Tutorial: Infrared Localization with Intel Edison

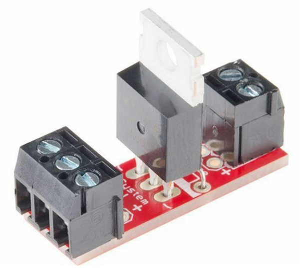
Authors: Anthony Nguyen, Pranjal Rastogi, Raymond Andrade

Parts Required:

LED - Infrared 950nm:

IR Receiver Diode - TSOP38238:

SparkFun MOSFET Power Control Kit:



*Does not come soldered

The listed parts can be found at the following links:

Component	URL
LED - Infrared 950nm	https://www.sparkfun.com/products/9349
IR Receiver Diode - TSOP38238	https://www.sparkfun.com/products/10266
SparkFun MOSFET Power Control Kit	https://www.sparkfun.com/products/12959

Introduction to Infrared Communication:

Most infrared communication protocols require the emitted signal to be modulated between 36-40 kHz. This is important to know because most infrared receivers are made with a band pass filter to account for this modulation. The receiver component used in this tutorial is tuned to receive at 38 kHz.

The following images will provide the appropriate understanding of the basic functionality of both the emitter and receiver portions of the communication:

- Figure 1 shows a PWM signal, which would drive the IR emitter, at 38 kHz. This shows what it looks like to send a HIGH from 0 to .5ms, and a LOW from .5 ms to 1ms.

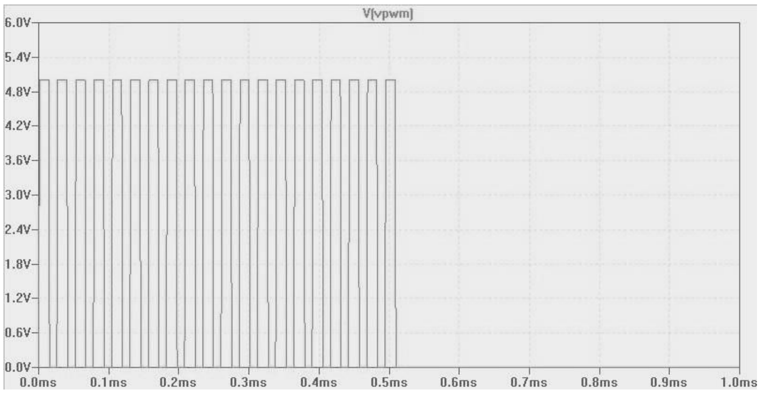


Figure 1. Example of IR emission

- Figure 2 shows how the waveform in Figure 1 would be received

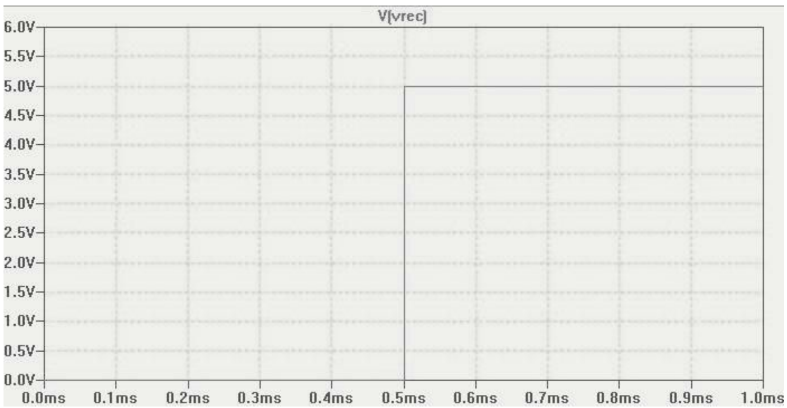


Figure 2. Example of IR reception

Figure 2 shows the behavior of the receiver during the reception of a HIGH and a LOW

Introduction to Infrared Communication Protocols:

There are many commercial infrared communication protocols used by companies such as Sony SIRC, Phillips RC5/RC6, and NEC. All these protocols have some sort of preamble which is mapped on to a pattern of HIGHs and LOWs of various durations. They differ in how data bits map to a HIGH/LOW pattern. But it is important to note that a HIGH is NOT a constant high voltage but a square wave of frequency 36-40 kHz, as shown in Figure 1.

In this tutorial, we create a custom IR Communication Protocol that is created to be usable and easy to understand. This protocol can be altered or optimized to fit the needs of any given application.

The components of our protocol is described as follows:

Preamble	2 ms HIGH followed by a 2 ms LOW. Sent (**preamble length**) times.
Bit 0	1 ms HIGH followed by a 5 ms LOW
Bit 1	5 ms HIGH followed by a 1 ms LOW
Preamble Length	**5 (default value)**

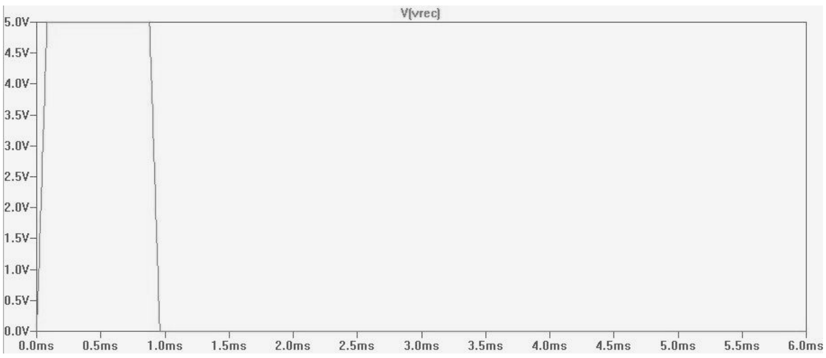


Figure 3. Example of Data 0

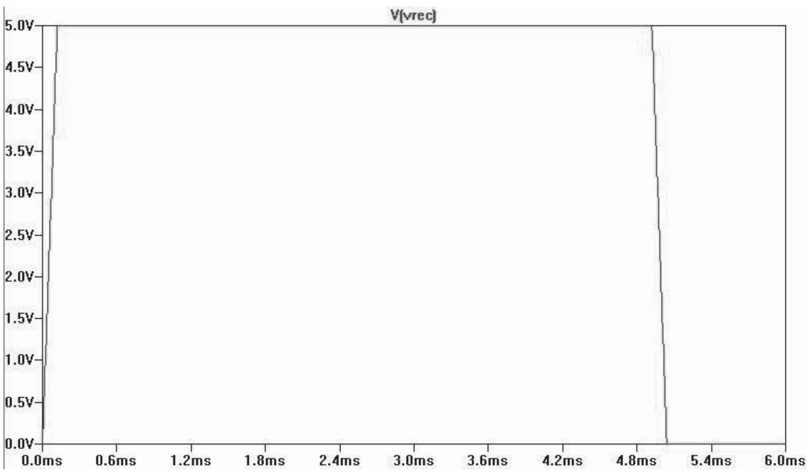


Figure 4. Example of Data 1

Transmitting Messages

In this tutorial, we will make use of the use this IR protocol described above in order to transmit 4 bit messages that correspond to locations. However, the protocol could be used to send any number of bits to have any desired meaning.

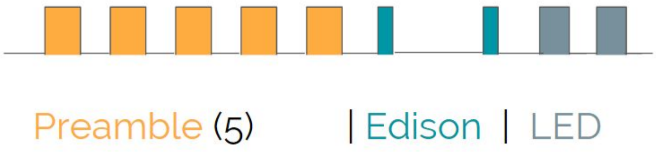


Figure 5. Our Transmitted Message

For our messages, we have 1 Preamble and 4 bits. The first 2 bits together comprise the Edison ID and the later 2 bits comprise the IR Emitter ID.

So in the above example, the total duration of the message (inclusive of preamble) will be 12 ms + 5 ms x 4 = 32 ms.

Implementation of Infrared Communication

1. First setup the emitter circuit. There are two ways to drive the emitter–
    a. Directly source current from the Edison as shown in figure 3
        i. The emitter range will be limited here due current limitations of the edison's pins

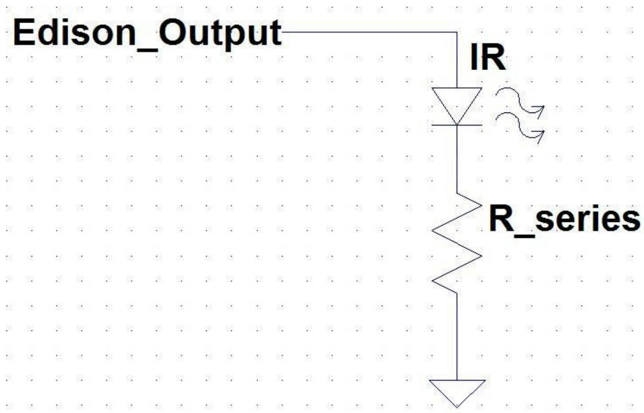


Figure 3. Diagram showing Edison directly sourcing IR emitter

*The series resistance here should be designed so that the edison does not source more current than it is able

b. Use a MOSFET driver PCB as shown in figure 4
    i. The emitter range can be completely utilized

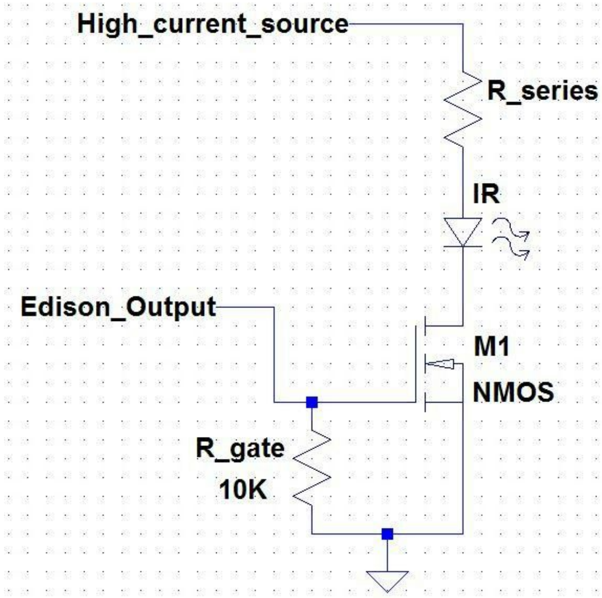


Figure 4. Schematic of MOSFET driver circuit

*The series resistance for the MOSFET circuit should be designed to limit the current through the emitter to be no more than 50 mA, as this is the maximum rated current for the component. The value will vary depending on the source used.

Deciding which way to drive you emitter depends on the application.

2. Login into the Edison and type the transmitter driver code (given in Appendix A) in file
    ir_transmit.c
3. Compile the code using -
    gcc –lmraa –o ir_transmit ir_transmit.c
4. Run the executable by typing the command -
    ./ir_transmit 5 2

   The first argument here sets length of preamble and the second one sets the Edison ID. After executing this command, the transmitter Edison is sending modulated signals continuously on pins 3, 5, 6 and 9.

The following table will show the relationship, in bits, between the pins, their representative locations, and the Edison and emitter ID:

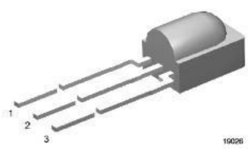
Edison #	Edison ID		Emitter ID		Location	Pin #
1	0	0	0	0	1	3
	0	0	0	1	2	5
	0	0	1	0	3	6
	0	0	1	1	4	9
2	0	1	0	0	5	3
	0	1	0	1	6	5
	0	1	1	0	7	6
	0	1	1	1	8	9
3	1	0	0	0	9	3
	1	0	0	1	10	5
	1	0	1	0	11	6
	1	0	1	1	12	9
4	1	1	0	0	13	3
	1	1	0	1	14	5
	1	1	1	0	15	6
	1	1	1	1	16	9

Figure 5. Transmitted messages for each location

If you have set the transmitter circuitry as indicated above, the infrared emitter should be blinking now. There are two easy ways to verify proper function for the transmitter:

- ❏ Use your smartphone camera - its filter is polarized, which allows you to see the emitted IR beams
- ❏ Use an oscilloscope - this will confirm that you have the proper signal being generated, and ensure the proper operation of the IR emitter by checking that the voltage drop is between 1.4-1.6 volts, across the emitter

5. Set up the receiver using the pinout figure below:



MECHANICAL DATA
Pinning:
1 = OUT, 2 = GND, 3 = V_S

*The pins shown can be directly connected to their associated pins on the Edison arduino breakout board.

6. Serially login into the receiver side Edison and type the receiver side code (given in Appendix B) in file

   ir_receive.c

7. Compile the receive side code using -

   gcc -lmraa –o ir_receive ir_receive.c

8. Run the executable by typing the following command-

   ./ir_receive 5

   The argument given in the above command is the preamble_length

   Some important notes:

   - ❏ Make sure to specify the same preamble length on the receive Edison as the one given on the transmitting Edison
   - ❏ Make sure that the receiver is positioned to correctly receive the emitted signal
   - ❏ If the receiver is too close or too far from the emitter the signal will not be received correctly

9. After you have correctly aligned your receiver and run the receiver code, you should be able to receive correct locations.

   The output for a properly received message will be as follows:

   Returning location 8

   If the signal is not received correctly, the code will have the following output:

   Not enough bits matched, Return 0

Appendix A : Transmit Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mraa/pwm.h>
#include <mraa/gpio.h>

int preamble_length = 5;
#define SCALING_FACTOR 5
#define PREAMBLE_DELAY 750000/SCALING_FACTOR // 10 miliseconds

// by default, the edison has ID 0
int edisonID = 0;

#define SHORT_DELAY 350000/SCALING_FACTOR // 5 miliseconds
#define LONG_DELAY 1400000/SCALING_FACTOR // 20 miliseconds
#define MID_DELAY 1050000/SCALING_FACTOR // 15 miliseconds

#define LOW 0
#define HIGH 1
#define DUTY .5f

mraa_pwm_context pwm1;
mraa_pwm_context pwm2;
mraa_pwm_context pwm3;
mraa_pwm_context pwm4;

void send_preamble_sequence ( int preamble_length ) {
	int i = 0, j = 0;
	printf("\nSending preamble_length = %d ", preamble_length);

	for(i = preamble_length; i > 0; i--){

		mraa_pwm_write(pwm1,DUTY);
		mraa_pwm_write(pwm2,DUTY);
		mraa_pwm_write(pwm3,DUTY);
		mraa_pwm_write(pwm4,DUTY);
		for(j = PREAMBLE_DELAY; j > 0; j--);

		mraa_pwm_write(pwm1,0);
		mraa_pwm_write(pwm2,0);
		mraa_pwm_write(pwm3,0);
		mraa_pwm_write(pwm4,0);
		for(j = PREAMBLE_DELAY; j > 0; j--);
	}
}

void send_low_bit() {
	int i = 0;
	printf("0 ");
	mraa_pwm_write(pwm1,DUTY);
	mraa_pwm_write(pwm2,DUTY);
	mraa_pwm_write(pwm3,DUTY);
	mraa_pwm_write(pwm4,DUTY);
	for(i = SHORT_DELAY; i > 0; i--); // 5 ms
```

```
		mraa_pwm_write(pwm1,0);
		mraa_pwm_write(pwm2,0);
		mraa_pwm_write(pwm3,0);
		mraa_pwm_write(pwm4,0);
		for(i = LONG_DELAY; i > 0; i--);	// 20 ms
}

void send_high_bit() {
		int i = 0;
		printf("1 ");
		mraa_pwm_write(pwm1,DUTY);
		mraa_pwm_write(pwm2,DUTY);
		mraa_pwm_write(pwm3,DUTY);
		mraa_pwm_write(pwm4,DUTY);
		for(i = LONG_DELAY; i > 0; i--);	//20 ms

		mraa_pwm_write(pwm1,0);
		mraa_pwm_write(pwm2,0);
		mraa_pwm_write(pwm3,0);
		mraa_pwm_write(pwm4,0);
		for(i = SHORT_DELAY; i > 0; i--);	// 5 ms
}

int main( int argc, char *argv[] ) {
		int transmit_counter = 0;
		int i = 0;
		// GPIO Initialization - Edison has 4 PWM pins

		pwm1 = mraa_pwm_init(3);
		mraa_pwm_period_us(pwm1, 26);
		mraa_pwm_enable(pwm1, 1);

		pwm2 = mraa_pwm_init(5);
		mraa_pwm_period_us(pwm2, 26);
		mraa_pwm_enable(pwm2, 1);

		pwm3 = mraa_pwm_init(6);
		mraa_pwm_period_us(pwm3, 26);
		mraa_pwm_enable(pwm3, 1);

		pwm4 = mraa_pwm_init(9);
		mraa_pwm_period_us(pwm4, 26);
		mraa_pwm_enable(pwm4, 1);


		// if we pass in an argument, use it for the preamble_length, takes
		// priority over what's in the file. please pass in an integer
		if (argc == 2) {
				preamble_length = atoi(argv[1]);
				printf("Argument(1) Passed In! Preamble Length set to: %d\n", preamble_length);
		}
		else {
				printf("Preamble Length set to default value: %d\n", preamble_length);
		}
		// if we pass in an argument, use it for the edisonID, takes
		// priority over what's in the file. please pass in an integer
		if (argc == 3) {
				preamble_length = atoi(argv[1]);
```

```
        printf("Argument(1) Passed In! Preamble Length set to: %d\n", preamble_length) ;
        edisonID = atoi(argv[2]);
        printf("Argument(2) Passed In! EdisonID set to: %d\n", edisonID);
}
else {
        printf("edisonID set to default value: %d\n", edisonID);
}
// This will transmit IR data on all 4 pins at once
// on a single Edison board


// continuously Transmit IR Signal when the program is running
While ( 1 ) {

        // Preamble - Signals the Receiver Message Incoming
        send_preamble_sequence ( preamble_length );

        // Sending Edison Board ID # - 2 bits, MSB then LSB
        switch (edisonID) {
                case 0:
                        printf("EdisonID: 0 - ");
                        send_low_bit();    // Send lsb bit 0 = LOW
                        send_low_bit();    // Send msb bit 1 = LOW
                        break;
                case 1:
                        printf("EdisonID: 1 - ");
                        send_low_bit();    // Send msb bit 1 = LOW
                        send_high_bit();   // Send lsb bit 0 = HIGH
                        break;
                case 2:
                        printf("EdisonID: 2 - ");
                        send_high_bit();   // Send msb bit 1 =
                        send_low_bit();    // Send lsb bit 0 = LOW
                        break;
                case 3:
                        printf("EdisonID: 3 - ");
                        send_high_bit();   // Send lsb bit 0 = HIGH
                        send_high_bit();   // Send msb bit 1 = HIGH
                        break;
                default:
                        send_low_bit();    // Send lsb bit 0 = LOW
                        send_low_bit();    // Send msb bit 1 = LOW

        }

        // Sending Edison IR Emitter ID # - 2 bits, MSB then LSB

        // pwm1,DUTY = 00 = short-long/short-long = 5-20/5-20
        // pwm2,DUTY = 01 = short-long/long-short = 5-20/20-5
        // pwm3,DUTY = 10 = long-short/short-long = 20-5/5-20
        // pwm4,DUTY = 11 = long-short/long-short = 20-5/20-5

        // First Bit
        mraa_pwm_write(pwm1,DUTY);
        mraa_pwm_write(pwm2,DUTY);
        mraa_pwm_write(pwm3,DUTY);
        mraa_pwm_write(pwm4,DUTY);
        for(i = SHORT_DELAY; i > 0; i--); // 5 ms
```

```
        mraa_pwm_write(pwm1,0);
        mraa_pwm_write(pwm2,0);
        //mraa_pwm_write(pwm3,DUTY, DUTY);
        //mraa_pwm_write(pwm4,DUTY, DUTY);
        for(i = MID_DELAY; i > 0; i--); // 15 ms

        //mraa_pwm_write(pwm1, 0);
        //mraa_pwm_write(pwm2, 0);
        mraa_pwm_write(pwm3,0);
        mraa_pwm_write(pwm4,0);
        for(i = SHORT_DELAY; i > 0; i--); // 5 ms

        // Second Bit
        mraa_pwm_write(pwm1,DUTY);
        mraa_pwm_write(pwm2,DUTY);
        mraa_pwm_write(pwm3,DUTY);
        mraa_pwm_write(pwm4,DUTY);
        for(i = SHORT_DELAY; i > 0; i--); // 20 ms

        mraa_pwm_write(pwm1,0);
        //mraa_pwm_write(pwm2,DUTY, DUTY);
        mraa_pwm_write(pwm3,0);
        //mraa_pwm_write(pwm4,DUTY, DUTY);
        for(i = MID_DELAY; i > 0; i--); // 15 ms

        //mraa_pwm_write(pwm1, 0);
        mraa_pwm_write(pwm2,0);
        //mraa_pwm_write(pwm3, 0);
        mraa_pwm_write(pwm4,0);
        for(i = SHORT_DELAY; i > 0; i--); // 5 ms

    }// end while loop

    mraa_pwm_write (pwm1, 0);
    mraa_pwm_enable(pwm1, 0);
    mraa_pwm_write (pwm2, 0);
    mraa_pwm_enable(pwm2, 0);
    mraa_pwm_write (pwm3, 0);
    mraa_pwm_enable(pwm3, 0);
    mraa_pwm_write (pwm4, 0);
    mraa_pwm_enable(pwm4, 0);

    return 0;
}//end main
```

Appendix B : Receive Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mraa/pwm.h>
#include <mraa/gpio.h>

#define ARDUINO_GPIO_PIN   8
#define SCALING_FACTOR 5
#define SAMPLING_RATE 30000/SCALING_FACTOR
#define MAX_SAMPLING_RATE 10
int preamble_length = 5;
#define TOTAL_SAMPLES 3000

#define UPPERBOUND_PREAMBLE 24
#define LOWERBOUND_PREAMBLE 16

#define UPPERBOUND_LOWVALUE 15
#define LOWERBOUND_LOWVALUE 5

#define UPPERBOUND_HIGHVALUE 45
#define LOWERBOUND_HIGHVALUE 35

#define PREAMBLE_RELAXED_DETECTION_THRESHOLD 2
// this  indicates how many values from among 10 that
//are examined can be outside the range of 15 to 25 , but still we can go ahead with preamble detection
// For example if preamble_relaxed_detection = 1 , one value out of 10 can be 13 and still we detect
preamble

int location_detection_threshold = 1;
// this indicates number of matching locations for us to conclude that we detected the location correcly
// for example if LOCATION_DETECTION_THRESHOLD is set to 2 , then if we got 2 out of 5 locations as 13
, we
// can conclude that location was indeed 13
// getting 3 out of 5 locaitons as matching is rather a stringent criterion
// and will result in sending a zero location to the server

int main(int argc, char *argv[]) {

   // Initialization
   volatile unsigned int i = 0, j = 0, k = 0;
   volatile unsigned int num_values = 0;
   volatile unsigned int l = 0;
   volatile unsigned int received_bit_count = 0 ;
   mraa_gpio_context gpio;
   gpio = mraa_gpio_init(ARDUINO_GPIO_PIN);
   mraa_gpio_dir(gpio, MRAA_GPIO_IN);
   // Detect the length of the preamble we expect from this message
   // Defaults to 5, get passed in as the 1st argument to pwm_receive
   if (argc == 2) {
      preamble_length = atoi(argv[1]);
      printf("Argument(1) Passed In! Preamble Length set to: %d\n", preamble_length);
   }
   else {
      printf("Preamble Length set to default value: %d\n", preamble_length);
```

```
  }
  if (argc == 3) {
     preamble_length = atoi(argv[1]);
     printf("Argument(1) Passed In! Preamble Length set to: %d\n", preamble_length);

     location_detection_threshold = atoi(argv[2]);
     printf("Argument(2) Passed In! Location detection threshold set to: %d\n",
location_detection_threshold);
  }
  else {
     printf("Location detection threshold set to default value: %d\n", location_detection_threshold);
  }

 // Raw Data Sampling
  // Detect the value received as either 1 or 0
  // Store in raw_data[]
 // raw_data stores 0 or 1 dep on what was received
  int raw_data[TOTAL_SAMPLES] ;
  unsigned int samples_remaining = TOTAL_SAMPLES;
  volatile int irSig = 0 ;
  i = 0;
  while(samples_remaining > 0 ) {
     raw_data[i] = mraa_gpio_read(gpio);
     for (j = 0 ; j < SAMPLING_RATE ; j++);
     i++;
     samples_remaining--;
  }//end while loop
  i = 0;

 printf("Sampled Bits are : \n");
 for (i = 0; i < TOTAL_SAMPLES; i++) {
    printf("%d", raw_data[i]);
  }
  i = 0;
  // Consolidating the raw data
  // Count the sequence of 1 or 0 received in a row
  // Store in values[]

  unsigned int counting_high_or_low = raw_data[0];
  unsigned int high_duration = 0;
  unsigned int low_duration = 0;
  // values stores the number of consecutive 0's or the number of consecutive 1's
  unsigned int values[100];

  if (raw_data[0] == 0) {
     low_duration = 1;
  }
  else if (raw_data[0] == 1) {
     high_duration = 1;
  }
  int count = 1;
  while( count < TOTAL_SAMPLES )
  {

     if ( raw_data[count] == 0 )
     {
        if (counting_high_or_low == 1)
              {
```

```
            counting_high_or_low = 0;
            low_duration = 1;
            values[i] = high_duration;
            i++;
        }
        else
        {
            low_duration++;
        }
    }
    else if (raw_data[count] == 1)
    {
        if ( counting_high_or_low == 0 )
        {
            counting_high_or_low = 1;
            high_duration = 1;
            values[i] = low_duration;
            i++;
        }
        else
        {
            high_duration++;
        }
    }
    else {
        printf("\nraw_data[count] error\n");
        printf("count = %d\n", count);
        printf("raw_data[count] = %d\n", raw_data[count]);

    }
    count++;
}/*end of while loop */
num_values = i ;
// Inferring preamble and bits from values array
// Search values[] for sequences of preamble and 4 bits
// Store in all_received_bits[]
i = 0;
j = 0;
k = 0;
unsigned int fsm_state = 0 ;
unsigned int preamble_found = 1 ;
int bit = -1;
int preamble_relaxed_detection;
int all_received_bits[25];
while (i < num_values)
{
    preamble_relaxed_detection = 0 ;
    for (j = i; j < ( i + preamble_length*2 ); j++)
    {

        if (values[j] < LOWERBOUND_PREAMBLE || values[j] > UPPERBOUND_PREAMBLE)
        {
          if (values[j] > LOWERBOUND_PREAMBLE - 5 && values[j] < UPPERBOUND_PREAMBLE + 5 )
          {
              if ( preamble_relaxed_detection < PREAMBLE_RELAXED_DETECTION_THRESHOLD )
              {
                  preamble_relaxed_detection++ ;
                  continue ;
```

```
            }
          }
          preamble_found = 0;
          break;
        }
      }
      if (preamble_found == 1)
      {
        for(l = 0; l < preamble_length * 2; l ++) {
          i++;
        }
        for(k = 1 ; k <= 4 ; k++)
        {
          bit = -1;

          if( values[i]>=LOWERBOUND_HIGHVALUE && values[i] <= UPPERBOUND_HIGHVALUE &&
values[i+1] >=LOWERBOUND_LOWVALUE && values[i+1] <= UPPERBOUND_LOWVALUE )
          {
            bit = 1;
          }

          if( values[i]>=LOWERBOUND_LOWVALUE && values[i] <= UPPERBOUND_LOWVALUE &&
values[i+1] >=LOWERBOUND_HIGHVALUE && values[i+1] <= UPPERBOUND_HIGHVALUE)
          {
            bit = 0;
          }

          i += 2;
          all_received_bits[received_bit_count] = bit;
          received_bit_count++;
        }
      }
      else {
        i++;
        preamble_found = 1;
      }
    }

    int messages = received_bit_count/4;
   num_values = i ;
    // Bit Verifier
    // Inspect all_received_bits[]
    // Store in TEMP
    i = 0;
    j = 0;

    int valid_sequence = 1;
    int location[6] = {0, 0, 0, 0, 0, 0};
    int messages_to_process = 0;
    if (messages < 6) {
      messages_to_process = messages;
    }
    else {
      messages_to_process = 6;
    }
    // we process at most 6 messages, at minimum 0

    for(i = 0; i < messages_to_process * 4; i += 4) {
```

```
        for (j = 0; j < 4; j++) {
            if (all_received_bits[i+j] == -1)
            {
                valid_sequence = 0;
            }
        }
        if (valid_sequence == 1) {
            location[i/4] =  all_received_bits[i+3]* 1 + all_received_bits[i+2]* 2 + all_received_bits[i+1] * 4 +
all_received_bits[i+0] * 8;
            // locations go from 1-16
            location[i/4] = location[i/4] + 1;
        }
        else {
            printf("Received an invalid sequence\n");
            valid_sequence = 1;
        }
    }

    // If we find a total of location_detection_threshold number of matching
    // locations out of 5, return the location

    if (messages_to_process > 0) {

        int match_counter = 0;
        for(i = 0; i < messages_to_process; i++) {
            for (j = 0; j < 4; j++) {
                if (location[i] == location[j]) {
                    match_counter++;
                }
                if (match_counter >= location_detection_threshold && location[i] != 0 ) {
                    printf("\nReturning location %d\n", location[i]);
                    return location[i];
                }
            }
            match_counter = 0;
        }
    }
    printf("Not enough bits matched, Return 0\n");
    return 0;
}//end main
```