1300 Henley Court
Pullman, WA 99163
509.334.6306
www.store.digilentinc.com

Unit 5: IrDA Communications Protocols

Revised March 13, 2017
This manual applies to Unit 5.

1 Introduction

This unit demonstrates how to use interrupts and the core timer to decode two IrDA protocols, in an effort to teach approaches for decoding different IrDA protocols used for remote device control. The IrDA is an implementation of wireless serial communications capable of simplex as well as half-duplex operation.

I have encountered no fewer than 100 IrDA encoding protocols using pulse modulated infrared beams of light. Of the four IR remote control units I have found around my home, no two use the same encoding scheme. With so many existing IrDA protocols used for remote control devices, I present a method for decoding and encoding the NEC protocol that can be used on different schemes. I have verified this methodology for decoding the Pulse Length (also called Pulse Distance Modulated) protocol. Lab 5 will challenge you to follow the methodology presented to develop an interface for an IrDA remote control device of your own choosing.

Some silicon devices will decode NED encoded IrDA to 9600 Baud UART outputs, such as the ST3679. Other silicon devices will alleviate the burden of pulse encoding and decoding, such as the Microchip MCP2122, which allows direct interface with the processor UART. The above two IrDA hardware devices are designed to implement wireless line-of-sight asynchronous communications and do not interface with NEC IR remote control devices.

2 Objectives

1. Identify the IrDA protocol by capturing the bit stream on the IR_RX pin using the Analog Discovery 2.
2. Develop a C function that executes an algorithm to decode the IrDA bit stream.
3. Display the received codes on the LCD and send to the UART communications port.

3 Basic Knowledge

1. How to configure I/O pins on a Microchip® PIC32 PPS microprocessor.
2. How to configure the Analog Discovery 2 to display logic traces.
3. How to implement the design process for embedded processor based systems.

4 Equipment List

4.1 Hardware

1. Basys MX3 trainer board
2. Standard USB A to micro-B cable
3. Workstation computer running Windows 10 or higher, MAC OS, or Linux

In addition, we suggest the following instruments:

4. Analog Discovery 2 Logic Analyzer

4.2 Software

The following programs must be installed on your development workstation:

1. Microchip MPLAB X® v3.35 or higher
2. PLIB Peripheral Library
3. XC32 Cross Compiler
4. WaveForms 2015
5. PuTTY Terminal Emulator

5 Takeaways

1. Understanding of the basics of IrDA protocols.
2. Using instrumentation to characterize data streams.
3. Use processor external interrupts to decode signal timing patterns.
4. Approaches to using state machines to process data.

6 Fundamental Concepts

6.1 IrDA Concepts

It is reported that 99% of all consumer electronics use IrDA remote controllers.[1] These remote control devices use low-cost, near infrared LEDs and photo-sensitive transistors to transmit and receive intensity modulated light beams. The fact that these semiconductor devices have a narrow, unobstructed field of view is seen either as a security advantage or a communications limitation, depending on your point of view. This is in comparison to RF communications, such as Bluetooth.

IrDA communications has two major applications: high speed data and remote control. Remote control applications use small data packets at low data rates, whereas high speed data applications require a communications stack to manage the re-assemblage of data packets. Table 6.1 compares the key features of IrDA and Bluetooth.

[1] "What is infrared?" http://irdajp.info/what.html

Table 6.1. IrDA vs. Bluetooth features comparison.[2]

	IrDA-Data	Bluetooth
Physical Media	Infrared	RF (2.4 GHz)
Communications Range	Up to at least 1m	10cm to 100m
Connection Type, Direction	Point-to-Point, Narrow Angle (30 degrees)	Multipoint, Omni-directional
Maximum Data Rate	4Mbps (16Mbps on the way)	1Mbps (aggregate)
Security	Physical limitations offer some built-in protection	Authentication, encryption, spread spectrum
Approximate Cost	under $2	under $5

7 Background Information

7.1 IrDA Physical Layer

The Basys MX3 platform is equipped with a Rohm RPM973-H11 infrared communications module that provides the digital interface, as shown in Fig. 7.1. The three microprocessor control pins shown in Fig. 7.2 are the IR_TX, IR_RX, and the IR_PDOWN. The Rohm RPM973-H11 is in a low power mode whenever the IR_PDOWN control signal is in the high state. The IR_PDOWN must be in a low state to transmit or receive infrared signals. Both IR_TX and IR_RX are active low signals.

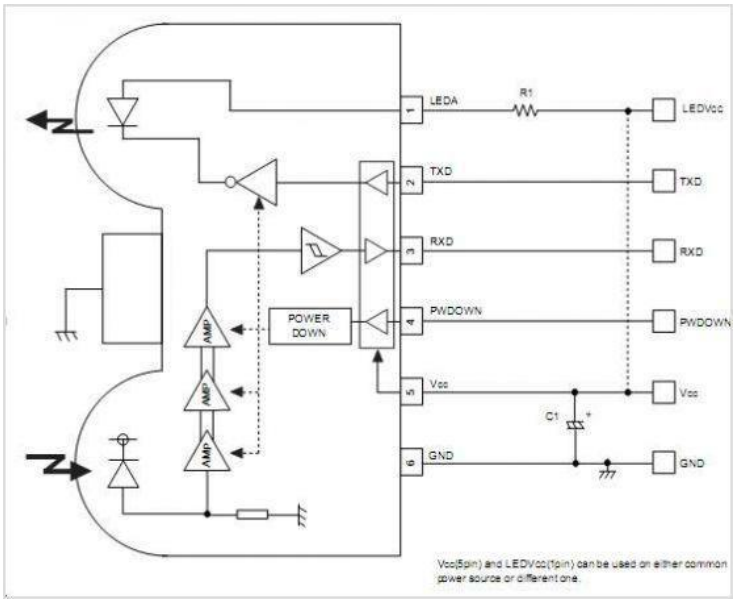

Figure 7.1. RPM973-H11 block diagram.

[2] http://www.barrgroup.com/Embedded-Systems/How-To/Wireless-Bluetooth-IrDA

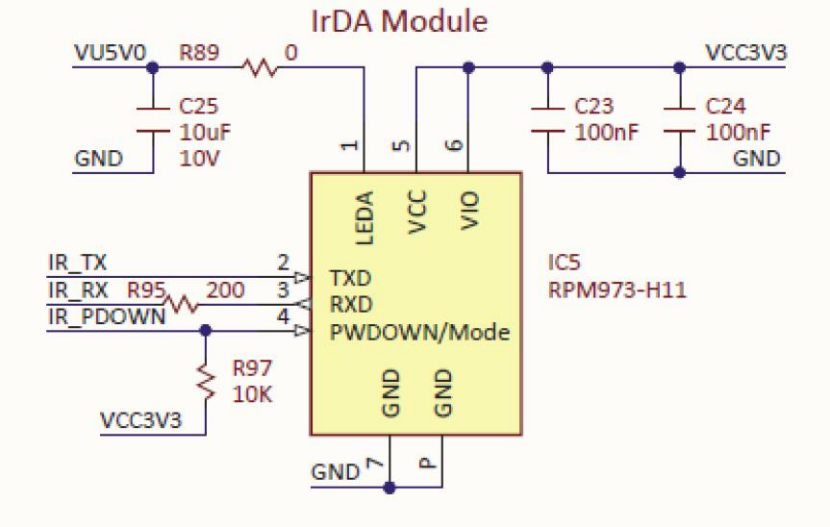

Figure 7.2. Schematic diagram of PIC32 connection to the RPM841-H11 Infrared Module.

Reference 3 lists 21 modulation protocols used for remote control devices. This reference also lists standard carrier frequencies used by these devices. In addition to different kinds of coding and different carrier frequencies, there are further variations in the data formats: with and without pre-burst, with different numbers of bits in a command, and with different bit lengths. As one soon realizes, one must match the IrDA protocol before two IrDA devices can communicate. The IrDA support reported in the PIC32MX370 data sheet (see Section 20 of Reference 6) requires an external IrDA Encoder/Decoder, such as the MC2120 suggested in the Microchip Analog Design Note ADN006. The Basys MX3 processor platform DOES NOT contain any hardware IrDA encoder/decoder. The Rohm RPM973-H11 simply asserts the IR_RX signal low whenever an IR signal of sufficient intensity is detected. Likewise, the IR LED is turned on whenever the IR_TX signal is asserted high. When using the Rohm RPM973-H11 on the Basys MX3 processor platform, all encoding and decoding of the IR pulses must be implemented by the PIC32 processor.

7.2 Characterization of the NEC IrDA Protocol

The NEC protocol will be characterized to illustrate the methodology of discovering the protocol used by an IrDA remote controller. The general procedure is to capture logic analyzer traces and match the data profile to an existing IrDA protocol. This is commonly referred to as reverse engineering.

Figure 7.3 shows that the IR LED modulation for a 38 kHz carrier is determined as the inverse of the period between LED pulses. The LED is only on for about 2.4 μs out of the 26.3 μs carrier frequency period. The duty cycle of the LED is less than 10% resulting in lower power consumption.

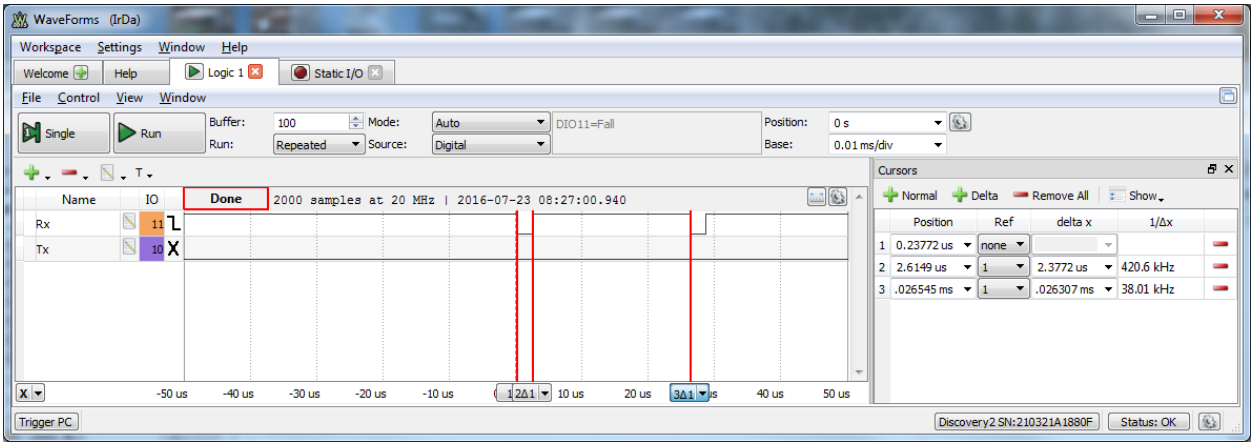

Figure 7.3. IrDA 38 kHz carrier signal.

Figure 7.4 shows a complete IrDA message using the NEC protocol. The signal is low whenever an infrared light is detected. The NEC protocol message contains 32 bits organized in eight bit bytes sent with LSB first. After a 9 ms leader and a 4.5 ms gap, a 560 µs bit marker signals the start of the LSB of the first data byte, as shown in Fig. 7.5.

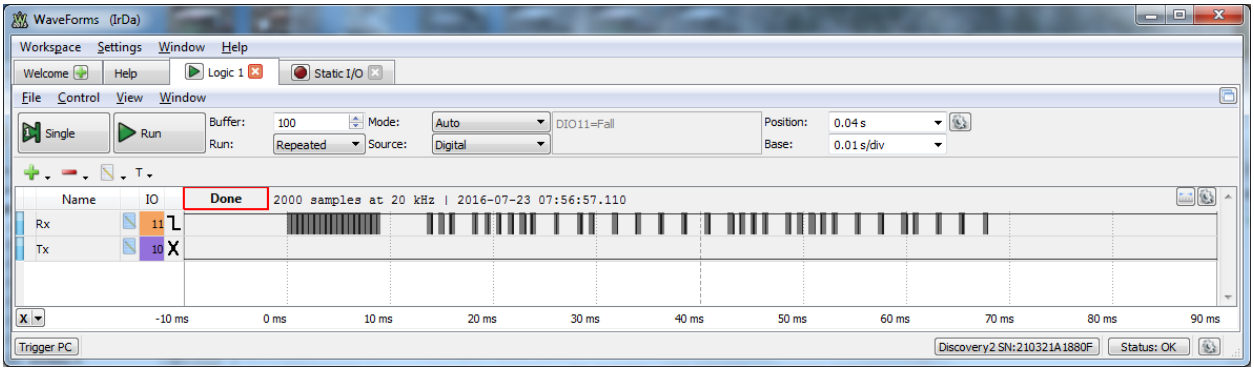

Figure 7.4. Screen capture of a NEC IrDA control message.

Figures 7.5 and 7.6 show expanded views of portions of the NEC IrDA protocol, showing the sync followed by three of the 24 data bits. Note that the differentiation between a ONE bit and a ZERO bit is the length of the gap following each 560 µs period of 38 kHz modulated infrared pulses. The last byte contains 9 pulse bursts to allow the last bit to be framed correctly.

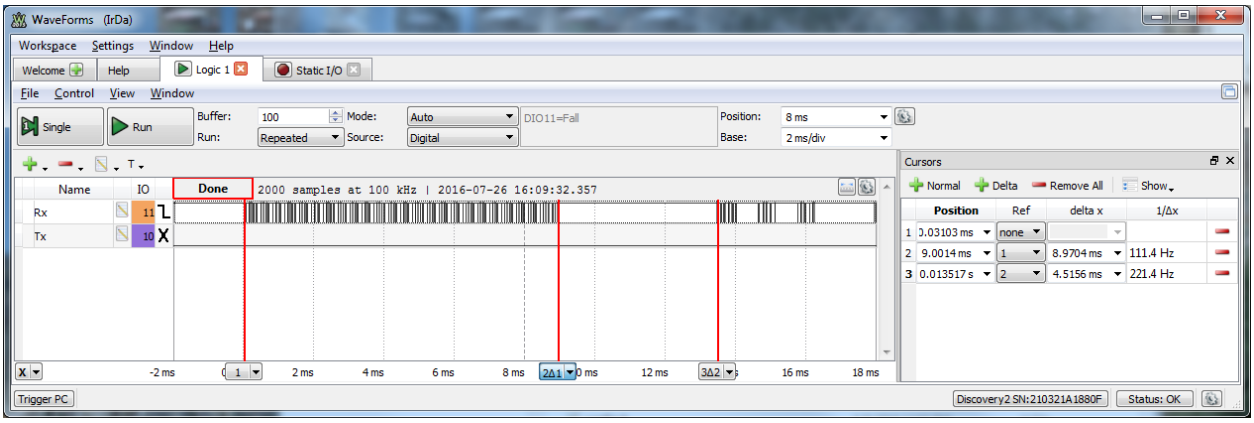

Figure 7.5. Details of the message leader and first three data bits.

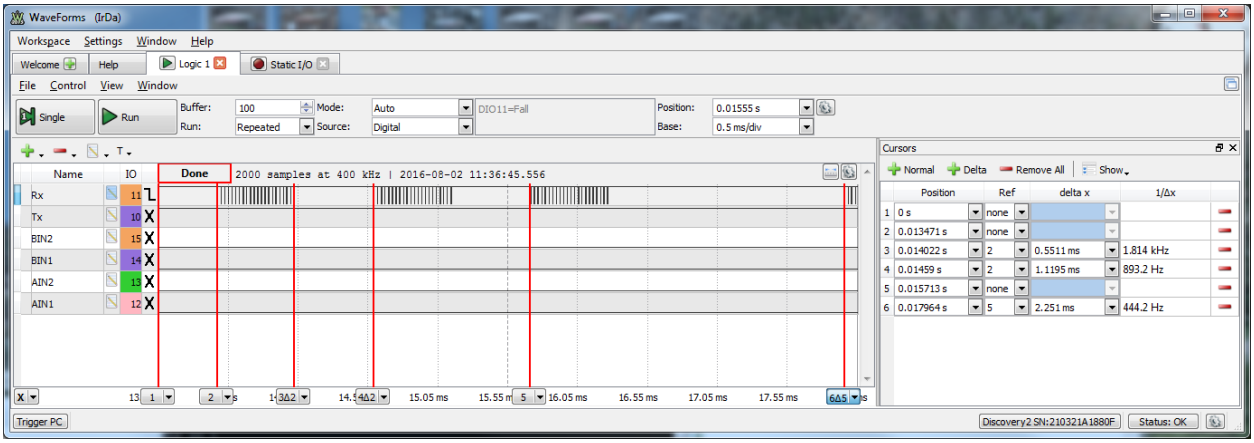

Figure 7.6. 38 kHz pulse burst timing representing two ZERO bits and a ONE bit.

Figures in Reference 4 are reproduced below to illustrate the timing of the NEC protocol. The encoding of an NEC protocol message in Fig. 7.7 shows that the entire packet consists of two bytes of data: the address byte and the command byte. Each byte is followed by its one's complement. The consequence of this encoding is that although the time to transmit a ONE bit is twice that of transmitting a ZERO bit, the time to send any NEC encoded message is constant regardless of the value of the address and control bytes.

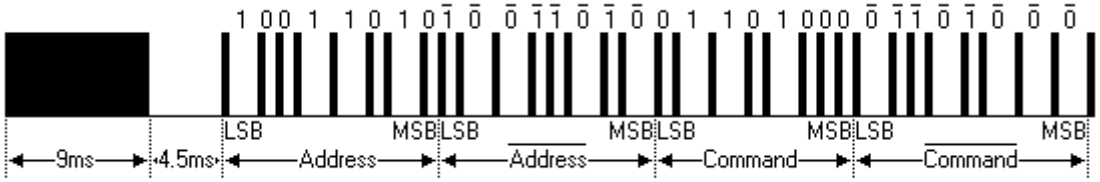

Figure 7.7. NEC encoded IrDA message.

If the key on the remote control unit is held depressed for longer than 110 ms, a repeat code is continuously sent that contains no address or command data, as illustrated in Fig. 7.8.

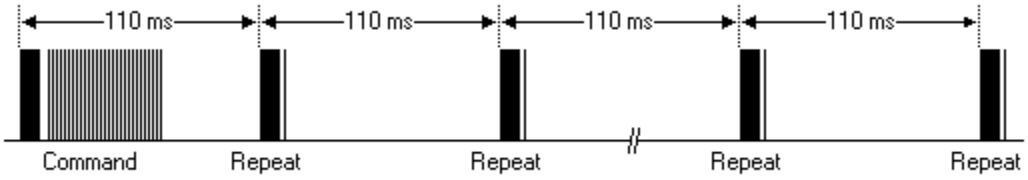

Figure 7.8. Timing diagram of a repeated NEC encoded IrDA message.

7.3 Decoding the NEC IrDA Protocol with the PIC32MX370 Processor

From the previous discussion, it is obvious that the PIC32 UART is not suitable for implementing the IrDA NEC remote control protocol. My investigation of internet documents has not produced any meaningful information as to how to implement this design. From my 30 years of microprocessor design experience, I have concluded that there are two possible approaches: application of digital signal processing concepts, namely Fast Fourier Transforms (FFT), or edge timing using PIC32 Input Compare capability or external interrupts. Interestingly, both methods involve frequency domain concepts.

The modulation and demodulation of the IrDA signal involves binary coding of the 38 kHz carrier signal. As shown above, the bit value information for the NEC protocol is contained in the period length between the 38 kHz IR beam oscillations.

My approach is to first map the PPS input of the IR_RX signal on Port B pin 6 to the external interrupt INT1 as shown in Table 7.1 below. The INT1 is configured to generate an interrupt on the falling (negative) edge of the IR_RX signal using the initialization code shown in Listing 7.1.

Listing 7.1. Initialization of INT1

```
// Set up IrDA RX interface
   INT1R = 0b00000101;     // Mapping IrDA Rx to RPB6 --> INT1
// Set up INT1 for negative edge triggering
   IEC0bits.INT1IE = 0;    // Disable INT1
   IPC1bits.INT1IP = 2;    // Set Interrupt 1 for priority level 2
   IPC1bits.INT1IS = 0;    // Set Interrupt 1 for sub-priority level 0
   INTCONbits.INT1EP = 0;  // Set for falling edge
   IFS0bits.INT1IF = 0;    // Clear the INT1 interrupt flag
   IEC0bits.INT1IE = 1;    // Enable INT1
```

Table 7.1. Table 12-1 from the PIC32MX370 datasheet for the PPS Input Pin Selection for the PIC32MX370 Processor.

Peripheral Pin	[*pin name*]R SFR	[*pin name*]R bits	[*pin name*]R Value to RPn Pin Selection
INT1	INT1R	INT1R<3:0>	0000 = RPD1 0001 = RPG9 0010 = RPB14 0011 = RPD0 0100 = RPD8 0101 = RPB6 0110 = RPD5 0111 = RPB2 1000 = RPF3(4) 1001 = RPF13(3) 1010 = Reserved 1011 = RPF2(1) 1100 = RPC2(3) 1101 = RPE8(3) 1110 = Reserved 1111 = Reserved
T3CK	T3CKR	T3CKR<3:0>	
IC1	IC1R	IC1R<3:0>	
$\overline{\text{U3CTS}}$	U3CTSR	U3CTSR<3:0>	
U4RX	U4RXR	U4RXR<3:0>	
U5RX	U5RXR	U5RXR<3:0>	
$\overline{\text{SS2}}$	SS2R	SS2R<3:0>	
OCFA	OCFAR	OCFAR<3:0>	

1. This selection is not available on 64-pin USB devices.
2. This selection is only available on 100-pin General Purpose devices.
3. This selection is not available on 64-pin USB and General Purpose devices.
4. This selection is not available when USBID functionality is used.

When the interrupt occurs, the value of the core timer operating at 40 MHz is captured. The previous core timer value is subtracted from the present core timer value to determine the period between interrupts. If this period is approximately 560 μs, a 38 kHz signal is present. The actual IR pulse length is of little concern provided it is sufficiently long to generate a processor interrupt. For our characterization case, the actual measured IR beam pulse width is 2.377 μs.

Listing 7.2 shows the C code implementing the INT1 ISR. The time between interrupts contains the needed information, hence variable “t1_2” must be declared as static. The static variable, “start_timing” remembers that

the variable “t1_2” has been initialized after a system reset. The only information needed by the NEC decoding algorithm is the time since the last interrupt.

Listing 7.2. INT1 ISR

```
void __ISR(_EXTERNAL_1_VECTOR, IPL2SOFT) ext_int1_isr(void)
{
unsigned long t1_1;               // Current time of interrupt
static unsigned long t2_1;        // Time of previous interrupt
unsigned long dt_1;               // Time interval between edges
unsigned int bitCount = 0;        // Number of decoded bits
static int start_timing = 0;      // Time capture buffer has been initialized.


   t1_1 = ReadCoreTimer();
   if(!start_timing)              // Check for first interrupt after reset
   {
       t2_1 = t1_1;               // Initialize previous time on reset.
       start_timing = 1;
   }
   else
   {
       dt_1 = t1_1 - t2_1;       // Compute time interval
       bitCount = irda_nec(dt_1);  // Call decoding function
       t2_1 = t1_1;              // Update time of last interrupt
   }
   IFS0bits.INT1IF = 0;           // Clear the interrupt flag
}
```

The core timer increments at the rate of a count for each 1/40,000,000 seconds or 0.0025 μs per count. Referring back to the NEC characterization figures, we find that the sync period should be reported 0.0045 seconds multiplied by 40,000,000 counts per second, or approximately 180000 core timer counts. When a ONE bit is encoded, the number of core timer counts since the last interrupt is (0.0025 – 0.00056) seconds multiplied by 40,000,000 counts per second, or approximately 67000 core timer counts. When a ZERO bit is encoded, the number of core timer counts since the last interrupt is (0.00112 – 0.00056) seconds multiplied by 40,000,000 counts per second, or approximately 22400 core timer counts. Since the accuracy of the crystals used to generate the IR pulses is unknown, 2% accuracy can be assumed and still allow for adequate discrimination between symbols. Table 7.2 lists the core timer count range for the NEC protocol symbols.

Table 7.2. Core Timer Count ranges.

Symbol	Minimum Core Timer Count	Maximum Core Timer Count
Sync	176400	183600
ONE bit	65660	68340
Zero bit	21952	22848

7.4 Encoding the NEC IrDA Protocol with the PIC32MX370 Processor

In order to encode an IrDA for the NEC protocol, the 38 kHz modulated bit stream must be generated. I chose to generate the 38 kHz pulsed signal using the PWM output of the PIC32 processor. Table 7.3 is a copy from the PIC32MX370 data sheet showing the PPS mapping of OC5 to Port B bit 7, which is connected to the IR_TX signal line. Listing 7.3 is the code for the IrDA initialization.

Table 7.3. PIC32MX370 PPS output mapping for bit 7 of Port B.

RPn Port Pin	RPnR SFR	RPnR bits	RPnR Value to Peripheral Selection
RPD9	RPD9R	RPD9R<3:0>	0000 = No Connect 0001 = $\overline{U3RTS}$ 0010 = U4TX 0011 = REFCLKO 0100 = U5TX 0101 = Reserved 0110 = Reserved 0111 = $\overline{SS1}$ 1000 = SDO1 1001 = Reserved 1010 = Reserved 1011 = OC5 1100 = Reserved 1101 = C1OUT 1110 = Reserved 1111 = Reserved
RPG6	RPG6R	RPG6R<3:0>	
RPB8	RPB8R	RPB8R<3:0>	
RPB15	RPB15R	RPB15R<3:0>	
RPD4	RPD4R	RPD4R<3:0>	
RPB0	RPB0R	RPB0R<3:0>	
RPE3	RPE3R	RPE3R<3:0>	
RPB7	RPB7R	RPB7R<3:0>	
RPB2	RPB2R	RPB2R<3:0>	
RPF12(4)	RPF12R	RPF12R<3:0>	
RPD12(4)	RPD12R	RPD12R<3:0>	
RPF8(4)	RPF8R	RPF8R<3:0>	
RPC3(4)	RPC3R	RPC3R<3:0>	
RPE9(4)	RPE9R	RPE9R<3:0>	

The 38 kHz carrier signal is generated by setting the Timer 2 counter register equal to the PBCLOCK divided by 38000. The PWM output is turned off by setting the OC5 reset time greater than the Timer 2 period. The 9% duty cycle 38 kHz signal is turned on by setting the OC5 reset time to the product of 0.09 times the Timer 2 period. Generating the NEC sync signal and encoding the one's and zero's is just a matter of turning the PWM on and off for the specific duration, as specified by Reference 3.

Listing 7.3. IrDA initialization

```
#define IRDA_38K_IDLE        264
#define IRDA_38K_ON          23
#define IRDA_38KHZ_PD        262
void irda_init(void)
{
int i;
// IrDA Power Down Control
   TRISGCLR = BIT_1;       // Set IR_pdown as output();
   LATGCLR =  BIT_1;       // Set IR_pdown(0);

// Set up IrDA TX interface
   PORTSetPinsDigitalOut(IOPORT_B, BIT_7); // IR_TX
   RPB7R = 0b00001011;     // Mapping OC5 to RPB7
```

```
// Enable OC5 for PWM operation
   OpenTimer2((T2_ON | T2_SOURCE_INT | T2_PS_1_1), IRDA_38KHZ_PD);
   OpenOC5((OC_ON|OC_TIMER_MODE16|OC_TIMER2_SRC|OC_PWM_FAULT_PIN_DISABLE),
           IRDA_38K_IDLE, IRDA_38K_IDLE);

// Set up IrDA RX interface
   INT1R = 0b00000101;     // Mapping IrDA Rx to RPB6 --> INT1
   PORTSetPinsDigitalIn(IOPORT_B, BIT_6); // IR_RX
// Set up INT1 for negative edge triggering
   IEC0bits.INT1IE = 0;    // Disable INT1
   IPC1bits.INT1IP = 2;    // Set Interrupt 1 for priority level 2
   IPC1bits.INT1IS = 0;    // Set Interrupt 1 for sub-priority level 0
   INTCONbits.INT1EP = 0;  // Set for falling edge
   IFS0bits.INT1IF = 0;    // Clear the INT1 interrupt flag
   IEC0bits.INT1IE = 1;    // Enable INT1
}
```

8 References

1. Remote Control, https://en.wikipedia.org/wiki/Remote_control
2. Consumer IR, https://en.wikipedia.org/wiki/Consumer_IR
3. Data Formats for IR Remote Control, http://www.vishay.com/docs/80071/dataform.pdf
4. AN #157 Implementation of IR NEC Protocol, http://www.mcselec.com/index.php?option=com_content&task=view&id=223
5. Remote Control with IrDA® Transceivers, http://datasheet.octopart.com/TFBS4710-TT1-Vishay-datasheet-12514678.pdf
6. SB –Projects NEC Protocol, http://www.sbprojects.com/knowledge/ir/nec.php
7. PIC32MX330/350/370/430/450/470 Data Sheet, http://ww1.microchip.com/downloads/en/DeviceDoc/60001185E.pdf
8. Understanding Sony IR remote codes, LIRC files, *and the Arduino library,* http://www.righto.com/2010/03/understanding-sony-ir-remote-codes-lirc.html