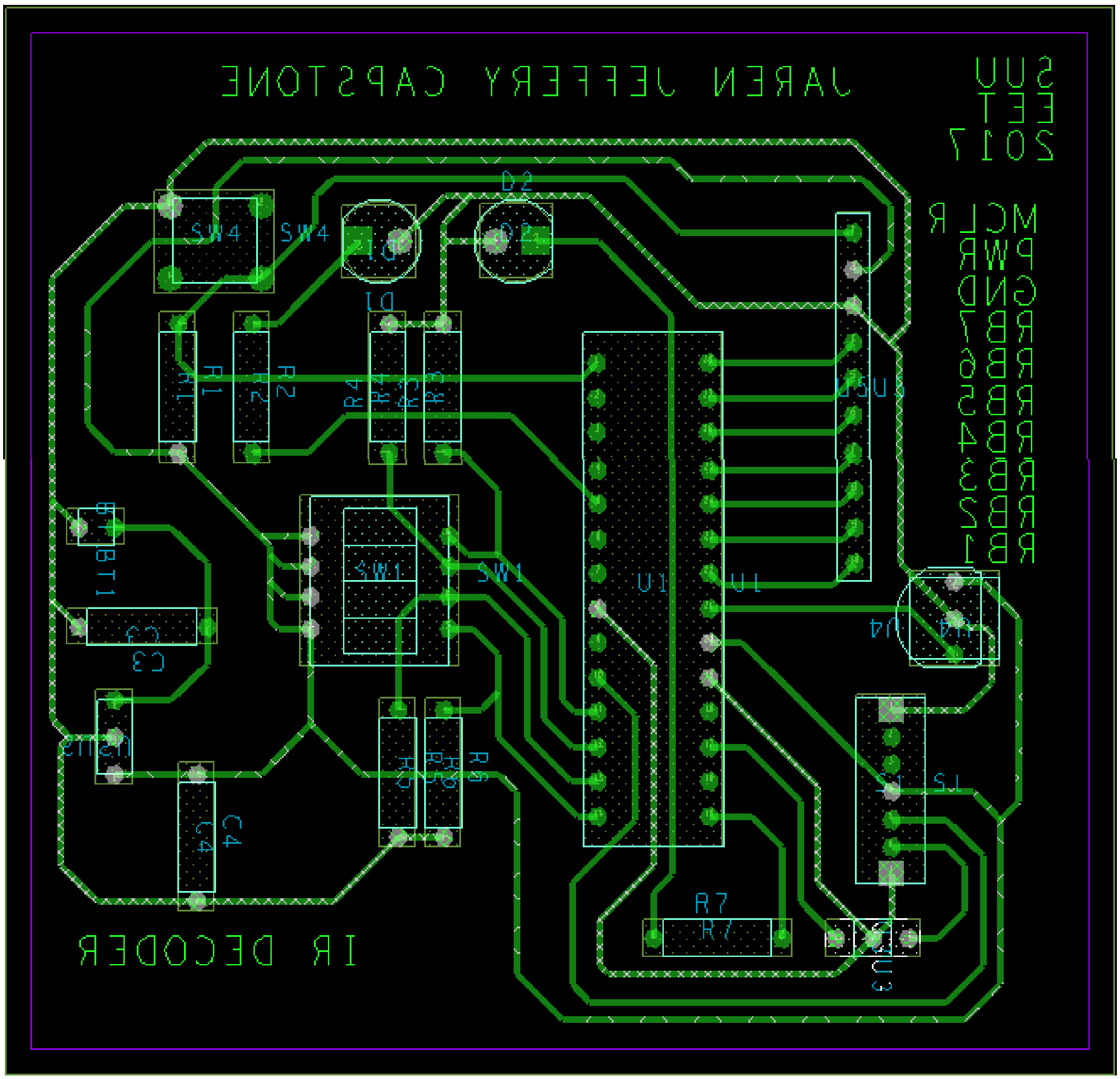
```
if (resultH == 0x385 && addr == 0x11fa){       //PLAY BUTTON
    next_line();
    printf("cmd:%x   PLAY", resultH);
    }
if (resultH == 0x387 && addr == 0x31e0){       //STOP BUTTON
    next_line();
    printf("cmd:%x   STOP", resultH);
    }
if (resultH == 0x3cd && addr == 0xdb){       //BUTTON FOUR
    next_line();
    printf("cmd:%x   B4", resultH);
    }
if (resultH == 0x3cd && addr == 0x10da){       //BUTTON FIVE
    next_line();
    printf("cmd:%x   B5", resultH);
    }
if (resultH == 0x3cd && addr == 0x20d9){       //BUTTON SIX
    next_line();
    printf("cmd:%x   B6", resultH);
    }
if (resultH == 0x3cd && addr == 0x30d8){       //BUTTON SEVEN
    next_line();
    printf("cmd:%x   B7", resultH);
    }
if (resultH == 0x3ce && addr == 0xc7){       //BUTTON EIGHT
    next_line();
    printf("cmd:%x   B8", resultH);
    }
 if (resultH == 0x3ce && addr == 0x10c6){       //BUTTON NINE
    next_line();
    printf("cmd:%x   B9", resultH);
    }
 if (resultH == 0x3cc && addr == 0xdf){       //BUTTON ZERO
    next_line();
    printf("cmd:%x   B0", resultH);
    __delay_ms(250);
    clear_lcd();
    printf("BOOM BABY");
    RB2 = 1;
    __delay_ms(500);
    RB2 = 0;
    }

//////////////////////////////////////////////////////////////////////////
//POWER BUTTON
//////////////////////////////////////////////////////////////////////////
if (resultH == 0x3ca && addr == 0x20d5){       //POWER BUTTON
    next_line();
```

```
    printf("cmd:%x  POWER", resultH);   //Print out the command portion in hexadecimal
    clear_lcd();

    if (toggle==0) {
      RC4 =1;
      toggle = 1;
      printf("cmd:%x POWER ON", resultH);
      addr = 0;
    }
    else{
      RC4 = 0;
      toggle = 0;
      printf("cmd:%x PWR OFF", resultH);
      }
  }
}

int data[24] = 0;     //holds data while code is being decoded
int starting_bit = 0;        //holds whether decoding has starting_bit
int bits_recieved = 0;  //bits that have been received
int time_count = 0;   //the current time since the last bit was received

/*************************************************************************
 *              INTERRUPT ROUTINE & DECODING PROCESSES
 * **********************************************************************
 * * *********************************************************************
*/

void interrupt interruptRoutine()//handles interrupts
{

  if (RBIF)         //on positive change of RB0 'raises' the flag, set in the Init()
  {
   if(RB0)              //on ir signal received
    {
      T0IE = 1;          //turn on counting
      TMR0 = 255;        //preset counter
      time_count = 0;   //reset time
    }
    else                 //on falling edge
    {
      T0IE = 0;          //disable timer

      if(!starting_bit)        //wait for initial starting code which has duration roughly 80-90 counts
      {
        if(time_count >= 1 && time_count <= 100)
        {
          starting_bit = 1;
```

```
         }
         else
         {
            starting_bit = 0;  //if count exceeds 100 then everything resets back to zero
            bits_recieved = 0;  //and awaits for a new IR code to be sent
         }
      }
      else //decode a bit in the series of bits
      {
         if(time_count >0 && time_count <22)  // counts to see if bit [i] will be a zero or a one
         {
            data[bits_recieved] = 0;
         }
         else if(time_count >=23 && time_count <= 42)
         {
            data[bits_recieved] = 1;
         }
         else
         {
            starting_bit = 0;  //set for timeout so that if there is an error it will reset bits and count
            bits_recieved = 0;
         }

         if(bits_recieved == 23)//when code is fully decoded print result
         {
            detectCode(data);
            starting_bit = 0;
            bits_recieved = 0;
         }
         else
         {
           bits_recieved++;//increment bit count

         }
      }
   }

   RBIF = 0;                  //reset interrupt flag
}
else if(T0IF)
{
   time_count = time_count + 1;       //increase clock time
   TMR0 = 179;                       //preset timer will give me 50 uS
   T0IF = 0;                        //reset interrupt flag

if(time_count>150)                   //abort after timeout
   {
      T0IE = 0;             //turn off timer interrupt
```

```
        starting_bit = 0;     //reset 'starting_bit' to zero
        bits_recieved = 0;     //back to 0 bits
    }
  }
}
/***************************************************************************
 *              STANDARD LCD SCREEN COMMANDS AND CONTROLS
 * ***********************************************************************
 * * ***********************************************************************
*/
clear_lcd()
  {
  putchar (0xFE);
  putchar (0x01);
  __delay_ms (10);
  }
void putch (unsigned char byte)
{
  while (!TRMT); //Needed to make 'printf' statement work
  TXREG = byte;   //wait until TX buffer is empty
  }
next_line ()
{
  //SUU LCD
  putchar (0xFE);
  putchar (0xC0); //codes to locate cursor at position 0 on line 2

}
top_line ()
{
  //SUU LCD
  putchar (0xFE);
  putchar (0x80); //codes to locate cursor at position 0 on line 1

}
```


IR Sensor and Decoder
Title
Size
A
Document Number
<Doc>
Rev
<RevCode>
U1
PIC16F886
MCLR
RA0
RA1
RA2
RA3
RA4
RA5
GND
RA7
RA6
RC0
RC1
RC2
RC3
RB7
RB6
RB5
RB4
RB3
RB2
RB1
RB0
VDD
VSS
RC7/RX
RC6/TX
RC5
RC4
U4
IR_Sensor
Vout
GND
PWR
U5
H1
H2
H3
H4
H5
H6
H7
H8
H9
H10
U3
LCD_Serial
VCC
GND
RX
S1
3sw_toggle
R1
NC
NC3
VCC
R2
R3
NC7
R7
100ohms
D2
LED
R1
10K
R2
180 Ohm
D1
LED
SW4
SW1
SW DIP-4
RES (3-6)
10Kohms
U2
VIN
GND
VOUT
22uf
47uf
BT1
BATTERY
9V
VCC +5


JAREN JEFFERY CAPSTONE
SUU
EET
2017
MCLR
PWR
GND
RB7
RB6
RB5
RB4
RB3
RB2
RB1
IR DECODER
SW4
D1
D2
R1
R2
R4
SW1
U1
U4
J2
R7
U3
C3
C4
BT1

Timers: Timer0 Tutorial (Part 2)

DS51702A

- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

Preface

NOTICE TO CUSTOMERS

All documentation becomes dated, and this manual is no exception. Microchip tools and documentation are constantly evolving to meet customer needs, so some actual dialogs and/or tool descriptions may differ from those in this document. Please refer to our web site (www.microchip.com) to obtain the latest documentation available.

Documents are identified with a "DS" number. This number is located on the bottom of each page, in front of the page number. The numbering convention for the DS number is "DSXXXXXA", where "XXXXX" is the document number and "A" is the revision level of the document.

INTRODUCTION

This chapter contains general information that will be useful to know before using the Timers Tutorial. Items discussed in this chapter include:

- Document Layout
- The Microchip Web Site
- Customer Support
- Document Revision History

DOCUMENT LAYOUT

This document provides an introduction to Timer0.

THE MICROCHIP WEB SITE

Microchip provides online support via our web site at www.microchip.com. This web site is used as a means to make files and information easily available to customers. Accessible by using your favorite Internet browser, the web site contains the following information:

- **Product Support** – Data sheets and errata, application notes and sample programs, design resources, user's guides and hardware support documents, latest software releases and archived software
- **General Technical Support** – Frequently Asked Questions (FAQs), technical support requests, online discussion groups, Microchip consultant program member listing
- **Business of Microchip** – Product selector and ordering guides, latest Microchip press releases, listing of seminars and events, listings of Microchip sales offices, distributors and factory representatives

CUSTOMER SUPPORT

Users of Microchip products can receive assistance through several channels:

- Distributor or Representative
- Local Sales Office
- Field Application Engineer (FAE)
- Technical Support
- Development Systems Information Line

Customers should contact their distributor, representative or field application engineer (FAE) for support. Local sales offices are also available to help customers. A listing of sales offices and locations is included in the back of this document.

Technical support is available through the web site at: http://support.microchip.com

DOCUMENT REVISION HISTORY

Revision A (January 2008)

- Initial Release of this Document.

MICROCHIP

TIMERS TUTORIAL

Timers: Timer0 Tutorial (Part 2)

OBJECTIVES

At the end of this lab you should be able to:

1. Develop application firmware to generate TMR0 overflow interrupts for specified time periods.
2. Develop application firmware using an external clock source with the Timer0 module.
3. Develop external Timer0 clock source applications that meet PIC16F690 Electrical Specifications.

PREREQUISITES

In order to successfully complete this lab you should:

1. Understand basic circuit theory.
2. Understand basic digital electronic components such as gates, multiplexers and memory registers.
3. Understand binary numbering systems and basic binary arithmetic.
4. Have some programming experience in the C Language.
5. Have completed the "*Introduction to MPLAB® IDE/PICC-Lite™ Compiler Tutorial*" (DS41322).
6. Have completed "*Timers: Timer0 Tutorial (Part1)*" (DS51682).

EQUIPMENT REQUIRED

This lab has been developed so that no hardware is required other than a PC. However, you will need the following:

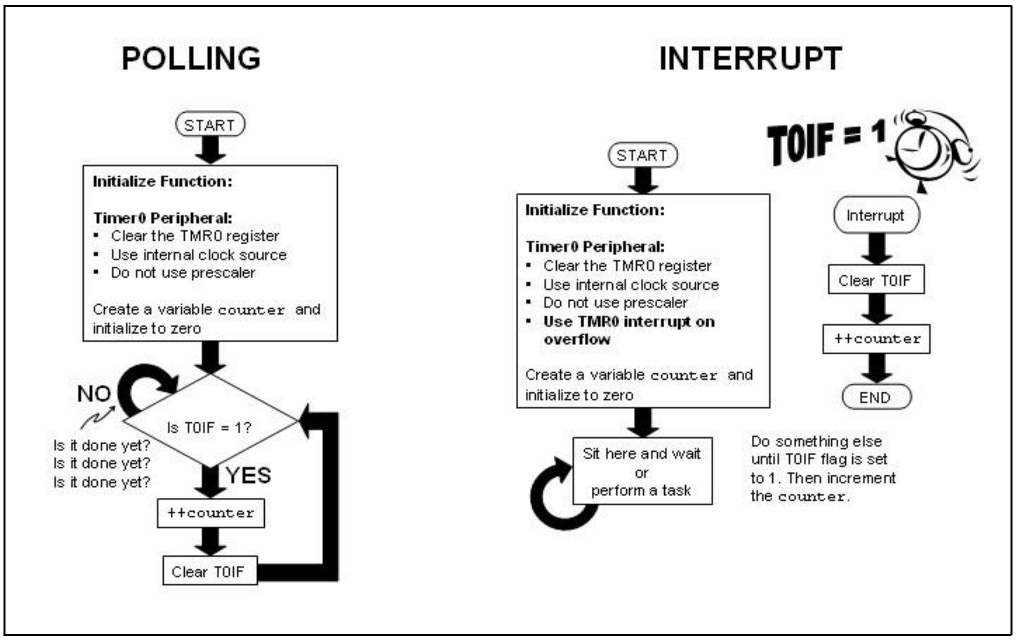
1. You will need to download the free MPLAB Integrated Development Environment available at the following url: http://www.microchip.com

   When prompted, unzip the contents of the file into a temporary folder on your desktop and then install.
2. Install the free HI-TECH PICC-LITE™ compiler (refer to the download instructions).
3. Once both programs are installed, complete the "*Introduction to MPLAB® IDE/PICC-Lite™ Compiler Tutorial*" (DS41322) if you haven't already. This lab assumes that you have done so and will expand on that knowledge.
4. It is also recommended that you download a copy of the PIC16F690 data sheet (DS41262) from www.microchip.com.

TIMER0 INTERRUPT

In the previous lab we incremented a variable, `counter`, whenever the Timer0 value register TMR0 overflowed from 255 to 0. To do this, a "Polling" algorithm was used where the Timer0 Interrupt Flag (T0IF) was checked periodically to see if it was set to '1'. This indicated that the TMR0 register had overflowed and that the `counter` variable should be incremented. Now, you may notice that this type of algorithm of periodically checking the T0IF ties up the processor for however long it takes to perform the check. This may be acceptable for some applications, however, there will be times when you would like the processor to devote its attention to a different task and only take care of, or "service", incrementing the `counter` variable when the T0IF flag overflows without needing to constantly check its status. This is easily accomplished on mid-range PIC® microcontrollers, such as the PIC16F690, using interrupts which serve as an alarm, signaling that a particular event has occurred (such as when the T0IF flag is set). In Figure 1-1, the left-hand flowchart represents the polling algorithm used in the previous lab while the right-hand flowchart represents an alternative approach in the form of an interrupt routine.

FIGURE 1-1: POLLING AND INTERRUPT ALGORITHMS TO INCREMENT COUNTER VARIABLE



The PIC16F690 can be configured to perform a specific task when an interrupt occurs. This is called the Interrupt Service Routine or "ISR" for short. When any interrupt occurs, and there could be more than one, the processor will immediately stop what it is doing and jump to the ISR to service the interrupt. Once completed, the processor returns to what it was doing in code, prior to being interrupted. If multiple peripheral interrupts are used, a prioritization algorithm will need to be included in the ISR to determine which interrupt is serviced first. This lab will concentrate on Timer0 interrupts only and not introduce any others.

CONFIGURING TIMER0 INTERRUPTS

The PIC16F690, as with any other PIC mid-range microcontroller, can be configured to generate an interrupt when the TMR0 register overflows from 255 to 0 (11111111_2 to 00000000_2). To accomplish this, we must utilize the Interrupt Control (INTCON) register. Figure 1-3 shows the INTCON register with the bits used in this tutorial.

FIGURE 1-2: INTERRUPT CONTROL REGISTER (INTCON)

INTCON

7	6	5	4	3	2	1	0
GIE	PEIE	T0IE	INTE	RABIE	T0IF	INTF	RABIF

Not used in this Tutorial

GIE: Global Interrupt Enable bit

T0IE: Timer0 Overflow Interrupt Enable bit

T0IF: Timer0 Overflow Interrupt Flag bit

There are basically three primary Configuration bits used to configure any interrupt. First, the Global Interrupt Enable bit (GIE) acts as a sort of "Master Switch" that must be set to enable interrupt capability on the PIC mid-range microcontroller. The GIE will automatically clear to '0' whenever an interrupt occurs, ensuring that no other interrupts can occur during execution of the ISR. Therefore, once the ISR is completed, the GIE must be set again to enable future interrupts. Next, each peripheral will have individual interrupt enable bits. These individual interrupt enable bits may be contained within a separate Peripheral Interrupt Register (PIRx). However, the Timer0 peripheral interrupt enable bit is contained within the INTCON register. The Timer0 Overflow Interrupt Flag bit (T0IF) is set, and remains set until cleared in software, when a Timer0 overflow has occurred. This bit needs to be cleared if further interrupts are required for this peripheral. The following recommended sequence should be used when configuring any interrupt and following any ISR:

1. Clear the interrupt flag (Timer0 Overflow Interrupt Flag).
2. Enable the individual peripheral interrupt (set the Timer0 Overflow Interrupt Enable bit).
3. Enable PIC mid-range MCU interrupt capability by setting the Global Interrupt Enable bit.

Interrupt configuration using these steps will ensure that interrupts do not occur during initialization, causing unexpected results.

HANDS-ON LAB 1: TIMER0 INTERRUPTS

Purpose:

In this lab, a counter variable will increment each time the TMR0 register overflows from 255 to 0. To accomplish this, we will configure INTCON so that an interrupt occurs whenever the T0IF (TMR0 Overflow Interrupt Flag) is set, indicating an overflow. To implement an interrupt using the PICC-LITE compiler, the interrupt function qualifier must be used followed by the chosen name of the Interrupt Service Routine (refer to Example 1-1).

EXAMPLE 1-1: TIMER0_ISR

```
void interrupt Timer0_ISR(void)
{ if (T0IE && T0IF) //are TMR0 interrupts enabled and //is
        the TMR0 interrupt flag set?
        {
        T0IF=0;             //TMR0 interrupt flag must be cleared in software
                            //to allow subsequent interrupts
        ++counter;          //increment the counter variable by 1
        }
}
```

PROCEDURE

Part 1: Configuring Timer0 Interrupts

1. Create a new project in MPLAB IDE using the following:
   a) Select the PIC16F690 as the device.
   b) Select HI-TECH PICC-LITE™ as the Language Toolsuite.
   c) Create a folder on your C:\ drive and store the project there.
2. In the MPLAB IDE workspace, create a new file and copy the code in Example 1-2 into it.

EXAMPLE 1-2: HANDS-ON LAB CODE

```
#include <pic.h>


 //Configure device
           __CONFIG(INTIO & WDTDIS & PWRTDIS & MCLRDIS &
                     UNPROTECT & BORDIS & IESODIS & FCMDIS);



//-----------------------DATA MEMORY

unsigned char counter;            //counter variable to count
                                  //the number of TMR0 overflows

//----------------------PROGRAM MEMORY

/*----------------------------------------------------------
           Subroutine: Timer0_ISR
           Parameters: none
           Returns:     nothing
           Synopsys: This is the Interrupt Service Routine for
Timer0 overflow interrupts. On TMR0 overflow the counter variable
is incremented by 1 -----------------------------------------------
-----------*/ void interrupt Timer0_ISR(void)
{
         if (T0IE && T0IF)   //are TMR0 interrupts enabled and //is
                                       the TMR0 interrupt flag set?
           {
           T0IF=0;                //TMR0 interrupt flag must be
                                  //cleared in software
                                  //to allow subsequent interrupts
           ++counter;               //increment the counter variable
                                  //by 1
           }
}
```

EXAMPLE 1-2: HANDS-ON LAB CODE (CONTINUED)

```
/*-----------------------------------------------------------
           Subroutine: INIT
           Parameters: none
           Returns:     nothing
           Synopsys:    Initializes all registers
                        associated with the application
-----------------------------------------------------------*/
Init(void)
{
           TMR0 = 0;              //Clear the TMR0 register

/*Configure Timer0 as follows:

           - Use the internal instruction clock   as
           the source to the module
           - Assign the Prescaler to the Watchdog
           Timer so that TMR0 increments at a 1:1
           ratio with the internal instruction
           clock*/

           OPTION = 0B00001000;
           T0IE = 1;                //enable TMR0 overflow interrupts
           GIE = 1;               //enable Global interrupts

}

/*-----------------------------------------------------------
           Subroutine: main
           Parameters: none
           Returns:     nothing
           Synopsys:    Main program function
-----------------------------------------------------------
*/ main(void) {
           Init();                //Initialize the relevant registers

           while(1)               //Loop forever
           {
           }
}
```

Save as a .C file.

3. Build the project by pressing the Build Project icon. There should be no errors.
4. Select the MPLAB SIM as the debugger.
5. Open a Watch window and add the TMR0, INTCON and OPTION_REG Special Function Registers.
6. Add the `counter` symbol. Configure the Watch window to allow binary, hexadecimal and decimal values to be seen.
7. Press the Animate icon in the Debugger toolbar and confirm that the following occurs:

   a) On a TMR0 overflow (255 ⇨ 0) the T0IF flag is set (INTCON<2>).

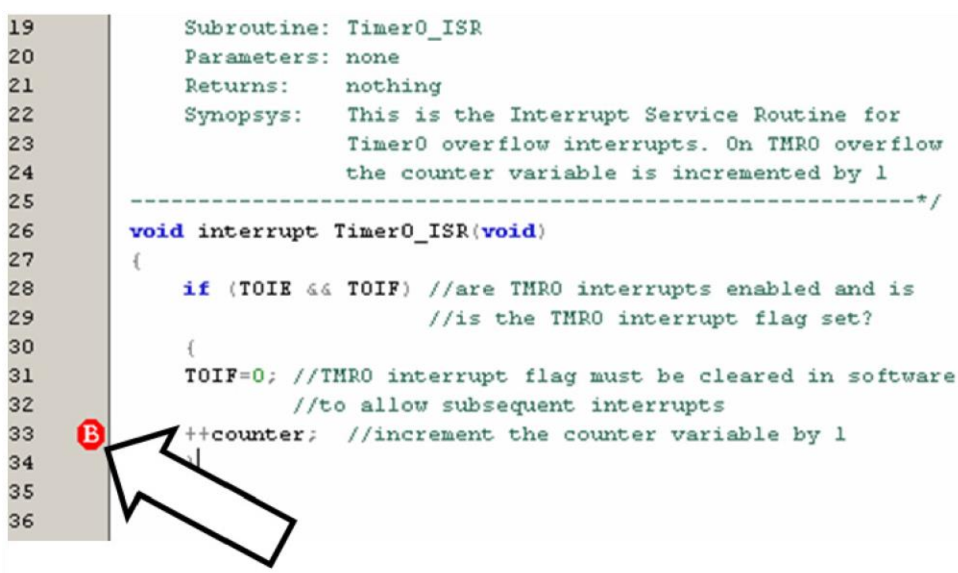
   b) Using the Watch window, confirm that when the TMR0 flag sets, the Timer0_ISR() interrupt routine is executed and the `counter` variable is incremented by one.

Part 2: Timing Analysis

Next, we will check to see how fast the `counter` variable is actually incrementing.

1. Open the Stopwatch by selecting *Debugger > Stopwatch*.
2. The simulator Processor Frequency automatically defaults to 20 MHz. This will need to be changed to the oscillator frequency used on this particular PIC microcontroller. To change the Processor Frequency select *Debugger>Settings* and change the Processor Frequency to 8 MHz (max. internal oscillator frequency on the PIC16F690) under the Osc/Trace tab.
3. Setup a breakpoint next to the line in the `interrupt Timer0_ISR()` subroutine that increments the `counter` variable (see Figure 1-3).

FIGURE 1-3: INTERRUPT CONTROL REGISTER (INTCON)



> **Note:** The specific line number in your code may differ from that shown.

Setting the breakpoint here will allow the Stopwatch to analyze the time interval between successive `counter` variable increments.

In the "*Timers: Timer0 Tutorial (Part1)*" (DS51682), an equation was introduced to determine the length of time for successive `counter` variable increments (see Equation 1-1).

EQUATION 1-1: DETERMINING INTERNAL INSTRUCTION CLOCK CYCLE PERIOD

Internal instruction cycle = 1 / [(Processor Frequency) / 4] = 1 / (8 MHz / 4) = 500nS

Since the TMR0 register is 8-bits wide, it will take 256 internal instruction cycles for an overflow to happen ($2^8 = 256$). Since the PIC16F690 is configured to run off of the 8 MHz internal oscillator, we can use Equation 1-2 to determine TMR0 overflow.

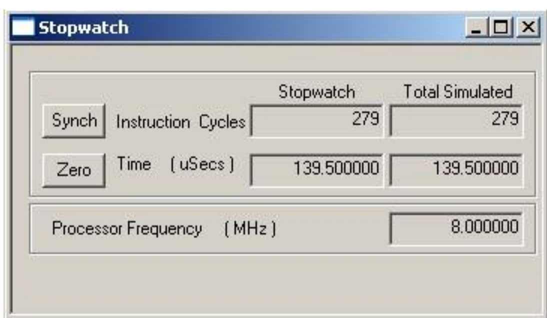
EQUATION 1-2: DETERMINING TMR0 OVERFLOW PERIOD

TMR0 overflow = Internal instruction cycle x 2^8 (we must count the zero) = 500nS x 256 = 128µS

On any TMR0 overflow, the Interrupt Service Routine (`Timer0_ISR()`) will execute by clearing T0IF and then increment the `counter` variable. Let's check to see if this is what happens.

4. Press the **Reset** button on the simulator toolbar.
5. Press the **Run** button to execute the program up until the breakpoint is encountered. The Stopwatch window should resemble Figure 1-4.

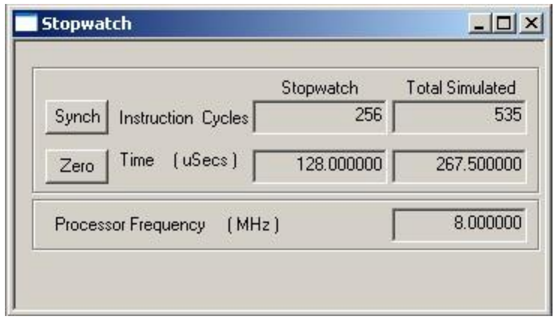
FIGURE 1-4: STOPWATCH WINDOW



Notice the Stopwatch indicates it took 139.5 µS to reach the breakpoint. This doesn't agree with Equation 1-2. However, don't forget that there is some extra code that the central processing unit (CPU) will need to execute before configuring the Timer0 peripheral such as device configuration, variable declarations and so on.

6. Press the **Zero** button in the Stopwatch window. This clears the Stopwatch to zero without resetting the CPU. Press the **Run** button once again in the simulator toolbar. The Stopwatch should now resemble Figure 1-5.

FIGURE 1-5: UPDATED STOPWATCH WINDOW



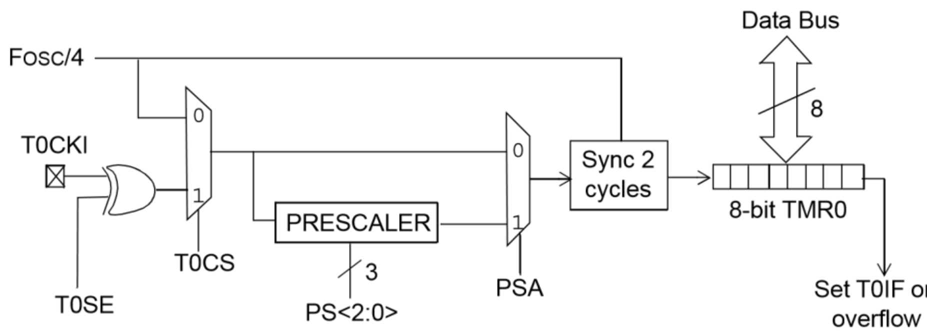
The Stopwatch should now indicate that it has taken precisely 128 μS to reach the breakpoint as per Equation 1-2. In *"Timers: Timer0 Tutorial (Part1)"* (DS51682), the time it took using polling instead of interrupts to increment the `counter` variable was close but not exactly what was calculated. Why do you think that is?

It can be concluded that in timing sensitive applications, it's a good idea to utilize TMR0 interrupts. In this way, if TMR0 overflows, the processor immediately stops whatever it is doing, services the interrupt (executes the interrupt subroutine) and then resumes its previous task.

USING AN EXTERNAL CLOCK SOURCE

When the topic of Timers was first introduced in *"Timers: Timer0 Tutorial (Part1)"* (DS51682), it was mentioned that these peripherals could be used as timers or counters. The only difference is how the module is used. Up until this point, the labs have focused on using Timer0 as a timer. In this section, Timer0 is used as a counter. PIC microcontrollers allow the use of an external source to drive the TMR0 register via connection to the Timer0 Clock Input pin (T0CKI) (refer to Figure 1-6 and Figure 1-7). This external source could be an oscillator or simply a pushbutton connected to the pin. Also notice in the block diagram that the T0CKI signal enters an XOR gate along with the Timer0 Source Edge Select bit (T0SE) from the OPTION register. This allows the TMR0 register value to increment on either the high-to-low (negative edge) or low-to-high (positive edge) transition of the signal on the T0CKI pin. Following the signal path through the block diagram, this signal can also be prescaled. Perhaps the application requires that the TMR0 register is incremented every 2^{nd} negative edge of the input signal or every 256^{th} edge of the positive going edge. Remember that in order to use this prescaler the PSA bit in the OPTION register needs to be cleared to zero. Also note that the input signal passes through a 2-cycle synchronization circuit to ensure synchronization with the PIC16F690 instruction clock.

FIGURE 1-6: SIMPLIFIED BLOCK DIAGRAM OF TIMER0 MODULE



BIT VALUE	TMR0 RATE
000	1:2
001	1:4
010	1:8
011	1:16
100	1:32
101	1:64
110	1:128
111	1:256

FIGURE 1-7: OPTION REGISTER SHOWING THE TIMER0 CLOCK SOURCE SELECT AND SOURCE EDGE SELECT BITS

OPTION

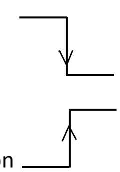
7	6	5	4	3	2	1	0
RABPU	INTEDG	**T0CS**	**T0SE**	**PSA**	**PS2**	**PS1**	**PS0**

T0CS: Timer0 Clock Source Select bit

1 = Transition on T0CKI pin

0 = Internal instruction cycle clock (Fosc/4)

T0SE: Timer0 Source Edge Select bit



1 = Increment on high-to-low transition

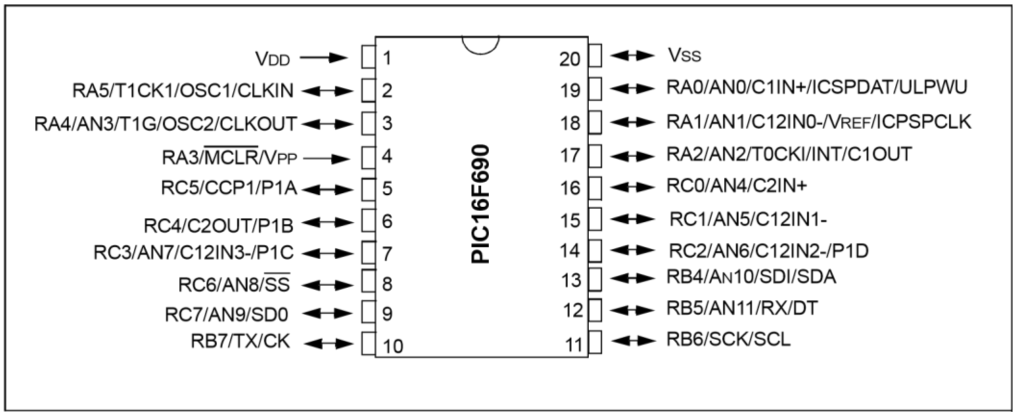
0 = Increment on high-to-low transition

Note: Configuring the Timer0 module using the OPTION register is discussed in greater detail in *"Timers: Timer0 Tutorial (Part1)"* (DS51682).

As shown in Figure 1-8, which shows the pin-out diagram for the PIC16F690, the T0CKI pin shares functionality as follows:

- RA2 represents the bit 2 position of the PORTA register
- INT represents an external interrupt pin
- C1OUT represents the output of the Comparator 1 module
- AN2 represents an input to the Analog-to-Digital converter module.

FIGURE 1-8: PIC16F690 PIN-OUT DIAGRAM



The AN2 feature of this pin means it can be used for either digital or analog signals. The PIC16F690, has been designed so that the analog pins (i.e., ANx) will default to analog when the PIC MCU powers up. Since this pin will be used for a digital signal, analog functionality is disabled using a Special Function Register called the Analog Select Register (ANSEL) as shown in Figure 1-9.

FIGURE 1-9: ANALOG SELECT REGISTER

ANSEL: ANALOG SELECT REGISTER

ANS7	ANS6	ANS5	ANS4	ANS3	ANS2	ANS1	ANS0
7	6	5	4	3	2	1	0

ANS<7:0>: Analog Select bits

Analog select between analog or digital function on pins AN<7:0> respectively

1 = Pin is configured as Analog input

0 = Pin is configured as Digital input

Referring to the pin-out diagram for the PIC16F690 in Figure 1-8, notice that there are actually 12 analog configurable pins (i.e., AN0 → AN11). The Analog Select High (ANSELH) register can be configured as well if needed for these pins. However, in this application, the only pin of interest is the T0CKI/AN2 pin. These other pins will be discussed in greater detail in other labs and as always, for more information on this or any other feature of this product, refer to the data sheet.

When using an external input signal of any kind, it is important to pay particular attention to electrical specifications and timing parameters listed in the data sheet. The 2-cycle synchronization block shown in Figure 1-6 samples the input signal on the T0CKI pin and synchronizes it with the clock used by PIC16F690. Therefore, there are some important equations to know when not using the prescaler for Timer0 (see Equation 1-3 and Equation 1-4):

EQUATION 1-3: MINIMUM HIGH PULSE WIDTH OF T0CKI SOURCE SIGNAL WITH NO PRESCALER

$$T_{T0H} = \left(\frac{2}{\text{PIC MCU OscillatorFrequency}}\right) + 20nS = \text{minimum HIGH T0CK1 signal pulse width}$$

Example:

If using the 8 MHz internal oscillator, use Equation 1-4 and Equation 1-5.

EQUATION 1-4: MINIMUM LOW PULSE WIDTH OF T0CKI SOURCE SIGNAL

$$T_{T0H} = \left(\frac{2}{8MHz}\right) + 20nS = \textit{a minimum HIGH pulse of 270nS}$$

EQUATION 1-5: MINIMUM LOW PULSE WIDTH OF T0CKI SOURCE SIGNAL WITH NO PRESCALER

$$T_{T0H} = \left(\frac{2}{\text{PIC MCU OscillatorFrequency}}\right) + 20nS = \text{minimum LOW T0CK1 signal pulse width}$$

Example:

If using the 8 MHz internal oscillator, the minimum low pulse width can be calculated as shown in Equation 1-6.

EQUATION 1-6: MINIMUM LOW PULSE WIDTH OF T0CKI SOURCE SIGNAL WITH 8 MHz INTERNAL OSCILLATOR

$$T_{T0H} = \left(\frac{2}{8MHz}\right) + 20nS = \textit{a minimum } \text{LOW } \textit{pulse } \text{width } \textit{of 270nS}$$

The internal sampling that occurs on the T0CKI signal takes two clock cycles of the PIC microcontrollers oscillator. Divide 2 by the oscillator frequency in Hz to obtain an answer in seconds (same as multiplying the oscillator frequency in seconds by 0.5). The 20 nS added at the end of the equations represents a small 20 nS RC delay present within the device. In this lab, the 8 MHz internal oscillator is used. Therefore, it is necessary to ensure that the incoming signal stays High and/or Low for a minimum of 270 nS when not using the prescaler. If the incoming signal is a TTL square wave, this means the period can be no less than 270nS + 270nS = 540nS or a frequency of (1/540nS) = 1.8 MHz

To use the prescaler on the T0CKI source signal, Equation 1-7 is used.

EQUATION 1-7: T0CKI SOURCE SIGNAL MINIMUM PERIOD

$$T_{0H} = 20\text{nS OR } \frac{\left(\frac{4}{PIC\ MCU\ \text{OSCILLATOR FREQUENCY } (Hz)}\right) + 40nS}{\textit{Prescale value (i.e. 2,4...256)}} \quad \textit{whichever is greater}$$

T_{0H} = minimum T0CKI signal period

Example:

If using the 8 MHz internal oscillator and a Timer0 prescale value of 64, the minimum T0CKI signal period is calculated as shown in Equation 1-8.

EQUATION 1-8: T0CKI SOURCE SIGNAL USING 8 MHz INTERNAL OSCILLATOR AND TIMER0 PRESCALE VALUE OF 0

$$T_{0H} = \frac{\left(\frac{4}{8MHz}\right) + 40nS}{64} = 8.4\text{nS } \textit{which is less than 20nS.}$$

Therefore, use a minimum period of 20 nS

In Equation 1-5, if the calculated value is less than 20 nS, a minimum period of 20 nS must be maintained. Otherwise, maintain a minimum period at the calculated value.

HANDS-ON LAB 2: USING AN EXTERNAL CLOCK SOURCE

Purpose:

In this lab, the counter variable will still increment each time the TMR0 register overflows from 255 to 0. This time, an external signal will be used as the Timer0 clock source and the TMR0 register configured to increment on the low-to-high transition of the signal. To simulate an external clock source the Stimulus feature of MPLAB SIM is used.

Procedure:

Part 1: Using MPLAB SIM Stimulus

1. Using either the project created in the previous lab or a new project, change the OPTION register configuration value in the `Init()` to allow

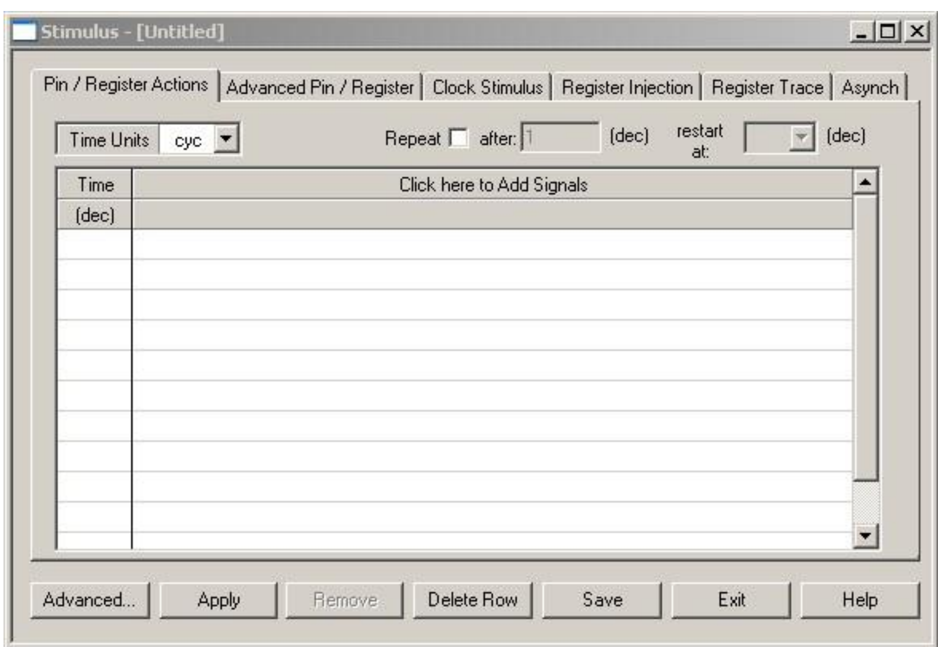
for the use of an external clock source and configure the ANSEL register so that the T0CKI pin is a digital input as shown in code Example 1-3.

EXAMPLE 1-3: CHANGES TO `INIT()`

```
Init(void)
{

    ANSEL = 0B111111011;//Configure T0CKI/AN2 as a digital I/O

    TMR0 = 0;//Clear the TMR0 register


/*Configure Timer0 as follows:

    -   Use the T0CKI pin and external source as
        the source to the module
    -   Increment the TMR0 register on the low-to-
        high transition of the external source
    -   Assign the Prescaler to the Watchdog Timer
        so that TMR0 increments at a 1:1 ratio
        with the internal instruction clock*/

    OPTION = 0B00101000;
    T0IE = 1;//enable TMR0 overflow interrupts
    GIE = 1; //enable Global interrupts

}
```

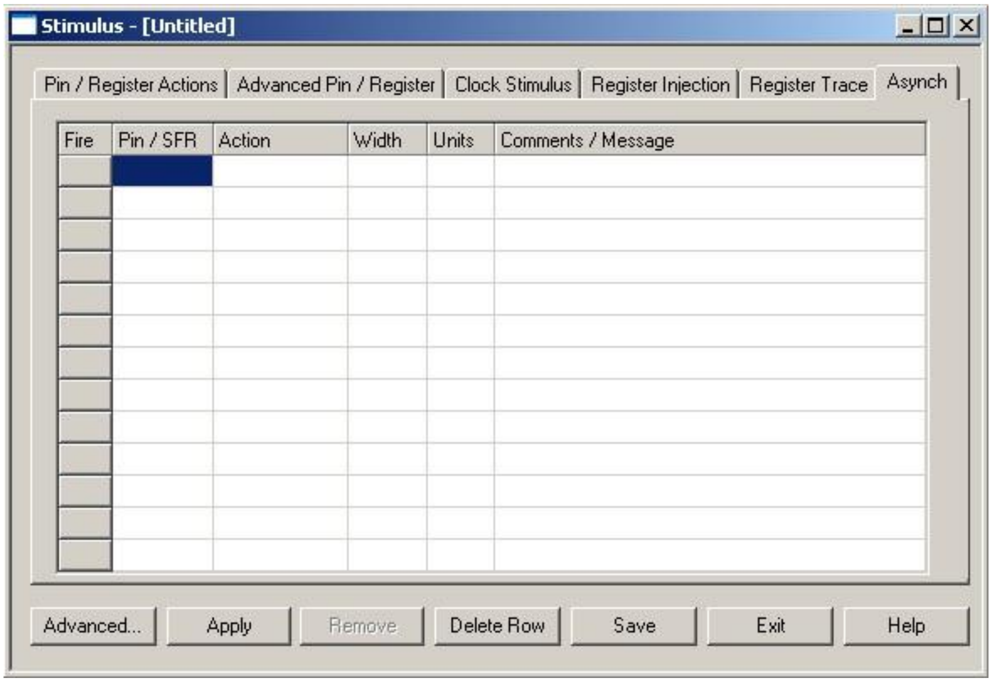
2. Re-compile the code and ensure that there are no errors.
3. Make sure that the MPLAB SIM simulator is selected as the debugger. Open the Stimulus Tool by selecting *Debugger>Stimulus>New Workbook*. The window in Figure 1-10 should now appear.

FIGURE 1-10: STIMULUS WORKBOOK WINDOW



4. Next, select the Asych tab in the Stimulus window. The Stimulus window should resemble Figure 1-11.

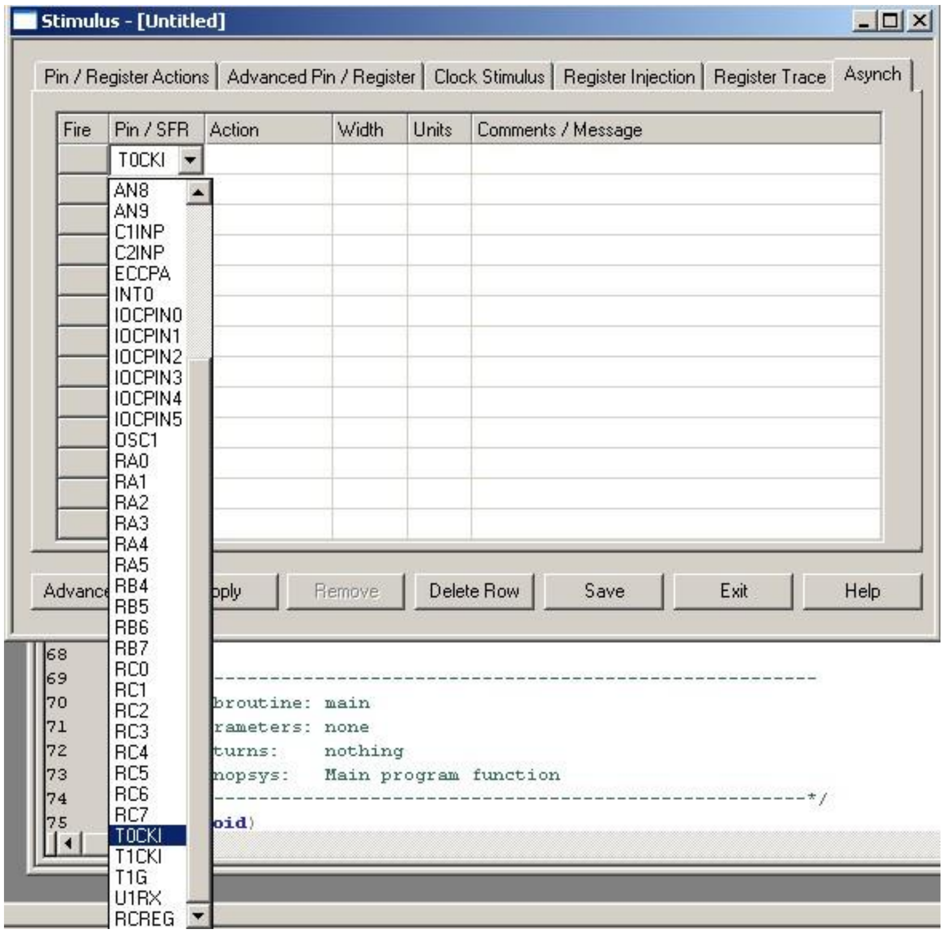
FIGURE 1-11: ASYNCH TAB IN STIMULUS WORKBOOK



Stimulus is a tool used to simulate a signal on a pin external to the PIC MCU or to actually generate a change to a bit in a peripheral's Special Function Register. This can be accomplished either synchronously by applying a predefined series of signal changes to an I/O pin, or asynchronously as we will use in this lab. Synchronous applications of the Stimulus feature will be discussed in other labs.

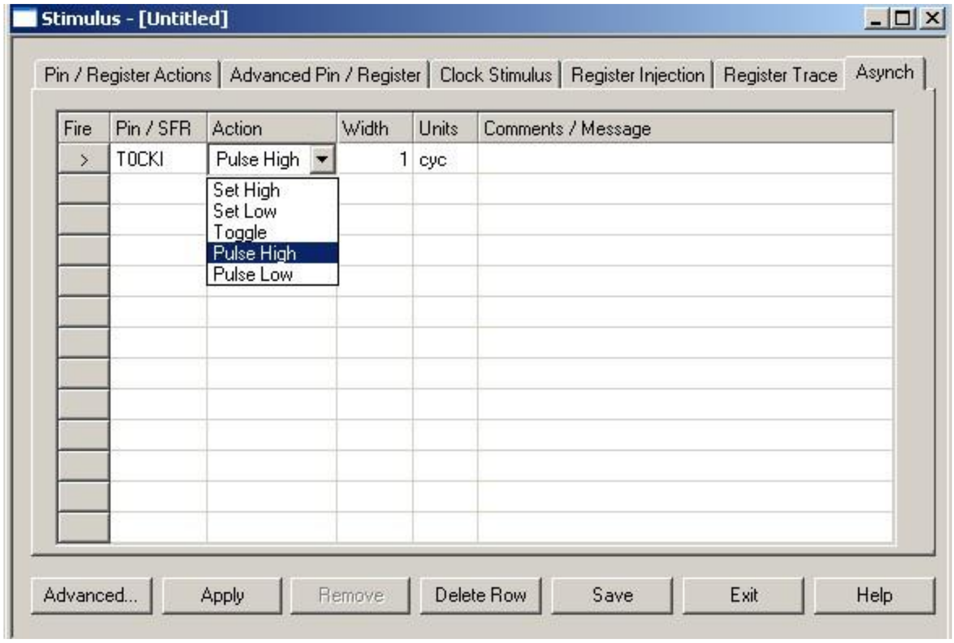
5. Click on the cell immediately below the Pin/SFR heading and select the T0CKI pin (see Figure 1-12).

FIGURE 1-12: SELECTING THE T0CKI PIN



6. Click the cell immediately under the Action Tab and select Pulse High (see Figure 1-13).

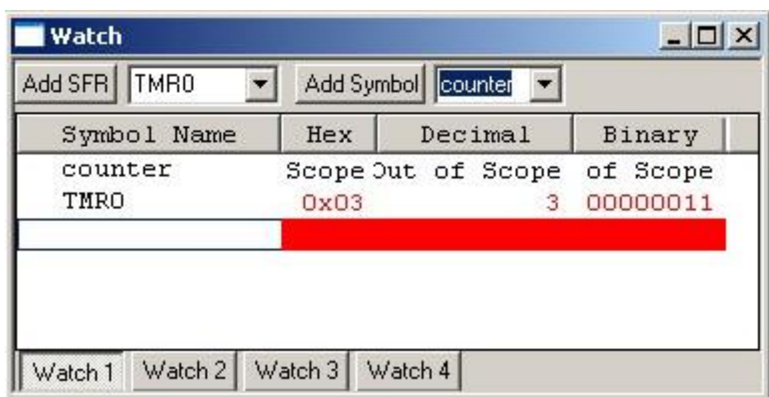
FIGURE 1-13: SELECT PULSE HIGH ACTION



The Stimulus tool is now configured so that during a simulation Run, pressing the **Fire** button next to the T0CKI cell will pulse that pin input High.

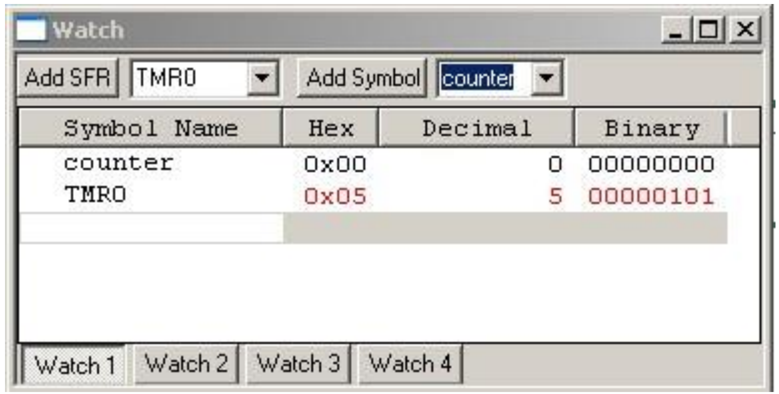
7. Open the Watch window and add the TMR0 Special Function Register as well as the `counter` symbol (see Figure 1-14).

FIGURE 1-14: WATCH WINDOW WITH TMR0 AND COUNTER REGISTERS



8. Click **Reset** then **Run** buttons in the debugger toolbar.
9. While the simulation is running, press the **Fire** button in Stimulus next to the T0CKI cell 5 times.
10. Next, stop the simulation and observe the changes to the TMR0 register. The Watch window should resemble Figure 1-15.

FIGURE 1-15: UPDATED WATCH WINDOW



Note that the TMR0 register has a value of 5 corresponding to the number of **Fire** button presses.

Try pressing the **Fire** button enough times to generate a TMR0 overflow interrupt that will increment the `counter` variable.

EXERCISES

1. Using the code and `Init()` function from Lab 1, configure the prescaler to generate a TMR0 interrupt that will increment the counter variable for each of the following periods **exactly** assuming that the PIC16F690 internal 8 MHz oscillator is used:
   a. 8.192 mS
   b. 1.024 mS
   c. 15.616 mS
2. Using Equation 1-2, develop a new equation that determines the prescaler value based off the required overflow period. Develop the equation further to determine a value to preload into TMR0 to generate an interrupt that doesn't fit neatly into a specific prescaler value.
3. Configure the application code used in question 1 to increment the `counter` variable every 1 second.
4. Calculate the minimum external clock source periods on the T0CKI pin for the following (these assume you are using the PIC16F690):
   a. Using an external crystal oscillator of 20 MHz and a Timer0 prescaler value of:i. 32 ii. 64
   b. Using an external crystal oscillator of 20 MHz with the Timer0 prescaler disabled.
   c. Using the internal 32 kHz oscillator with the Timer0 prescaler set to 128.
5. Refer to the PIC16F690 data sheet (DS41262), Table 17-5 in the Electrical Specifications section. Suppose that all equations have been performed

correctly. What is an external condition that could affect the synchronous operation of the application using an external source on the T0CKI pin?

MICROCHIP

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com Web Address: www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423 Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523 Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario, Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166 Fax: 81-45-471-6122

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or 82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-572-9526
Fax: 886-3-572-6459

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611 Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

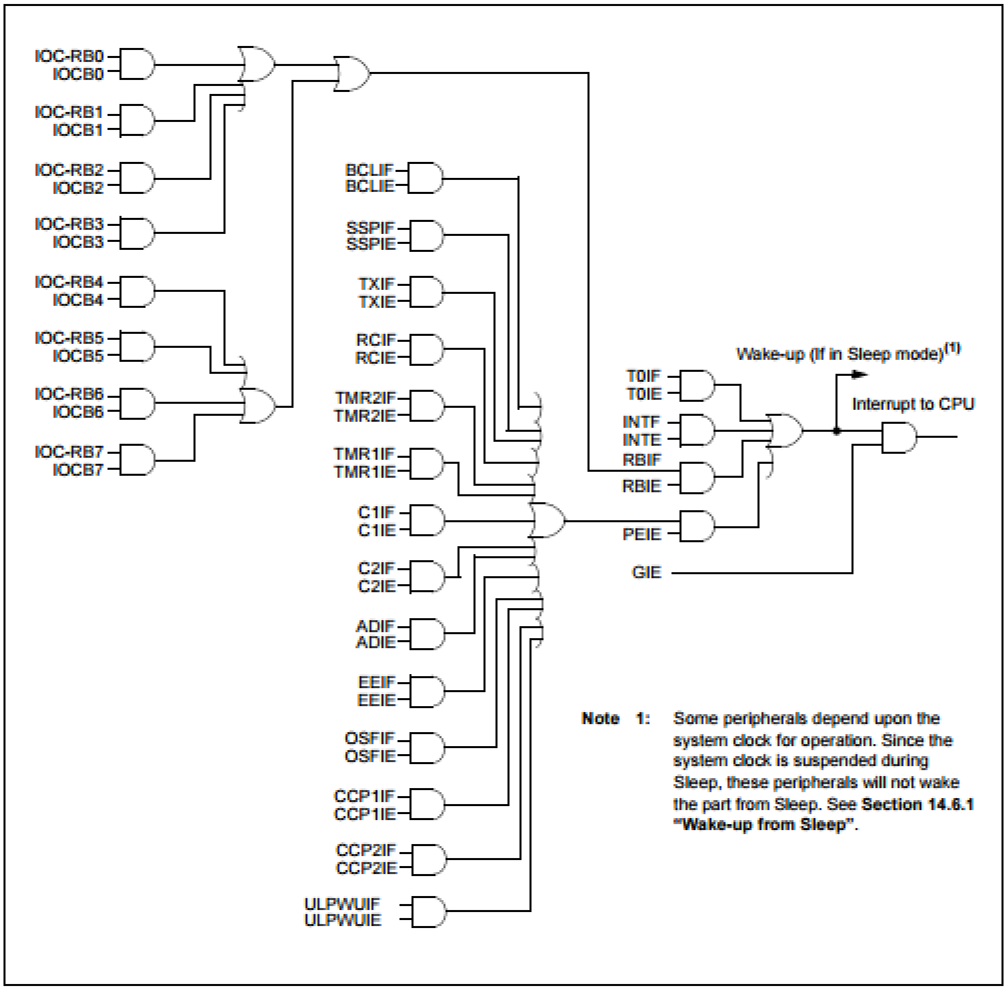
UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

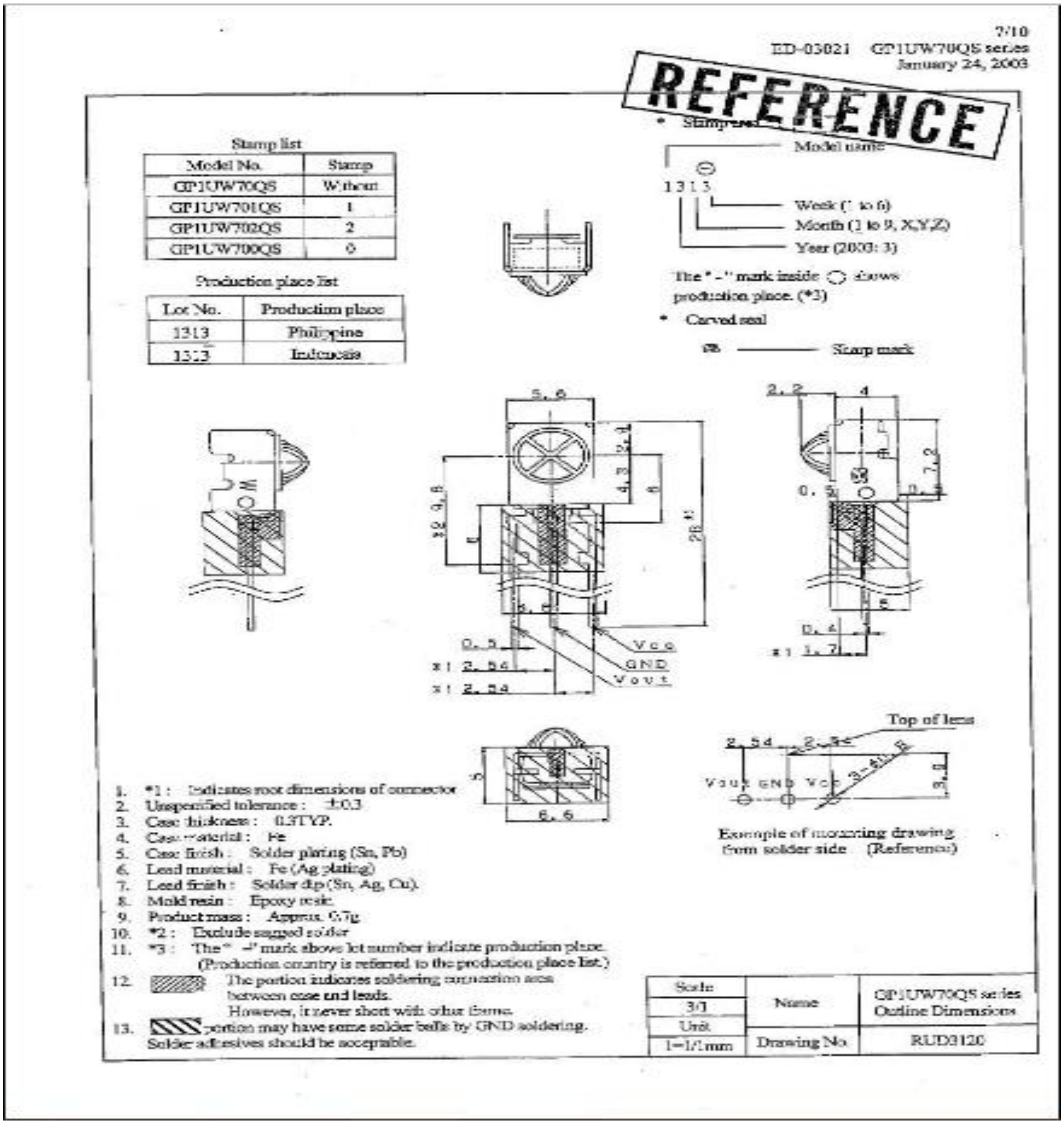
01/02/08

Interrupt Logic Diagram

I enable the IOC in the INIT() part of my code on line 59.

FIGURE 14-7: INTERRUPT LOGIC





7/10
ED-03021 GP1UW70QS series
January 24, 2003

REFERENCE

Stamp list

Model No.	Stamp
GP1UW70QS	Without
GP1UW701QS	1
GP1UW702QS	2
GP1UW700QS	0

Production place list

Lot No.	Production place
1313	Philippine
1313	Indonesia

* Stamped
Model name
1313
Week (1 to 6)
Month (1 to 9, X,Y,Z)
Year (2003: 3)
The "-" mark inside ◯ shows production place. (*3)
* Carved seal
Sharp mark

Example of mounting drawing from solder side (Reference)

1. *1 : Indicates root dimensions of connector
2. Unspecified tolerance : ±0.3
3. Case thickness : 0.3TYP.
4. Case material : Fe
5. Case finish : Solder plating (Sn, Pb)
6. Lead material : Fe (Ag plating)
7. Lead finish : Solder dip (Sn, Ag, Cu)
8. Mold resin : Epoxy resin
9. Product mass : Approx. 0.7g
10. *2 : Exclude sagged solder
11. *3 : The "-" mark above lot number indicate production place. (Production country is referred to the production place list.)
12. The portion indicates soldering connection area between case and leads. However, it never short with other frame.
13. portion may have some solder balls by GND soldering. Solder adhesives should be acceptable.

Scale	Name	GP1UW70QS series Outline Dimensions
3/1		
Unit	Drawing No.	RUD3120
1=1/1mm		

Serial Baud Rates, Bit Timing and Error Tolerance

Introduction

Asynchronous serial transmission is a mechanism to pass data from one device to another. It is termed asynchronous because the transmission timing conforms to a predefined timing specification as opposed to a synchronous mechanism where an additional clocking signal will indicate when a new data bit is being transmitted.

Byte data is transmitted as a series of eight bits with a preceding start bit to indicate when transmission is beginning and with a stop bit which indicates when all bits have been sent and to allow the next start bit to be detected; there needs to be a transition in the signal line to detect the start bit and the stop bit guarantees this. A minimum of 10 bits will therefore be transmitted to send an 8-bit data value.

Asynchronous serial is transmitted at a baud rate and, for a digital signal, this equates to the maximum number of bits that can be sent per second. The time each bit is present for (the bit time) is the reciprocal of the baud rate -

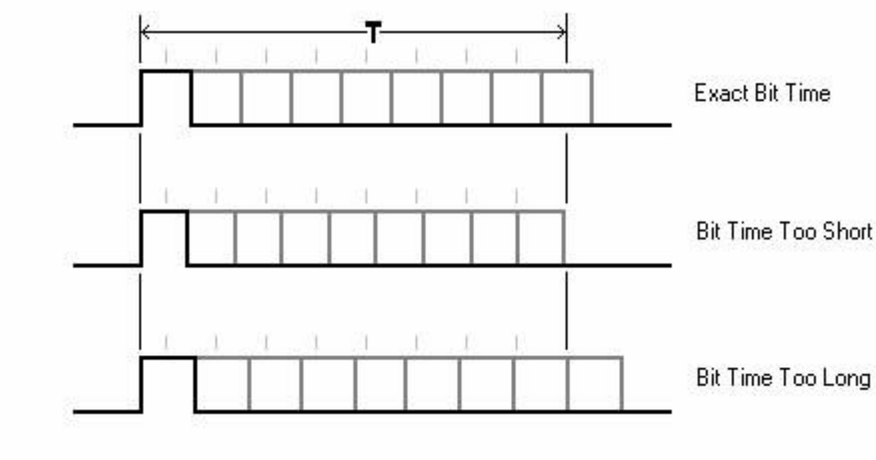
baud rate = 1 / bit time

bit time = 1 / baud rate

Asynchronous Serial Timing

A transmitting device should send its data at a specific baud rate with the correct bit time but that the bit timing actually used may sometimes be too short or too long.

The receiving device will expect the bit timing to be correct for the baud rate specified and will use that bit timing to determine the data received. What data is received will depend on the bit timing actually used by the transmitting device.



The top waveform shows 8-bit serial being received which has the correct bit time for a specific baud rate. The 8-bit data is preceded by a start bit which has the same bit time as each subsequent data bit.

To determine the 8-bit value sent the data stream is sampled in the middle of each data bit. The levels at those points will determine the data value. Note that the 8-bit data value is sent lsb first and msb last.

The internal bit timing synchronises to the leading edge of the start bit then one and a half bit times later a sample is taken in the middle of the first data bit. After a further bit time delay a sample is taken in the middle of the second data bit and so on until a

sample has been taken in the middle of the eighth data bit.

The time taken from synchronising to the leading edge of the start bit to sampling in the middle of the eighth data bit (T) is equal to 8.5 times the bit time ($Tbit_{exact}$) -

$$T = 8.5 \times Tbit_{exact}$$

The middle waveform shows a transmission when the bit time is too short ($Tbit_{short}$).

When it comes to sampling the middle of the eighth data bit that bit has just passed; the sampling renders an inaccurate sample, a corrupt data byte.

Sampling fails when -

$$9 \times Tbit_{short} < T$$

The bottom waveform shows a transmission when the bit time is too long ($Tbit_{long}$).

When it comes to sampling the middle of the eighth data bit that bit has not yet started; the sampling renders an inaccurate sample, a corrupt data byte.

Sampling fails when -

$$8 \times Tbit_{long} > T$$

When the bit time is too short

Sampling fails when -

$$9 \times Tbit_{short} < T$$

$$9 \times Tbit_{short} < 8.5 \times Tbit_{exact}$$

$$Tbit_{short} < 8.5/9 \times Tbit_{exact}$$

Correspondingly, sampling succeeds when -

$$Tbit_{short} >= 8.5/9 \times Tbit_{exact}$$

When the bit time is too long

Sampling fails when -

$$8 \times Tbit_{long} > T$$

$$8 \times Tbit_{long} > 8.5 \times Tbit_{exact}$$

$$Tbit_{long} > 8.5/8 \times Tbit_{exact}$$

Correspondingly, sampling succeeds when -

$Tbit_{long} <= 8.5/8 \times Tbit_{exact}$

Putting it all together

We have seen that sampling succeeds when -

$Tbit_{short} >= 8.5/9 \times Tbit_{exact}$

and

$Tbitlong <= 8.5/8 \times Tbit_{exact}$

We can therefore say a valid bit time (Tbit) can range from $Tbit_{short}$ to $Tbit_{long}$ and when sampled using a $Tbit_{exact}$ timing the data will be sampled correctly and return the correct data value result -

$Tbit = Tbit_{short}$ to $Tbit_{long}$

$Tbit = (8.5/9 \times Tbit_{exact})$ to $(8.5/8 \times Tbit_{exact})$

Expressed in terms of percentage -

Tbit = (94.44% of $Tbit_{exact}$) to (106.25% of $Tbit_{exact}$)

$Tbit = Tbit_{exact}$ -5.56% / +6.25%

When we apply this to some common baud rates we can see the valid range of bit timings (in approximate microseconds) allowed for thatc baud rate -

Baud Rate	$Tbit_{exact}$	$Tbit_{short}$	$Tbit_{long}$
600	1667	1574	1771
1200	833	787	885
2400	417	394	443
4800	208	196	221

9600	104	98	110

Baud rate tolerance

When shortest and longest allowed bit times are converted to baud rates we can see the range of valid baud rates which can be sampled correctly using the nominal baud rate sampling time -

Baud Rate	Minimum	Maximum
600	565	635
1200	1130	1271
2400	2257	2538
4800	4525	5102
9600	9091	10204

This equates to a tolerance in baud rate errors of approximately +/- 6%

Note, that because baud rate and bit times are reciprocals of each other, the acceptable error percentages in bit time are not the same as the acceptable error percentages for baud rate.

Reflection

After finally completing the project, I have learned a considerable amount about the PIC16F886/7 Microcontroller, baud rates, IR protocols and how data streams are sent, and timers used within the actual microcontroller. The skills and concepts I have learned have laid a foundation for me to expand my knowledge of the vast family of microcontrollers and everything relating to their utilities. I enjoyed the coding process the most as it proved to be the most challenging aspect of the entire project as well as the most rewarding.