8 May 2017

BeepBeep: Teensy Audio Processing for Ranging

Alexis Burnight
Department of Electrical Engineering, University of Iowa, Iowa City IA

Abstract: The Teensy 3.2 development board was used with the Audio Adapter Board in attempts to reconstruct the BeepBeep algorithm in real time. The Audio Adapter Board was not compatible with the FreeRTOS, and therefore the audio component and real time component were implemented on separate Teensy boards as a proof of concept. Since the implementation occurred on separate devices there was no indication of how the real time implementation compared to the COTS BeepBeep program. However, this experiment added a great deal of understanding to the different components in a real time system and served as a lesson to practical programming for embedded systems.

Introduction

BeepBeep is a Microsoft protocol that allows for ranging and localization. The protocol relies upon the independent recording of high frequency audio signals to relate the transmission time to the distance or location of a device. Since each device records independently, this protocol reduces some of the synchronization errors found in other methods. BeepBeep was designed for use with commercial off the shelf (COTS) devices such as mobile devices with an accessible recorder and speaker. This allows for simple implementation into already built systems. The protocol had a maximum efficiency for ranging of 4.5 m with a maximum error of 0.6 cm. Since this system was implemented using COTS devices, the goal was to examine how the ranging would change if the system were implemented with a real time operating system (RTOS). This would allow for a more precise signal detection time that may increase the accuracy of the ranging. However, due to some limitations found in the compatibility of various components used in the experiment, the experiment was not fully implemented. Therefore, this paper discusses some of the methods of implementation attempted, the necessary design considerations for a real time system, and how future implementations could expand upon these errors.

BeepBeep Ranging Algorithm

The BeepBeep algorithm uses two devices exchanging high frequency linear chirp signals to determine the distance between them. The host device (device A) initializes transmission whereby both devices begin recording. After recording has begun, device A sends a linear chirp signal. Near a second later device B echoes the signal and both devices stop recording. Each device determines the location of both signals from their recording device. Device A determines the time elapse of the recording from its own signal and the echoed signal B and similarly, device B between signal A and its echoed signal response. Device B then communicates the time elapse, which is then used with equation 1 to determine the distance (in cm) between the two devices.

$$D = \frac{c}{2}\left[(t_{AB} - t_{AA}) - (t_{BB} - t_{BA})\right] + d_{AA} + d_{BB}$$

Equation 1: The distance between two devices is determined using the time elapsed between the independent recording of two chirps and the distance between the devices recorder and speaker. For example, the recording of the chirp from device A by device B (t_{BA}) is used with the speed of sound to determine the distance of that wave offset by the distance between device B's recorder and speaker (d_{BB}).

The elapsed time is more accurate on the individual devices since it is determine by counting the amount of clock cycles between the signals and dividing by the clock frequency of the device. This eliminates any need for clock synchronization and the error induced by small fluctuations in clock speed.

Signal Design and Recognition

The linear chirp signal sent from the devices consists of a 2 kHz to 8 kHz, 50 ms signal sampled at a 44.1 kHz frequency. The efficiency of the signal is increased from a 2 kHz warm up wave. The signal from the recorder is determined by correlating the 2205-point reference chirp and a 2205-point sliding window of the incoming recorder data. The correlation is analyzed to determine if there is a sufficient spike in the correlation data corresponding to the chirp signal. This signal location is determined by taking the ratio of the normalized L2-norm of a 100-point sliding window with respect to the normalized L2-norm of the 4410-point correlation signal. When this signal strength exceeds a predetermined threshold (based upon environmental factors), the signal location has been found. Since the potential for multiple pathway may effect the determination of the signal location, the threshold is set such that only the first signal determination is used. This is the reason for the 1-second delay between the sending of the chirp from device A and device B. See the experimental methods section for visual aids on the chirp signal and correlation signal.

Experimental Methods

The BeepBeep algorithm was first simulated in python. The NumPy python library was used to store the reference signal and a pseudo-random noise generated signal. The library also provided tools for calculating the signal normalization, and the L2-norm of the correlation. The SciKit Learn python module was used to generate the chirp signal and calculate the correlation between two signals (see Appendix 1 for full script). Lastly, the MatLab interface was used to gather a visual representation of the two signals and the correlation (see Figure 1 and Figure 2).

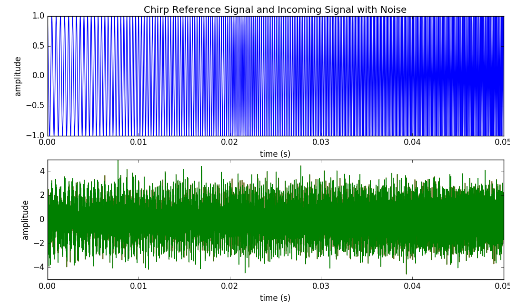

Figure 1: The linear chirp reference signal is shown on the top, while the noise generated signal is shown on the bottom. These signals were used to show how the BeepBeep algorithm determines the incoming signals location.

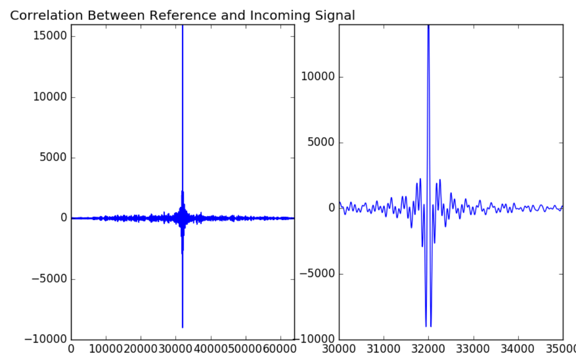

Figure 2: The correlation of the signals from Figure 1 is shown on the right. The left image shows the zoomed in portion where the spike in the correlation would correspond to the signal location.

The signal was accurately determined for a range of values (approximately 100 sample points). Therefore, many design options could be implemented here to increase the ranging accuracy such as averaging the window for which the signal was detected. However, the first instance the signal was recognized was used in this experiment. After using python to understand how the algorithm would be implemented on an embedded device, the Teensy was used to begin implementing the protocol in real time.

The Teensy 3.2 by PJRC was used as the development board for the project consisting of a 76 MHz clock MK20DX256 32-Bit ARM M4 Cortex. For the audio portion, this experiment used an Audio Adapter Board by PRJC specifically designed to sit on top of the Teensy 3.x boards, consisting of several line-in and line-out options, an SD card slot, a microphone, and a SGTL5000 audio codec chip. PRJC also supplies an Audio System Design (ASD) Tool to help configure the connections to and from the input/output signals (see Figure 3).

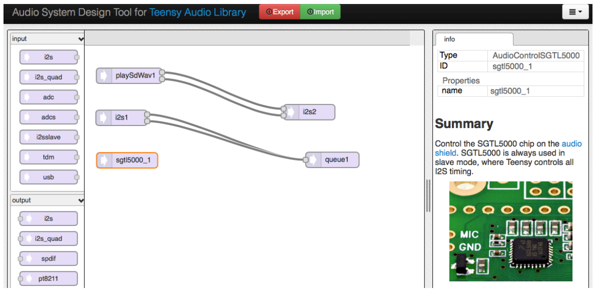

Figure 3: The Audio System Design tool was used to create the connections from the microphone to the recording queue and the SD card to the line-out (used with speaker or headphones). The SGTL500 must be included to controller the various aspects such as beginning recording or playing the chirp SD file.

The Teensy boards are specifically designed for compatibility with the Arduino development environment. Although C development can be used, the SDK for the Kinetis processor is sometimes difficult to work with. Therefore, the BeepBeep protocol was implemented with the Arduino IDE and programmed using the Teensyduino Teensy Bootloader software provided by PJRC.

The FreeRTOS real time operating system was used as the operating system on the Teensy. Bill Greiman ported the FreeRTOS library for use with AVR and ARM cortexes. The arduino library imports the various C components of the FreeRTOS library, incorporating a wrapper that allows for smooth development with Arduino. Furthermore, the syntax for the different components (i.e. tasks, queues, etc.) follows the same syntax as the FreeRTOS manual and a set of example programs to allow for easy and seamless implementation.

The BeepBeep protocol was implemented to take advantage of as many of the RTOS features as possible. The FreeRTOS free online manual, “Mastering the FreeRTOSTM Real Time Kernel” was used to help understand the various components of a real time system. The protocol was designed to use 5 tasks for different specific functionality. The general flow of the system is shown in Figure 4. This implementation allows for various areas of flexibility by separating each individual function into different tasks. For example, the task that established and communicates between the two devices was implemented as serial communication but could later be updated to Bluetooth or WiFi communication for increase portability between the devices.

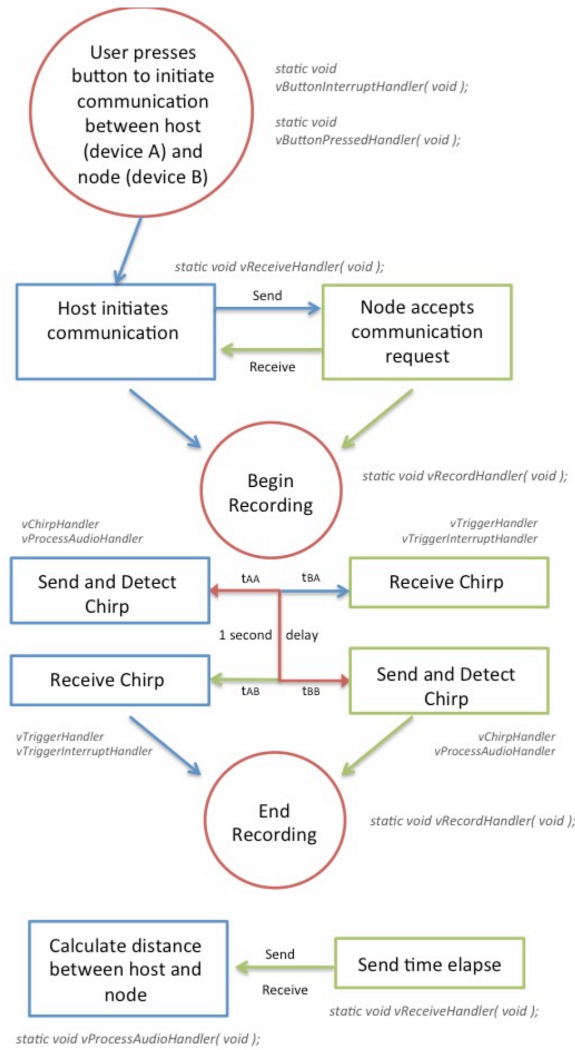

Figure 4: Flow diagram for the various tasks and states of the BeepBeep RTOS.

The 5 main functions handled by the task scheduler responded to the following parts of the system: a button press for the user starting the ranging sequence (vButtonHandler), communication between two devices (vReceiveHandler), the recording object (vRecordHandler), the chirp signal object (vChirpHandler), and the audio processing to determine the signal location (vProcessAudioHandler). Unfortunately due to the incompatibility of the Audio Board and the FreeRTOS library (see Results for more information) the program was implemented using two devices and communicated with a computer via serial communication (see Appendix A2 for program). A second Teensy with the Audio Board was used to send the chirp, record the incoming chirp, and process the recorded data. Each time a new recorded signal was processed a trigger was sent to the original device for a time stamp. Once the chirp signal was located, a second trigger sent to the host device notified the device to save the previous time stamp and use it for determining the elapsed time. This added one more function, vTriggerHandler. The various states of the program for the host device are shown in Table 1 and correspond to the timing diagram in Figure 5.

No.	Task	Function
1	Idle task	Wait for user to initiate ranging
2	vButtonInterrupHandler	Button pressed, notify handler xButtonPressed
3	vButtonPressedHandler	Receive task notification and take binary mutex so only one ranging request is processed at a time. Notify the send and receive task xReceive. Block until task notified again.
4	vReceiveHandler	Take task notification, initiate host node communication, and block until response received
5	Idle Task	All tasks blocked
6	vReceiveHandler	Receive communication from node into queue and notify device recorder. Block until notified again.
7	vRecordHandler	Take task notification and begin recording (trigger to recording Teensy). Notify chirp task (reverse order for node). Block until notified again.
8	vChirpHandler	Take task notification and send audio signal (trigger to audio Teensy). Notify xProcess and block until next time a ranging sequence occurs.
9	vProcessAudioHandler	Create trigger task for receiving audio signal detection. Block until data is available on xTimeQueue
10	Idle Task	Wait for trigger to occur.
11	vTriggerInterruptHandler	Place current time xGetTickCount into queue of size 2 xTimeQueue
12	vProcessAudioHandler	Take time stamp from xTimeQueue and save as t_{AA}. Block until data is in queue.
13	Idle Task	Wait for second trigger to occur.
14	vTriggerInterruptHandler	Place current time xGetTickCount into queue of size 2
15	vProcesAudioHandler	Take time stamp from xTimeQueue and save as t_{AA}. Block until data is in queue.
16	vReceiveHandler, vRecordHandler	Wait to receive elapsed time from the node device and stop recording.
17	vProcessAudioHandler	Wait for xReceiveQueue to contain elapsed time and calculate range. Kill vTriggerInterruptHandler. Notify xButton.
18	vButtonPressedHandler	Allow user to press button again.
19	Idle Task	Block all tasks until button pressed.

Table 1: Description of the different tasks with their relation to the timing diagram shown in Figure 5.

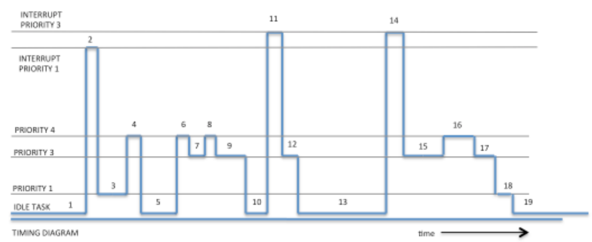

Figure 5: General timing diagram for the BeepBeep RTOS system design. See Table 1 for numerical descriptions.

Results and Discussion

The audio adapter board was not compatible with the FreeRTOS library. When the two were used simultaneously, the operating system would have a higher priority than the audio adapter board and the board would stop working. This was an unfortunate discovery in the development of the system. The original system design used Bluetooth to communicate between two separate Teensy devices. Since the second Teensy was used to control the audio signal output and recording, this could not be accomplished. The cause of the incompatibility is not fully understood but is more likely the cause of shared SPI communication pins. One forum post claimed that both of the pins used the same IRQ (interrupt request). However, the documentation on the pin set up for the FreeRTOS ARM Arduino library made it difficult to locate which pins were being used. Therefore, all attempts to change the used SPI pins for the audio board did not fix the problem.

By the time this issue was discovered, the project had already been developed as two halves: the audio processing half and the real time component.

Attempts to circumvent this problem resulted in overly complicated circuits, such as creating two separate speaker and microphone circuits that could, or infeasible options, such as forcefully stopping the scheduler. The only viable work around was to use two devices to complete the individual parts, again the audio processing and the real time system. Using this option, the second Teensy would trigger the first Teensy when it detected an audio signal. This option could not be fully implemented due to time constraints. Even so, this option would not have been a viable comparison for the BeepBeep COTS device implementation. The limitations were found both in the implementation of two separate devices and in the limitations of the audio adapter board itself. The audio adapter board only allowed for one process at a time, recording input signals or outputting a signal to a speaker. Therefore, the device would not have been usable to accurately determine t_{AA} or t_{BB} unless a sufficient distance separated the speaker and microphone such that the delay between the speaker and the microphone was greater than the switching time between the two. Furthermore, the connection between the two boards relied upon triggers. This adds latency between the actual signal and the signal registered by the host Teensy, unaccounted for by Equation 1. In order to remove the latency with the current set up, the audio processing would have to occur entirely on the secondary Teensy eliminating the major real time components. Since this project was focused on gaining an understanding of real time systems, this design was not chosen.

Even though the system as a whole did not work, the individual components did. The secondary Teensy was able to accurately interpret a chirp signal when the threshold for the L2-norm ratio was set to 3.5. Furthermore, the various RTOS components implemented worked without flaw. The software successfully implemented tasks, queues, notifications, interrupts, and semaphores. A greater understanding of the basic fundamentals of RTOS systems were greatly enhanced when making different design considerations such as when to use a semaphore, mutex, queue, or task notification to wake a task or how to determine the priority of tasks running at the same time. Even though the real time system was implemented to exemplify an understanding of the fundamental principles, there was room for improvement on both the secondary Teensy program and the host Teensy program. The secondary Teensy continually searched for a chirp. This should have been modified such that only the first chirp was recognized then the system paused. The reason for continuous signal processing was because the audio and recorder could not be used simultaneously. Secondly, the real time component on the host could be more effective if timing constraints and requirements were implemented to detect if the state of the system was unknown or an error may have occurred.

Conclusion

Although this experiment was not as successful as one would have hoped, it gave insight to the many design considerations for a real time system. Furthermore, it helped give a practical understanding of how to implement embedded systems. Unlike other types of system development, embedded systems need to be implemented in both an incremental and combinatorial manner. Since the implementation depends upon delicate components, small functions and portions should be tested individually before incorporating them into the design. Although this was done within the project, the major components of the design were never implemented together. Becoming aware of the need to test large components together, early and often, is a lesson that is irreplaceable. This experiment will serve two great purposes—an understanding of designing RTOS systems and an understanding of the best implementation practices—that will help enhances all future projects and experiments.

References

Peng, C.; Shen, G.; Zhang, Y.; Li, Y.; Tan, K. In *Beepbeep: a high accuracy acoustic ranging system using cots mobile devices*, Proceedings of the 5th international conference on Embedded networked sensor systems, ACM: 2007; pp 1-14

Appendix

A1. Python Script

```
#detected 31930 to 32050

import numpy as np
from numpy.linalg import norm
from scipy.signal import chirp, correlate
from sklearn.preprocessing import normalize
import matplotlib.pyplot as plt

def ranging(A, B, c, d_A_A, d_B_B):
    return c/2*(A-B) + d_B_B + d_A_A

#Generate chirp signal
t = np.linspace(0,.05, 32000)
w = chirp(t, f0=2000, f1=8000, t1=.05,
    method='linear')
noise = np.random.normal(0,1,32000)
y = noise
y = noise + w
plt.figure(1)
plt.subplot(211)
plt.xlabel('time (s)')
plt.ylabel('amplitude')
plt.axis([0,0.05,-1,1])
plt.title('Chirp Reference Signal and Incoming
    Signal with Noise')
plt.plot(t,w, 'b')
plt.subplot(212)
plt.xlabel('time (s)')
plt.ylabel('amplitude')
plt.axis([0,0.05,-5,5])
plt.plot(t,y, 'r')
plt.plot(t,w,t,y)
plt.show()

z = correlate(y, w)
t2 = np.linspace(0, 1, 64000)
t3 = np.linspace(32000,1, 35000)
plt.subplot(131)
plt.title('Correlation Between Reference and
    Incoming Signal')
plt.axis([0,64000,-10000,16000])
plt.plot(z)
plt.subplot(132)
#plt.title('Correlation Between Reference and
    Incoming Signal')
plt.axis([30000,35000,-10000,14000])
plt.plot(z)
plt.show()

t_Hsd = 20 #cross correlation ratio for signal
    detection
w0 = 100 #number of samples within a small
    window for cross correlational value
N = 2250 #number of sample points in the signal

max = 0
#for i in range(0,len(z)-N):
for i in range(30000,34000, 50):
    tmp = z[i:i+N]
   l_n = norm(tmp)/N #calculate the L2 Norm ??
    Compute around axis
    j = 0
    for j in range(0, len(tmp)-w0):
        tmp2 = tmp[j:j+w0]
        l_s = norm(tmp2)/w0 #calulate the L2
    Norm of a subsample
        if (l_s/l_n > t_Hsd):
                if (l_s/l_n > max):
                        max = l_s/l_n
                print('Signal detected at ' + str(i))
print('Maximumum signal ratio: ' + str(max))

t_A1 = 12 #tick count for signal sent
t_A3 = 14 #tick count for signal received
freq_A = 44000 #Sampling frequency of node A
    in Hz

t_B1 = 54
t_B3 = 52
freq_B = 44000 #Sampling frequency of node B
    in Hz

d_A_A = 1.2 #distance between the microphone
    and the speaker for A, known value in cm
d_B_B = 1.2 #distance between the microphone
    and the speaker for B, known value in cm

signal_A = (t_A3-t_A1)/freq_A
signal_B = (t_B3-t_B1)/freq_B

c = 340 #speed of sound constant (temperature
    and humidity dependent)

print("Range: %.1f cm" % ranging(signal_A,
    signal_B, c, d_A_A, d_B_B))
```

A2. Python BeepBeep Terminal Output

```
wireless-nat-inside:RTOS Project
    alexisburnight$ python beepbeep.py
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 31950
Signal detected at 32000
Signal detected at 32000
Signal detected at 32000
Signal detected at 32000
Signal detected at 32050
Maximumum signal ratio: 21.4009458745
Range: 172.4 cm
```

A3. Arduino BeepBeepHost File

```
#include <FreeRTOS_ARM.h>
#include "basic_io_arm.h"
#include <Audio.h>
#include <Wire.h>
#include <SPI.h>
#include <SD.h>
#include <SerialFlash.h>
#include "chirp.h"

#undef F
#define F(str) str

//const uint8_t LED_PIN = 13;
const uint8_t BUTTON_PIN = 16;
const uint8_t TRIGGER_CHIRP_PIN = 2;
//const uint8_t TIME_STAMP_PIN = 3;
const uint8_t TRIGGER_PIN = 4;
const uint16_t STACK_DEPTH = 200;
const uint16_t SPEED_OF_SOUND = 34.29;
const uint8_t STEP = 128;

const uint8_t d_Host = 3;
const uint8_t d_Node = 3;
uint8_t dt_Node = 621;



//--------------------------------------
//    Audio Processing Constants
//--------------------------------------
const uint16_t CHIRP_LENGTH = 2206;
const uint8_t W0_LENGTH = 100;
const uint16_t samplingFrequency = 44100;

int32_t timer_id;
boolean timer_started;


volatile uint32_t count = 0;

static void vButtonInterruptHandler( void );
static void vButtonPressedHandler( void );
static void vReceiveHandler( void );
static void vRecordHandler( void );
static void vProcessAudioHandler( void );
static void vTriggerInterruptHandler( void );
static void vTriggerHandler( void );
static void vTimerCallback( void );
static void vChirpHandler( void );

volatile uint16_t audioBuffer[CHIRP_LENGTH];

SemaphoreHandle_t xButtonSemaphore;
```

```
TaskHandle_t xBluetoothReceive;
TaskHandle_t xRecordBegin;
TaskHandle_t xChirp;
TaskHandle_t xProcess;
TaskHandle_t xButtonPressed;
QueueHandle_t xTimeQueue;
QueueHandle_t xReceiveQueue;
TaskHandle_t xTimeStamp;
TaskHandle_t xTrigger;
TimerHandle_t xTimer;
TickType_t xTime;


void setup() {
  // put your setup code here, to run once:
  Serial.begin(115200);
  while (!Serial) {} //Wait for serial to connect
  Serial.println("Serial Communication
     Established");

  xButtonSemaphore =
     xSemaphoreCreateMutex();
  xTimeQueue = xQueueCreate(4,
     sizeof(TickType_t));
  xReceiveQueue = xQueueCreate(1, sizeof(int));

  if (xTimeQueue == NULL) {
    Serial.println("failure");
  }

  // successfully created
  if (xButtonSemaphore != NULL) {
    xTaskCreate(vButtonPressedHandler, "Button
     Press Handler",
            STACK_DEPTH, NULL, //
     pvParameters
            1, //priority
            &xButtonPressed //replace if need task
     handler
           );
    /*
      xTaskCreate(vTriggerHandler, "Trigger
      Time",
      STACK_DEPTH, NULL,
      3,
      &xTrigger
      );
    */
    //xTimer = xTimerCreate("Signal Timer", 1,
     pdTRUE, NULL, NULL);

    xTaskCreate(vReceiveHandler, "Bluetooth
      Receive Handler",
            STACK_DEPTH, NULL,
            4,
            &xBluetoothReceive
           );

    xTaskCreate(vRecordHandler, "Recording
      Device Handler",
            STACK_DEPTH, NULL,
            3,
            &xRecordBegin
           );

    xTaskCreate(vChirpHandler, "Sending Chirp
      Handler",
            STACK_DEPTH, NULL,
            4,
            &xChirp
           );

    xTaskCreate(vProcessAudioHandler, "Audio
      Processing",
            STACK_DEPTH, NULL,
            3,
            &xProcess
           );

    pinMode(BUTTON_PIN, INPUT);
    attachInterrupt(BUTTON_PIN,
      vButtonInterruptHandler, RISING); //update
      to falling later

    pinMode(TRIGGER_CHIRP_PIN, OUTPUT);
    digitalWrite(TRIGGER_CHIRP_PIN, LOW);
    /*
       pinMode(TRIGGER_PIN, INPUT);
       attachInterrupt(TRIGGER_PIN,
      vTriggerInterruptHandler, RISING); //update
      to falling later
    */
    vTaskStartScheduler();

  }
  for (;;);
}

static void vButtonInterruptHandler( void ) {
  static signed portBASE_TYPE
     xHigherPriorityTaskWoken = pdFALSE;
```

```
  //update to notification
  xTaskNotifyFromISR( xButtonPressed, NULL,
      eNoAction, (signed
      portBASE_TYPE*)&xHigherPriorityTaskW
      oken );
  //if context switch is required

      portEND_SWITCHING_ISR( xHigherPriorit
      yTaskWoken);
}

static void vTriggerInterruptHandler( void ) {
  xTime = xTaskGetTickCount();
  //taskDISABLE_INTERRUPTS();
  static signed portBASE_TYPE
      xHigherPriorityTaskWoken = pdFALSE;
  //update to notification
  xQueueSendToBackFromISR(xTimeQueue,
      &xTime, (signed
      portBASE_TYPE*)&xHigherPriorityTaskW
      oken);
  //xTaskNotifyFromISR( xTrigger, NULL,
      eNoAction, (signed
      portBASE_TYPE*)&xHigherPriorityTaskW
      oken );
  //if context switch is required

      portEND_SWITCHING_ISR( xHigherPriorit
      yTaskWoken);
}

static void vButtonPressedHandler( void
      *pvParameters ) {
  for (;;) {
    //update such that there is a timeout
    ulTaskNotifyTake( pdTRUE,
      portMAX_DELAY );
    xSemaphoreTake( xButtonSemaphore,
      portMAX_DELAY );
    Serial.println("Button press noted");
    xTaskNotifyGive( xBluetoothReceive );
    ulTaskNotifyTake(pdTRUE,
      portMAX_DELAY);
    xSemaphoreGive( xButtonSemaphore );
  }
}

static void vReceiveHandler( void
      *pvParameters ) {
    ulTaskNotifyTake(pdTRUE, //clear on exit
                portMAX_DELAY );
```

```
//if(initiate) {
Serial.println("Pairing....");
Serial.println("Paired");
//} else {
//  Serial.println("Start signal was already
  established");
//}
//--------------Start Recording--------------
Serial.println("Waiting to receive serial data");
while (!Serial.available()) {}
char x;
while (Serial.available() > 0) {
  Serial.println(Serial.available());
  x = Serial.read();
}
Serial.print("Received: ");
Serial.println(x);
if (x == 's') {
  Serial.println("Sending signal to node
   device...");
  while (!Serial.available()) {}
  while (Serial.available() > 0) {
    x = Serial.read();
  }
  if (x == 'r') {
    Serial.println("Received signal from node
   device, begin recording...");
    xTaskNotifyGive( xRecordBegin );
  }
  ulTaskNotifyTake(pdTRUE, //clear on exit
               portMAX_DELAY );
}
ulTaskNotifyTake(pdTRUE,
  portMAX_DELAY);
int elapsed_time;
Serial.println("How long between chirps?");
  while (!Serial.available()) {}
  char s = Serial.read();
  int t = (s-'0');
  Serial.println(s);
  elapsed_time = 100*t;
  while (!Serial.available()) {}
  s = Serial.read();
  t = s-'0';
  Serial.println(t);
  elapsed_time += 10*(t);
  while (!Serial.available()) {}
  s = Serial.read();
  t = s-'0';
  elapsed_time += 1*(t);
```

```
    Serial.println(elapsed_time);

    //elapsed_time
    //xQueueSendToFront(xReceiveQueue,
      elapsed_time, 0);
  }

static void vRecordHandler( void *pvParameters )
      {
  for (;;) {
    ulTaskNotifyTake(pdTRUE, //clear on exit
                 portMAX_DELAY );
    vTaskDelay(pdMS_TO_TICKS(10));

    xTaskNotifyGive( xChirp );
    ulTaskNotifyTake(pdTRUE, //clear on exit
                 portMAX_DELAY );
    Serial.println("Recording device in use");
    //if buffer is full
    xTaskNotifyGive( xProcess );
    xTaskNotifyGive(xBluetoothReceive);
    ulTaskNotifyTake(pdTRUE, //clear on exit
                 portMAX_DELAY );
    Serial.println("Done recording");

    //Serial.println("I guess we have processed the
      data");

  }
}

static void vChirpHandler( void *pvParameters )
      {
  for (;;) {
    ulTaskNotifyTake(pdTRUE, //clear on exit
                 portMAX_DELAY );
    //--------------------------FIGURE OUT HOW
      TO ENTER CRITICAL STATE
    Serial.println("Chirping...");
    digitalWrite(TRIGGER_CHIRP_PIN, HIGH);
    vTaskDelay(pdMS_TO_TICKS(1));
    digitalWrite(TRIGGER_CHIRP_PIN, LOW);
    //queue1.begin();
    //vTaskDelay(pdMS_TO_TICKS(50));
    Serial.println("Done playing");
    Serial.println("Done chirping");
    xTaskNotifyGive(xRecordBegin);
  }
}

static void vProcessAudioHandler( void
      *pvParameters) {
  for (;;) {
    TickType_t *t_A;
    TickType_t *t_B;
    ulTaskNotifyTake(pdTRUE,
      portMAX_DELAY);
    //float norm = l2_norm(
    /*for(uint16_t i = 0; i < CHIRP_LENGTH; i
      +=STEP){
     Serial.print("Value of i: ");
     Serial.println(i);
     }*/
    pinMode(TRIGGER_PIN, INPUT);
    attachInterrupt(TRIGGER_PIN,
      vTriggerInterruptHandler, RISING); //update
      to falling later
    xTaskCreate(vTriggerHandler, "Trigger Time",
             STACK_DEPTH, NULL,
             3,
             &xTrigger
            );
    Serial.println("Created Trigger Task");
    xQueueReceive(xTimeQueue, &t_A,
      portMAX_DELAY);
    Serial.println("T_A received");
    vTaskDelay(pdMS_TO_TICKS(10));
    //taskENABLE_INTERRUPTS();
    xQueueReceive(xTimeQueue, &t_B,
      portMAX_DELAY);
    Serial.println("T_B received");
    TickType_t dt_host = t_B-t_A;
    Serial.println(dt_host);

    vTaskDelay(pdMS_TO_TICKS(10));
    xTaskNotifyGive( xRecordBegin );
    //xTaskNotifyGive(xBluetoothReceive);
    //xQueueReceive(xReceiveQueue, &dt_Node,
      portMAX_DELAY);
    Serial.println(distance(dt_host, dt_Node,
      d_Host, d_Node));
    xTaskNotifyGive( xButtonPressed );
  }
}


static void vTriggerHandler( void *pvParameters)
      {
  for (;;) {
    ulTaskNotifyTake( pdTRUE,
      portMAX_DELAY );
    TickType_t xTriggerTime;
```

```
    xTriggerTime = xTaskGetTickCount();
    Serial.println(xTriggerTime);
    Serial.println("Trigger Received");
    //xQueueSendToFront(xTimeData,
      xTriggerTime, 0);
  }
}

//--------------------------------------------------------
      --------------------
//                  Mathematical Processing
      Functions
//
//--------------------------------------------------------
      --------------------

static float distance( float dt_Host, float dt_Node ,
      float d_Host, float d_Node) {
  // value calculated in cm
  return (SPEED_OF_SOUND / 2 * (dt_Host -
      dt_Node)) + ((d_Host + d_Node) / 2);
}

void loop() {
} //should never reach here
```