Universal Remote

By John Tran, Daniel Barraca

Instructor Jeff Gerfen

CPE 329 - 01 Spring Project

3 June 2016

Introduction

This project aims to build a universal remote that can read an infrared signals from other remotes (ie: TVs, projectors, air conditioners, etc) to determine clock cycles and patterns, store the signal in memory, and emit the same signal upon user's request.

Demo:

https://youtu.be/8F5833JUIAU

System Requirements:

1. The system shall take in infrared digital inputs and store it before re-emitting the signal again.
2. Acceptable signals patterns include Japanese Branded Remotes, NEC , and Samsung
3. Input frequency must be 38kHz
4. System requires 4 AAA batteries to operate

Specifications	
MSP-EXP43062 Launchpad (with mini USB cable)	
Microcontroller	MSP430G2553
Microcontroller Package Type	20 Pin DIP
Supported Family	MSP430 Value Line
Clock Frequency	16MHz
Memory	16KB Flash, 512B RAM
Data	I2C (1), SPI (2), UART (1)
Power Supply Voltage	5.0V
I/O Ports	Port 1.0:7, Port 2.0:7, Port 3.0:7
Dimensions	65mm x 50mm
24LC256	
Pins	8 Pins
Vmax	3.4V
TSOP321	
Pins	3 Pins
Frequency	38kHz
QED123	
Wavelength	940nm
Forward Voltage	1.5V
Forward Current	1mA

System Architecture and Design

The system works by having an IR emitter modulated for 38kHz frequencies and switch/button inputs to prompt the system for input. The system then learns the code by using timers to measure logical highs and lows and use bit operations to store the code in a 32bit word. Once the word is filled (no signals can exceed 32bits) or there is no more incoming signal (if less than 32) then store the signal into the EEPROM. Once the user is ready to use it, the EEPROM will be read back out and the signal repeated once more from the emitter.

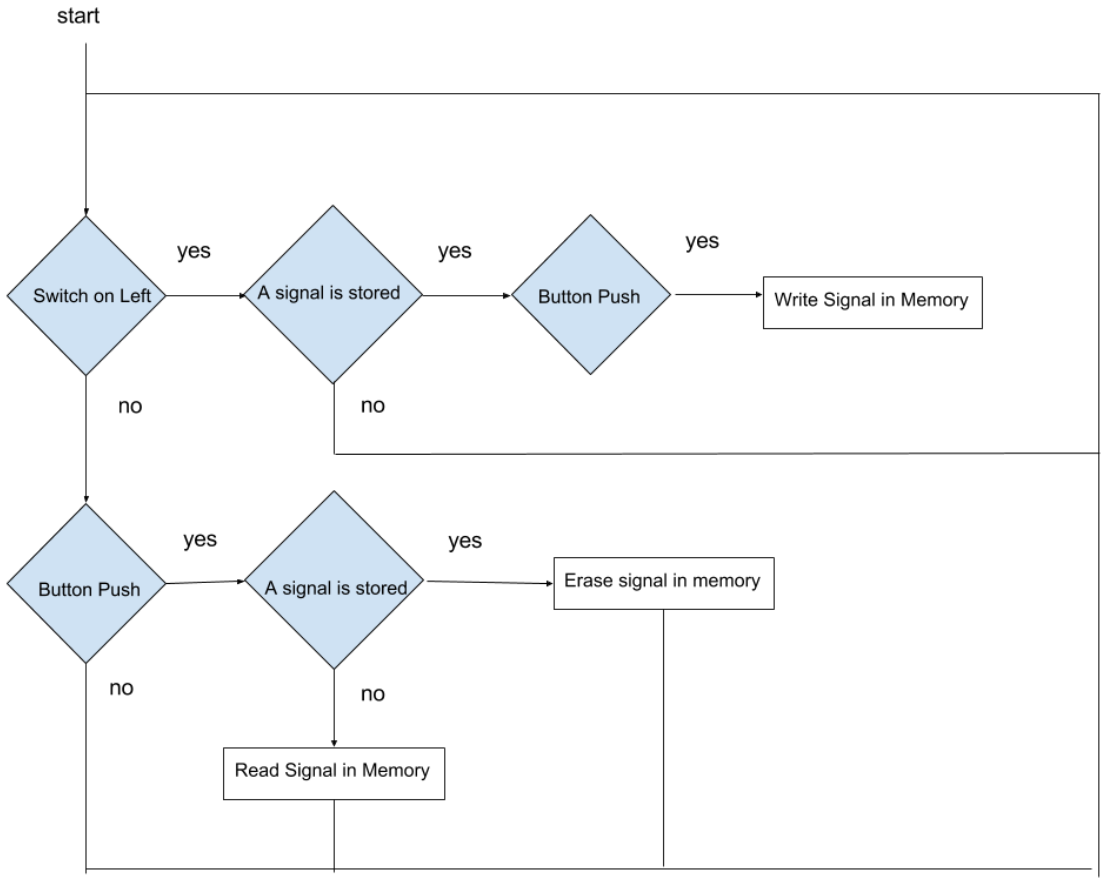

Fig 1: Flowchart

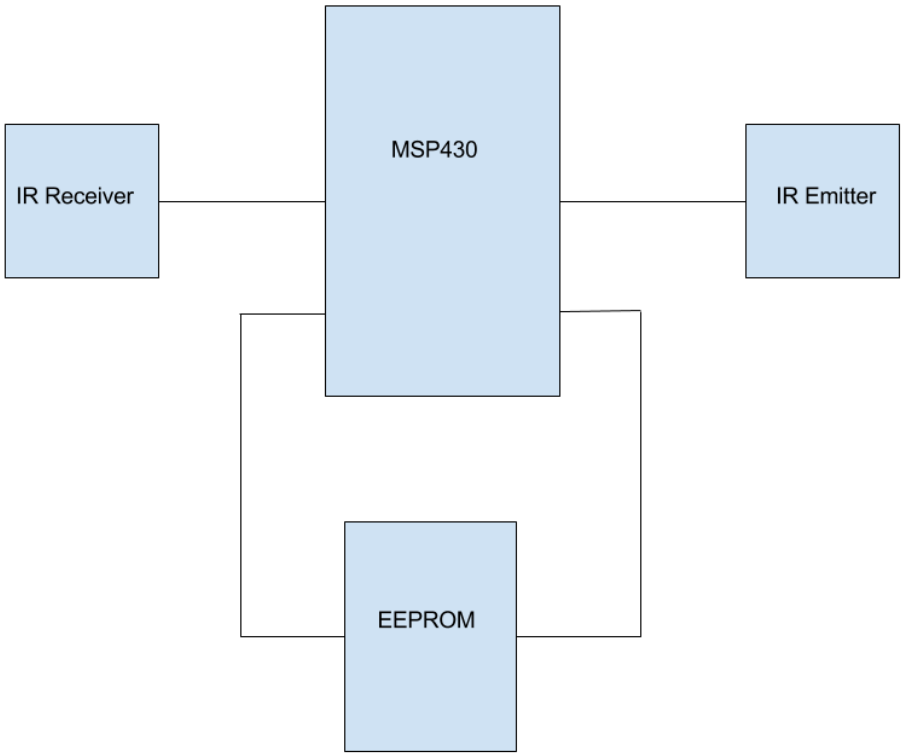

Fig 2: System Architecture

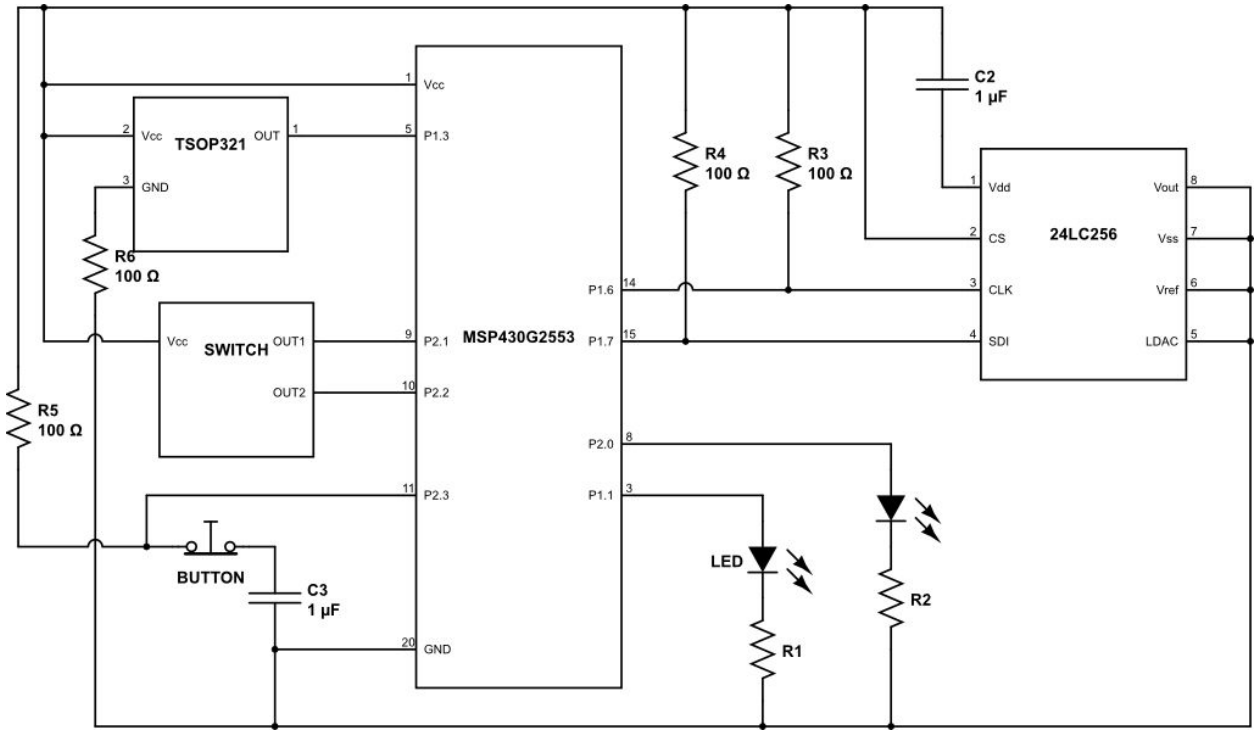

Fig3: Universal Remote Schematic

Bill of Materials:

Description	Part Number	Supplier Name	Quantity	Unit Price
MSP430 LaunchPad	296-27570-ND	Texas Instrument	1	10.37 USD
Plastic Infrared Light Emitting Diode	QED123	Fairchild Semiconductor	1	0.53 USD
IR Receiver Module for Remote Control Systems	TSOP321	Vishay	1	1.33 USD
CMOS Serial EEPROM	24LC256	microchip	1	.72 USD
Breadboard	239	Adafruit	1	7.95 USD
Housing	N/A	N/A	1	3.29 USD
TOTAL:				**24.19 USD**

Conclusion

Creating systems, particularly ones meant for production/consumption, require plenty of forethought, organization, and design. Keeping a careful and organized procedure may be tedious and time consuming at first, but once the execution phase starts it makes the whole process a lot easier to follow through. As Professor Gerfen said, "Be a designer, don't be a hack". There were plenty of moments when we did not anticipate a problem and we end up having to take shortcuts that made the final product not what we wanted initially.

For example, when we coded, tested, and debugged the MSP430 on launchpad - the system worked perfectly. Then we had to remove the MSP430 and install it onto a breadboard before putting it on a printed circuit board for the final product. That was the moment when everything fell apart because the MSP430 off the board did not have an accurate clock anymore (16MHZ SM) and was not able to produce 38kHz packets while doing other things thus creating inconsistent pulses ranging anywhere between 13 microseconds to 40 microseconds. This made us abandon the original plan to keep the remote compact and low profile all together and keep the MSP430 on launchpad while trying to stuff it into a bigger housing.

A full vertical integration of a design can offer a lot of unexpected learning opportunities. We went in thinking we'd only need to tie some circuits together and figure out a learning algorithm for variadic signals only to find that we need to learn how to do other things on top of our computer engineering discipline such as 3D printing a housing, making and designing printed circuit boards, disciplining ourselves to organize things, and even getting right down to sawing, drilling, and sanding all of which made the project a lot more fun. Most importantly however, we were able to reinforce the important concepts of clocks, timers, interrupts, I2C communications, UART, Baudings, and more from this project.

Overall the project turned out very well apart from it not looking like what we wanted it to be (the device looks like something that came out of WWII). We weren't able to receive all signals because of MSP430's limited computational power but it was still able to function with a select few standardized signal protocols that we had to research on.

Appendix A – User's Manual

Microcontroller MSP430
https://polylearn.calpoly.edu/AY_2015-2016/pluginfile.php/465336/mod_resource/content/0/MSP430x2xx%20Family%20Users%20Guide%20%28Rev.%20I%29%20-%20slau144i.pdf
Plastic Infrared LED QED123
https://www.fairchildsemi.com/datasheets/QE/QED123.pdf
EEProm 24LC256
http://ww1.microchip.com/downloads/en/DeviceDoc/21203M.pdf
IR Reciever TSOP321
http://www.vishay.com/docs/82490/tsop321.pdf

Appendix B - Formatted code

```
/*-------------------------------------------------------------------------*/
/*   @authors : Daniel Barraca John Tran                                   */
/*   @date : 3 June 2016                                                   */
/*    Eeprom code is by  Mathieu Bisson and  Evan Hirsch from              */
/*    Assignment 8 of CPE 329 course                                       */
/*-------------------------------------------------------------------------*/
#include <msp430.h>
#include <stdio.h>
#define TRUE 1
#define FALSE 0
#define SLAVE_ADDRESS 0x50  // Control byte
#define SCL_CLOCK_DIV 160    // SCL clock divider
#define BYTE_SIZE 8
#define SHORT_SIZE 16
#define TUPLE_SIZE 24
#define WORD_SIZE 32
/*-------------------------------------------------------------------------*/
/*                GLOBAL DECLARATION | INPUT DECODER                      */
/*-------------------------------------------------------------------------*/
unsigned long int testers = 0;

unsigned char written1 = FALSE;
unsigned long int inSig = 0;
unsigned long int startDuration = 0;
unsigned long int data = 0;
unsigned long int numBits = 0;
char done = FALSE;
char startBit = FALSE;
char b1sel = FALSE;
char armed = FALSE;
/*-------------------------------------------------------------------------*/
/*                  GLOBAL DECLARATIONS | I2C                              */
/*-------------------------------------------------------------------------*/
int PtrTransmit;
unsigned char I2CBufferArray[3];
unsigned char I2CBuffer;
unsigned char sentChar;

void InitI2C(unsigned char eeprom_i2c_address);
void I2CReadInit();
void I2CWriteInit();
void EEPROM_ByteWrite(unsigned int Address , unsigned char Data);
unsigned char EEPROM_RandomRead(unsigned int Address);

void packet38(unsigned long int duration) {   //pulse 38khz for duration of clock cycles
        unsigned long int dur = 0;

        while (dur < duration) {
          P2OUT |= BIT0;
          __delay_cycles(210);
          P2OUT &= ~BIT0;
          __delay_cycles(210);
          dur += 421;
```

```
        }
}

void signal32(unsigned long int start, unsigned long int integer) {
        int ndx;
        unsigned long int mask, result;
        mask = 0x80000000;
        ndx = 0;

        if (start == 0) {
                packet38(71000);   //high for 9000cs (9.0ms)
        }
        else {
                packet38(71000);   //high for 4500cs (4.5ms)
        }

        P2OUT &= ~BIT0;       //low for 4500cs (4.5ms)
        __delay_cycles(72000);

        while (ndx < 32) {
          result = (integer << ndx) & mask;
          if (result != 0) {
                  packet38(8000);
            P2OUT &= ~BIT0;
            __delay_cycles(27040);
          }
          else {
                  packet38(8000);
                  P2OUT &= ~BIT0;
                  __delay_cycles(8000);
          }
          ndx++;
        }

        packet38(8000);
        P2OUT &= ~BIT0;
}

void logical32(unsigned long int start, unsigned long int integer) {
        int ndx;
        unsigned long int mask, result;
        mask = 0x80000000;
        ndx = 0;

        if (start == 0) {
                P2OUT |= BIT0;   //high for 9000cs (9.0ms)
                __delay_cycles(144000);
        }
        else {
                P2OUT |= BIT0;   //high for 4500cs (4.5ms)
                __delay_cycles(72000);
        }

        P2OUT &= ~BIT0;   //low for 4500cs (4.5ms)
        __delay_cycles(72000);

        while (ndx < 32) {
```

```
            result = (integer << ndx) & mask;
            if (result != 0) {
                        P2OUT |= BIT0;
                        __delay_cycles(8960);
              P2OUT &= ~BIT0;
              __delay_cycles(27040);
            }
            else {
                        P2OUT |= BIT0;
                        __delay_cycles(8960);
                        P2OUT &= ~BIT0;
                        __delay_cycles(8960);
            }
            ndx++;
          }
}

/*-------------------------------------------------------------------------*/
/*              Initialization of the I2C Module.                  */
/*-------------------------------------------------------------------------*/
void InitI2C(unsigned char eeprom_i2c_address) {
          // Recommended initialisation steps of I2C module [Family User Guide]
          UCB0CTL1 |= UCSWRST;              // Enable SW reset
          UCB0CTL0 = UCMST + UCMODE_3 + UCSYNC;  // Master, I2C, Synchronous mode
          UCB0CTL1 = UCSSEL_2 + UCTR + UCSWRST;  // Use SMCLK, TX mode, Keep SW reset
          UCB0BR0 = SCL_CLOCK_DIV;          // fSCL = SMCLK/160 = 100kHz
          UCB0BR1 = 0;
          UCB0I2CSA  = eeprom_i2c_address;      // Slave Address
          UCB0CTL1 &= ~UCSWRST;             // Clear SW reset, resume operation
}

/*-------------------------------------------------------------------------*/
/*          Initialization of the I2C Module for Write operation.          */
/*-------------------------------------------------------------------------*/
void I2CWriteInit(void) {
          UCB0CTL1 |= UCTR;                 // UCTR=1 => Transmit Mode (R/W bit = 0)
          IFG2 &= ~UCB0TXIFG;                                       // Disable TXIFG since should be set by UCTXSTT
          IE2 &= ~UCB0RXIE;                 // Disable Receive ready interrupt
          IE2 |= UCB0TXIE;                  // Enable Transmit ready interrupt
}

/*-------------------------------------------------------------------------*/
/*          Initialization of the I2C Module for Read operation.           */
/*-------------------------------------------------------------------------*/
void I2CReadInit(void) {
          UCB0CTL1 &= ~UCTR;                // UCTR=0 => Receive Mode (R/W bit = 1)
          IFG2 &= ~UCB0RXIFG;                                                   // disable TXIFG since should be set by UCTXSTT
          IE2 &= ~UCB0TXIE;                 // disable Transmit ready interrupt+
          IE2 |= UCB0RXIE;                  // enable Receive ready interrupt
}

/*-------------------------------------------------------------------------*/
/*  Writes a data byte to a user-defined address in the EEPROM via I2C bus  */
/*-------------------------------------------------------------------------*/
void EEPROM_ByteWrite(unsigned int Address, unsigned char Data) {
 unsigned char adr_hi;
 unsigned char adr_lo;
```

```
 while (UCB0STAT & UCBUSY)                          // Wait until I2C module has
              ;
// finished all operations.

 adr_hi = Address >> 8;                             // Calculate high byte of address
 adr_lo = Address & 0xFF;                           // Calculate low byte of address

 I2CBufferArray[2] = adr_hi;                        // High byte address
 I2CBufferArray[1] = adr_lo;                        // Low byte address
 I2CBufferArray[0] = Data;                                                  // Data
 PtrTransmit = 2;                                   // Set I2CBufferArray Pointer

 I2CWriteInit();                                                                // Set flags/conditions for
writing
 UCB0CTL1 |= UCTXSTT;                               // Generate start condition
                                          // => I2C communication is started

 __bis_SR_register(LPM0_bits + GIE);                // Enter LPM0 w/ interrupts
 UCB0CTL1 |= UCTXSTP;                               // I2C stop condition
 while(UCB0CTL1 & UCTXSTP)                          // Ensure stop condition got sent
              ;
// UCTXSTP automatically cleared after STOP is generated
}

/*------------------------------------------------------------------------*/
/*   Reads a data byte from EEPROM via I2C bus using user-defined address    */
/*------------------------------------------------------------------------*/
unsigned char EEPROM_RandomRead(unsigned int Address) {
 unsigned char adr_hi;
 unsigned char adr_lo;

 while (UCB0STAT & UCBUSY)                          // Wait until I2C module has
              ;
// finished all operations

 adr_hi = Address >> 8;                             // Calculate high byte of address
 adr_lo = Address & 0xFF;                           // Calculate low byte of address

 I2CBufferArray[1] = adr_hi;                        // High byte address
 I2CBufferArray[0] = adr_lo;                        // Low byte address
 PtrTransmit = 1;                                   // Set I2CBufferArray Pointer

 // Write Address first
 I2CWriteInit();
 UCB0CTL1 |= UCTXSTT;                               // Generate start condition
                                          // => I2C communication is started
 __bis_SR_register(LPM0_bits + GIE);                // Enter LPM0 w/ interrupts

 // Read Data byte
 I2CReadInit();

 UCB0CTL1 |= UCTXSTT;                               // Generate second start condition for reading
 while(UCB0CTL1 & UCTXSTT)                          // Start condition sent?
              ;

 UCB0CTL1 |= UCTXSTP;                               // I2C stop condition
```

```
  __bis_SR_register(LPM0_bits + GIE);                   // Enter LPM0 w/ interrupts // comment this line out
  while(UCB0CTL1 & UCTXSTP)                                     // Ensure stop condition got sent
                ;
  return I2CBuffer;
}

/*-------------------------------------------------------------------------*/
/*              Write 32bit Word Signal into EEROM                 */
/*-------------------------------------------------------------------------*/
int WriteSignal(unsigned int startAddr, unsigned long int startTime,
                                unsigned long int code) {
        unsigned char storage[5];   //First two bytes reserved for start time
        unsigned short ndx = 0;
        unsigned int addr = startAddr;

        storage[0] = (startTime);
        storage[1] = (code & 0x000000FF);
        storage[2] = (code & 0x0000FF00) >> BYTE_SIZE;
        storage[3] = (code & 0x00FF0000) >> SHORT_SIZE;
        storage[4] = (code & 0xFF000000) >> TUPLE_SIZE;

        while (ndx < 5) {
                EEPROM_ByteWrite(addr, storage[ndx]);
                __delay_cycles(90000);
                addr++;
                ndx++;
        }

        return addr;
}

unsigned long int ReadSignal(unsigned short startAddr,
                                        unsigned long int *storedStart) {
        unsigned short ndx = 0;
        unsigned short addr = startAddr;
        unsigned long int temp = 0;
        unsigned long int storage[5];
        unsigned long int pull = 0;

        while (ndx < 5) {
                storage[ndx] = EEPROM_RandomRead(addr);
                __delay_cycles(90000);
                ndx++;
                addr++;
        }

        temp = storage[0];
        pull |= storage[1];
        pull |= storage[2] << BYTE_SIZE;
        pull |= storage[3] << SHORT_SIZE;
        pull |= storage[4] << TUPLE_SIZE;

        *storedStart = temp;

        return pull;
}
```

```
/*-------------------------------------------------------------------------*/
/*                          MAIN                                           */
/*-------------------------------------------------------------------------*/
//tar change

int main(void) {
        unsigned long int start = 0;
        unsigned long int code = 0;

        WDTCTL = WDTPW | WDTHOLD;               // Stop watchdog timer

  if (CALBC1_16MHZ==0xFF) {                     // If calibration constant erased
    while(1);                                   // do not load, trap CPU!!
  }
  DCOCTL = 0;                                   // Select lowest DCOx and MODx settings
  BCSCTL1 = CALBC1_16MHZ;                       // Set range
  DCOCTL = CALDCO_16MHZ;                        // Set DCO step + modulation

  P1DIR |= BIT0;
  P1OUT &= ~BIT0;

  P1DIR &= (~BIT3);                             // Set P1.3 SEL as Input
  P1OUT |= BIT3;
  P1IE  |= BIT3;
  P1REN |= BIT3;
  P1IES |= BIT3;                                // Falling Edge 1 -> 0
  P1IFG &= ~BIT3;                               // Clear interrupt flag for P2.3

  P2DIR |= BIT0;
  P2OUT &= ~BIT0;
  P2DIR |= BIT5;
  P2DIR &= ~BIT1;                               //Input Mode
  P2DIR &= ~BIT2;                               //Output Mode
  P2DIR &= ~BIT3;                               //Button 1
  P2DIR &= ~BIT4;                               //Button 2

  TACTL = TASSEL_2 + MC_2 + ID_3;               // SMCLK + Continuous Mode

        P1SEL |= BIT6 + BIT7;                   // Assign I2C pins to USCI_B0 (BIT6 = UCB0SCL, BIT7 = UCB0SDA)
        P1SEL2|= BIT6 + BIT7;                   // Assign I2C pins to USCI_B0 (BIT6 = UCB0SCL, BIT7 = UCB0SDA)
        _enable_interrupts();

        InitI2C(SLAVE_ADDRESS);                                                 // Initialize necessary I2C bits

        while(1) {
                if (numBits == WORD_SIZE) {
                        if (b1sel == TRUE && written1 == FALSE) {
                          WriteSignal(0X5566, startDuration, data);
                          written1 = TRUE;
                                code = ReadSignal(0X5566, &start);
                        }
                }

                if (written1 == TRUE) {
                        P1OUT |= BIT0;
                }
```

```
                else {
                        P1OUT &= ~BIT0;
                }

                if (!(P2IN & BIT1) && (P2IN & BIT2) && !(P2IN & BIT3) && (written1 == TRUE)) {
                        signal32(start, code);
                        if (start == 0) {
                                __delay_cycles(752000);
                        }
                        else {
                                __delay_cycles(752000);
                        }

                        armed = TRUE;
                }

                if ((P2IN & BIT1) && !(P2IN & BIT2) && !(P2IN & BIT3) && b1sel == TRUE && armed == TRUE) {
                        written1 = FALSE;
                   b1sel = FALSE;
                        armed = FALSE;
                        numBits = 0;
                        startBit = FALSE;
                        start = 0;
                        code = 0;
                   startDuration = 0;
                   data = 0;
                }
        }

        return 0;
}

/*-------------------------------------------------------------------------*/
/*          PORT 1 Interrupt Vector: Infrared Signal Input                 */
/*-------------------------------------------------------------------------*/
#pragma vector=PORT1_VECTOR
__interrupt void Port_1() {
        _disable_interrupts();

  static int lastTime = 0;
  inSig = TAR - lastTime;
  lastTime = TAR;

        if ((P2IN & BIT1) && !(P2IN & BIT3)) {         //Recieving mode |  B1
                if (numBits < WORD_SIZE) {
                  if (startBit == FALSE) {
                    P1IES &= ~BIT3;                  // Falling Edge 0 -> 1
                  }

                        if (inSig > 17000 && inSig < 19000) {    //start bit NEC
                                testers = inSig;
                                startDuration = 0;
                                P1IES |= BIT3;
                                startBit = TRUE;
                        }
                        else if (inSig > 8000 && inSig < 10000) { //start bit SAMSUNG
                                testers = inSig;
```

```
                                        startDuration = 1;
                                        P1IES |= BIT3;
                                        startBit = TRUE;
                                }
                                else if (inSig > 4000 && inSig < 5000) { //NEC 1 period
                                        data = data << 1;
                                        data |= 0x01;
                                        numBits++;
                                }
                                else if (inSig > 1500 && inSig < 3000) { //NEC 0 period
                                        data = data << 1;
                                        numBits++;
                                }
                        }

                        if (numBits == WORD_SIZE) {
                                b1sel = TRUE;
                        }
                }

                P1IFG &= ~BIT3;         // Clear interrupt flag for P2.3
                _enable_interrupts();
}

//-------------------------------------------------------------------------------
// The USCIAB0TX_ISR is structured such that it can be used to transmit any
// number of bytes by pre-loading TXByteCtr with the byte count. Also, TXData
// points to the next byte to transmit.
//-------------------------------------------------------------------------------
#pragma vector=USCIAB0TX_VECTOR
__interrupt void TX_ISR_I2C(void) {
        if(UCB0TXIFG & IFG2) {                  // Transmit Address
                UCB0TXBUF = I2CBufferArray[PtrTransmit];    // Load TX buffer, UCB0TXIFG flag cleared when data is written
to UCB0TXBUF
                PtrTransmit--;                  // Decrement TX byte counter
                if(PtrTransmit < 0) {
                        while(!(IFG2 & UCB0TXIFG))
                                ;
                        IE2 &= ~UCB0TXIE;                       // Disable TX Interrupts.
                        IFG2 &= ~UCB0TXIFG;                     // Clear USCI_B0 TX int flag
                        __bic_SR_register_on_exit(LPM0_bits);   // Exit LPM0
                }
        }
        else if(UCB0RXIFG & IFG2) {                                                     // Receive Data
                I2CBuffer = UCB0RXBUF;                  // Store received data in variable "I2CBuffer"
                __bic_SR_register_on_exit(LPM0_bits);   // Exit LPM0
        }
}
```