University of Illinois at
Urbana – Champaign

ECE 445

# Individual Progress Report

Team #14

Child Development Sensor

Eu Hong Woo (ewoo5)

# Contents

# Introduction

This project is separated into 2 main components --- the sensor node and the main hub. The sensor node will contain the ECG sensor, microphone and a Bluetooth module to communicate with the main hub. The main hub collects data from the main hub with a Bluetooth module, process it with a microcontroller, then uploads the data to an AWS server. We have so far separated our tasks in this manner: Xiang Wen has taken a few power electronics classes, so he will design the voltage regulators, and the ACDC converter on the main hub; Yang An will deal with the Bluetooth and WiFi communication protocols, AWS server, as well as the booting of the microcontroller on the main hub; I am tasked to learn how to use the ECG and microphone modules, as well as the communication protocols between the Bluetooth/WiFi modules and the microcontroller within the main hub. After we are done with each of our tasks, we will all work on the PCB design to integrate everything together.

Since Yang An and I are both working mainly on writing code for communication protocols, we will both work very closely so that our codes will be able to integrate with each other.

# Design and Verification

## ECG Module

Although we have mentioned in our design review that we will be using the BMD101 module, we have realized during our design process that obtaining an ECG module from Sparkfun was a better choice for a few reasons. The first reason being BMD101 costs more than an ECG module from Sparkfun. Secondly, modules designed by Sparkfun usually has more community support on the internet. The BMD101 module has more built in filtering components and an SOC which would give cleaner ECG readings, but given the time constraint, we will need to opt for the option which gives us more open source references so that we can always ask for help if we have any struggles during our design process. Also, since the BMD101 is relatively niche, the delivery time from our purchase prevents us from starting the design early.

The module we have gotten from Sparkfun is the AD8323 Heart Rate Monitor module. It outputs analog signals which basically is the measure of the amplified voltage amplitude. To make these values processible by our microcontrollers, we will use an ADC module. The ADC module we used is the ADS1115 module. This ADC module outputs using an I2C peripheral, so we will be getting 16-bit samples of data, with programmable sampling rate. This ADC module can have a maximum sampling rate of 860 samples/s [1]. However, we will be using it at mid-range, 500 samples/s since 400-500 samples/s is typically used in modern ECGs. If we find that it doesn't really work well, we can also lower it down to around 150 samples/s which is still plentiful since we are only using a 3-lead ECG.

Connecting these output onto the microcontroller's I2C interface, we were able to collect the following ECG waveform. The SNR ratio seems to be quite high, but it seems to suffice for our needs. If we compare our waveform to an ideal graph shown below, we can clearly see the R peaks as circled in red which is our main objective.

Furthermore, there two extra pins on the AD8232 module: LO+ and LO-. The module is able to sense if the electrodes are tightly attached to the user. If they are not attached properly, LO+ and LO- will both be pulled high which allows us to check this pins to see if any noisy data can be discarded.


Figure 1: ECG waveform with R-peaks circled in red.

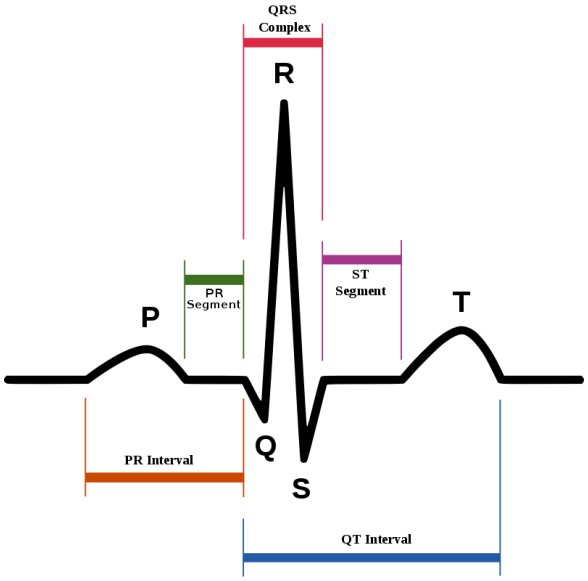



Figure 2: An idealized ECG waveform with the main peaks labeled. [2]

## Verification

The signals we have gotten so far can definitely be improved so that the SNR can be increased to around 60dB-A. We can verify this by passing this to a Matlab script which calculates SNR. Besides this, our main objective, as we have discussed with the professors funding this project, is to be able to import these ECG text files into the CardioEdit or QRSbrowser software, and have it automatically detect the R-peaks and graph the IBI (Inter-Beat Interval). We will improve the filters and plan to meet with Prof. Nancy L McElwain and Prof. Laura J Wright to verify that these software are detecting the correct R-peaks before November 23rd.

## Microphone & Audio Recording

For the microphone used for audio recording, we stuck with the initial module mentioned during our design review --- the SPH0645LM4H-B MEMs microphone. This microphone module is very small, has built-in low pass filter, only costs around $7, and provides a range of between 50 Hz to 15 kHz, which is plentiful for our needs. While this module outputs directly in digital form, which is a convenience as we

get to bypass some design steps, it remains as the most challenging design aspect of this since we are not familiar with the I2S protocol.

The I2S peripheral is commonly used to communicate between audio devices as it requires less handshaking and allows higher data transfer rate. In an I2S bus, there can be many receivers but only one transmitter, unlike an I2C bus which allows several of both. In our case, since we only plan to include one microphone on the on the sensor node, our setup will simply be a single transmitter and a single receiver. To lower design costs, we will not purchase a microcontroller with a built-in I2S port, instead, we will use the I2S ports available on the ESP32 Bluetooth module.

There are 3 ports in an I2S peripheral: clock (SCLK), word select/left-right select (LRCK), data (SD). To determine whether these ports are input or output in each of the transmitter and receiver side, we will need to set which is the master and slave. Since we want the receiver (the Bluetooth module) to determine the sampling rate of the microphone, we will set it as the master. This way, the clock signal will be set by the Bluetooth module (together with the word select), and the data (SD) will be an input to the Bluetooth module, just as shown in the Figure below:

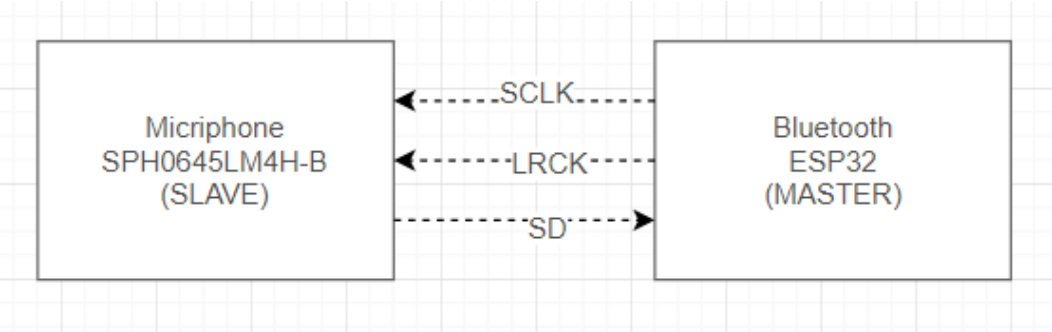



Figure 3: I2S master-slave configuration.

The roles of each ports can be explained with the timing diagram below. The LRCK signal is to differentiate between the right and left channels. When it is pulled high, the slave will only transmit samples from the right channel, and vice versa. As for SD, data is sent in MSB format, meaning that data will always be aligned to the left (MSB) regardless of any discrepancies between the bit-depth of the microphone and the data length of the I2S. For our microphone, the bit-depth is only 18 bits, so whatever data length is transmitted, only the 18 MSBs will be actual data (the rest is either zero padded or left as meaningless values). This makes things easier for us since we don't really need to match the sample length configurations on each side of the I2S bus. For the SCLK, its minimum frequency can be determined by the following formula:

$$SCLK = \#\ of\ channels\ \times sampling\ rate\ \times sample\ length$$

Although we only plan on recording mono audio, we can leave the number of channels as 2 since we can simply add another microphone module if we decide otherwise in the future. For testing, we use a sampling rate of 8 kHz, and sample length is simply the bit depth (18 bits). So, this gives us SCLK = 288 kHz. The microphone module only enters active mode if the SCLK is above 900 kHz, hence, we can use any frequency above 900 kHz. We can produce precise clock rates using a PLL provided in the ESP32 module.

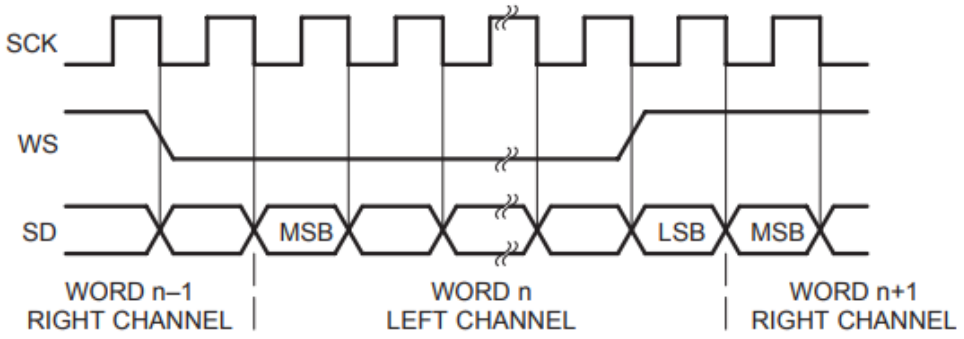



Figure 4: I2S protocol timing diagram. [3]

The pin configurations of on each of these modules can be simplified as shown below. In order to tell the microphone module whether it belongs to the left or right channel, we need to set voltage at the SEL pin. If SEL is driven low, it will transmit data when LRCK is driven low (corresponding to left channel). Arbitrarily, we will drive this pin low (set as left channel).

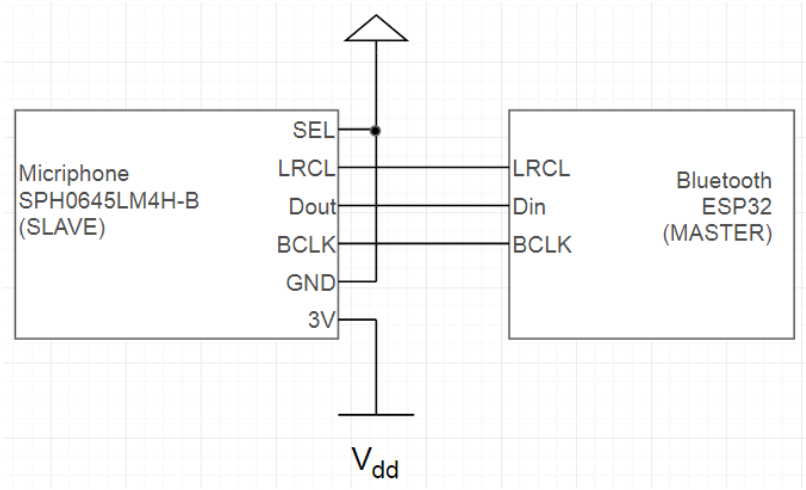



Figure 5: Pin connections used for our configuration.

After understanding how it works physically, we needed to deal with software aspect. To receive data from the ESP32 Bluetooth module, we will use the library support provided by Espressif themselves (the chip manufacturer) to setup a toolchain, compile and flash programs onto the board.

To configure the I2S ports on the ESP32 board, we will need to set the bitmasks on the *i2s_config_t* structure. There are a few things to note:

- There are two I2S ports on the ESP32 board, defined by the constants I2S_NUM0 and I2S_NUM1. We will only use I2S_NUM0.
- Communication format is the typical linear PCM instead of any logarithmic or compressed formats.
- From a few tests we have run, configuring the sample length to the exact 24-bit as specified in the SPH0645LM4H-B datasheet doesn't work. Luckily for us, data is sent MSB aligned, so we can have mismatching data length configuration on the ESP32 module, and we found out that 32-bit works somehow.

- We are not sure what PICs are used for the interrupts, so we will just leave it as the default value.
- The ESP32 module uses DMA buffers for transmitting and receiving, so we will choose to use a buffer length of 64, and a total of 8 buffers.

```
i2s_config_t i2s_config = {
    .mode = I2S_MODE_MASTER | I2S_MODE_RX,
    .sample_rate = SAMPLE_RATE,
    .bits_per_sample = 32,
    .channel_format = I2S_CHANNEL_FMT_RIGHT_LEFT,
    .communication_format = I2S_COMM_FORMAT_I2S | I2S_COMM_FORMAT_I2S_MSB | I2S_COMM_FORMAT_PCM,
    .dma_buf_count = 8,
    .dma_buf_len = 64,
    .use_apll = 1,
    .intr_alloc_flags = ESP_INTR_FLAG_LEVEL1
};
```

Figure 6: Code snippet of the I2S configuration structure.

After setting up the structure, we can start collecting data from the microphone. Generally, the program flow is as follows:

```
void mainapp {
      install i2s driver with i2s_config;
      set i2s pins;
      set sampling rate;
      while(true) {
            sample = pop from I2S_NUM0 DMA buffer;
            if there is sample read {
                  sample = sample >> 14; // because 18-bit depth
                  …
                  // do something with the sample data
                  …
            }
      }
      close i2s driver;
}
```

After receiving or storing the data, we would need to analyze or be able to play the audio. For our design process, will use the software, Audacity, to play and analyze the data. However, for our final implementation, we will need to convert our raw audio data into a playable format. The easiest file type to convert to is the WAV format. The WAV file extension is basically a raw file with a header as described in the figure below:

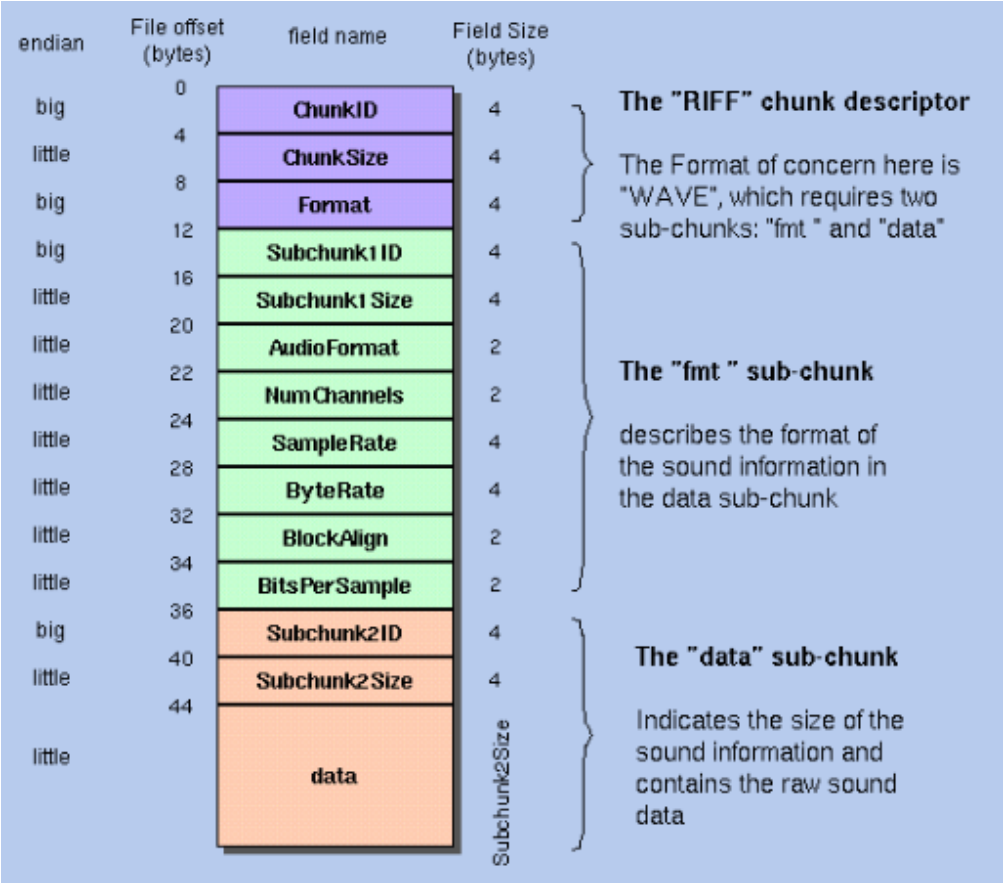



Figure 7: WAV format header specification. [4]

Based on this header definition, we have deduced a header to always to append-to-front of each of our audio packets (shown below in hex). The “xxxx xxxx” fields are to be determined during runtime because they store information about the size of the whole file and the size of the raw data.

```
5249 4646 xxxx xxxx 5741 5645 666d 7420
1000 0000 0100 0200 401f 0000 007d 0000
0400 1000 6461 xxxx xxxx 0000
```

Figure 8: Raw hex values used in our WAV headers.

The wave extension is a lossless file format, so it retains the quality of the raw data. However, there is a downside to it --- being a lossless file format, its size is huge (about 1 Mbits for 40 seconds of data). This can be handled by our Bluetooth module, but we will consider mp3 compression algorithms before uploading into the AWS server.

## Verification

There hasn’t been any verification work done on this part since we haven’t fully understood how the microphone i2s or UART works. The current plan is to have the microphone I2S and UART to work by November 9$^{th}$, then start improving on the SNR of our audio recordings. To verify our SNR is around 65 dB-A, we can use a silent recording and a voice recording and pass it to a Matlab script to calculate the SNR.

## UART

Serial communication is needed between the ESP32 Bluetooth module and our microcontroller (model not finalized yet) in the main hub. Also, since we do not plan on purchasing an Arduino board with built-in I2S ports, our only way of obtaining audio data for analyzing/playing is through a UART interface between the ESP32 Bluetooth module (this module has I2S) and a host PC. There are a plenty of resources and well written libraries like Pyserial which allows easy setup for a UART communication in any operating system. However, since Python is a not a compiled language (it is a scripting language), it is difficult for us to setup the boot sequence in the microcontroller such that there is a Python interpreter to execute our code. Thus, we will go with setting up UART interfaces in C language for POSIX OSes (Linux) using the RS-232 standard.

There are 3 default libraries in Linux which allows us to access and configure the device files: “fcntl.h”, “ioctl.h”, “termios.h”. In Linux, all instances are represented as files, which means there is a file corresponding to and storing the information for each device connected to the host machine. They are all stored under the “/dev” directory. Serial ports are given the names ttyS, hence, the first port will be ttyS0, second port will be ttyS1 and so on. We will use the *open(2)* function provided in *fcntl.h* to access the files corresponding to the serial port we are using. When opening the port, we need to use a few flags: O_NOCTTY, O_RDWR, O_NDELAY. The O_NOCTTY flag signifies if the device is the hosting terminal, which is not true, otherwise there will be many interrupts on the process. The O_RDWR flag is as the common read/write flag, and the O_NDELAY forces this process to start immediately regardless of the DCD line (hardware port which signifies if the device is connected).

After opening the file, we would need to load a copy of the attributes of the serial port into a *termios* structure, change it to the right configurations, then replace the default attributes in the file. For our purpose, we will use the options below:

- Baud rate of 115200 B/s to support high transmission rate.
- Local and read mode.
- No parity bits will be used. This is to simplify our design process; thus, we might implement this after everything is working.
- 8-bit character size
- 1 stop bit. We don’t need many of this since we are not using any parity bits and it allows shorter packets to be sent (faster).
- We don’t need hardware flow control provided by the RTS (Request to Send) and CTS (Clear to Send) pins because we know that the microphone will constantly send samples over if it is active, and the microcontroller will not need to send anything over to the Bluetooth module.
- Configure for raw input instead of canonical (line-by-line) input, which includes disabling echoing of any input.
- No software flow control.

A snippet of code for these configurations as well as the open/close procedure will be attached in the appendix.

Reading data from the port can be done using the *read(2)* function. We have chosen to leave the read function as a blocking function since we do not want to miss any samples and we do not know exactly how long is the delay is between each sample’s arrival.

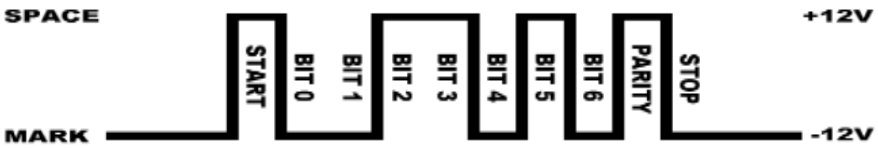



Figure 9: UART protocol timing diagram. [5]

There are three UART peripherals on the ESP32 breakout board. We would simply need to match the ports on both the devices. The pinout diagram for the ESP32 module is attached in the appendix.

We have tested our code in Ubuntu, and saved the audio files for playback (verified using the Audacity software).

## Plans

Judging from my progress so far, I am severely behind schedule as compared to my initial gantt chart as presented in the design review. This is mostly due to the fact that I was unfamiliar with the I2S and UART protocols which caused me to work on the same thing for almost a month. Comparing my progress to the rest of my team, I will still be able to make up all the lost time if I am able to follow the timeline laid out below:

| Task | Deadline |
|---|---|
| Finetune the audio recorder so. Make speech audible from the recordings | November 9th |
| Strip down the sensor modules, and design PCB for it | November 9th |
| Get SNR of ECG above 60 dB-A | November 23rd |
| Integrate with Bluetooth module | November 30th |

Table 1: Remaining tasks and deadline breakdown.

# References

[1] BMD101 datasheet [Online]. Available:

http://m5.img.dxcdn.com/CDDriver/CD/sku.241178.pdf

[2] Idealized ECG waveform [Online]. Available:

https://en.wikipedia.org/wiki/Electrocardiography#/media/File:SinusRhythmLabels.svg

[3] I2S Bus Specification [Online]. Available:

https://www.sparkfun.com/datasheets/BreakoutBoards/I2SBUS.pdf

[4] WAV file format [Online]. Available:

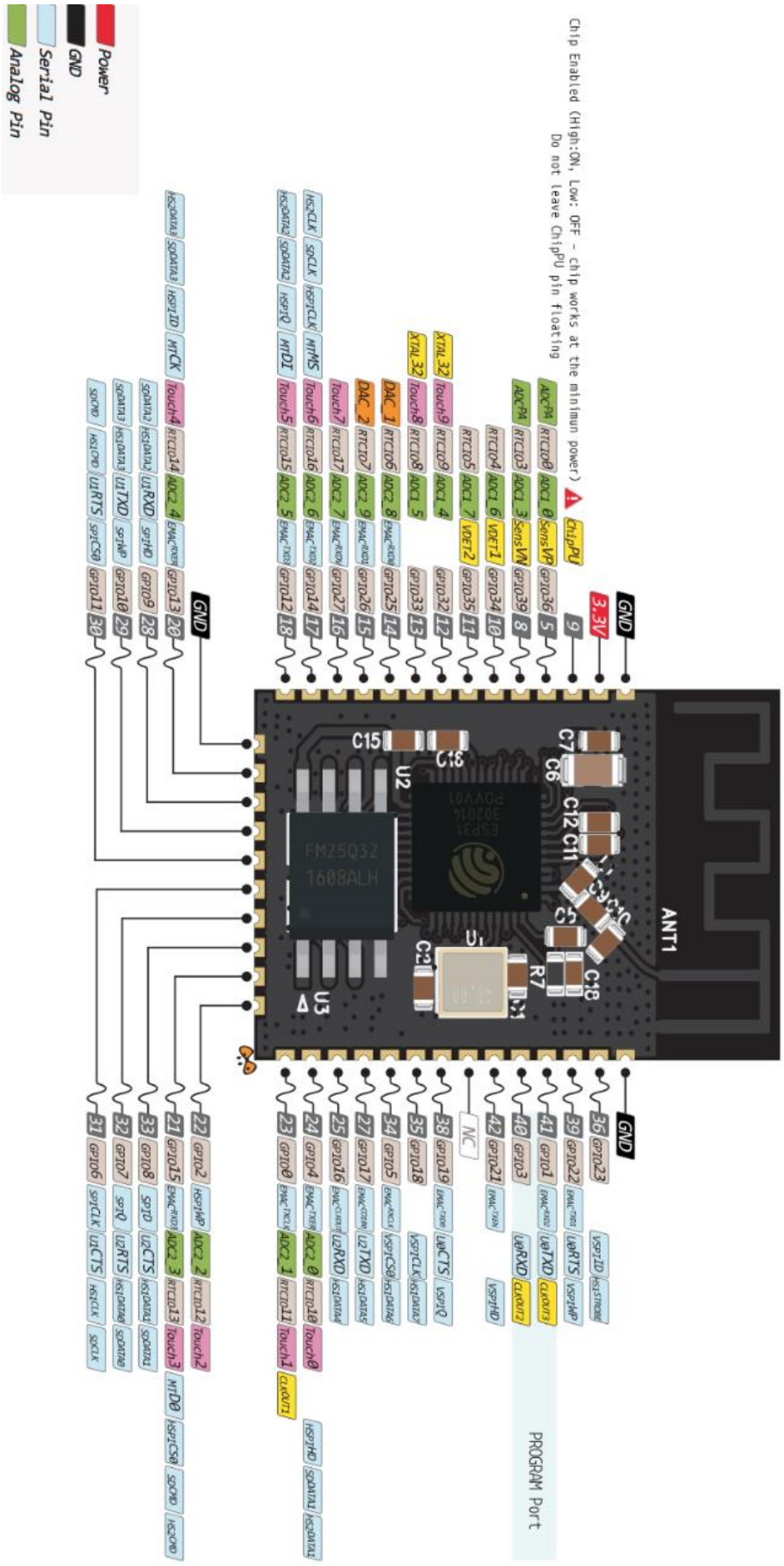
http://soundfile.sapp.org/doc/WaveFormat/

[5] POSIX Serial Programming [Online]. Available:

http://www.cmrr.umn.edu/~strupp/serial.html

# Appendix

```c
if( (serial_fd = open(argv[1], O_RDWR | O_NOCTTY | O_NDELAY)) == -1){
    /* Cannot open the port */
    printf("Unable to open port %s.\n", argv[1]);
    fclose(out);
    return 0;
}

//fcntl(serial_fd, F_SETFL, FNDELAY);
fcntl(serial_fd, F_SETFL, 0);

/*---------------------------------------- CONFIGURE SERIAL PORT ----------------------------------------*/
//SETTING THE BAUD RATE
struct termios options;
tcgetattr(serial_fd, &options); // get the current options for the port
cfsetispeed(&options, B115200); // these 2 sets the baudrate to 115200 regardless of the termios structure (new or old linux)
cfsetospeed(&options, B115200);

//SET NO PARUTY BITS (8N1)
options.c_cflag &= ~PARENB;
options.c_cflag &= ~CSTOPB;

//SETTING CHARACTER SIZE TO 8-BITS
options.c_cflag &= ~CSIZE;
options.c_cflag |= CS8;

//ENABLE HARDWARE CONTROL FLOW
options.c_cflag &= ~CRTSCTS;

//CONFIGURE FOR RAW INPUT
options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);

//DISABLE SOFTWARE FLOW CONTROL
options.c_iflag &= ~(IXON | IXOFF | IXANY);

// Enable receiver and set local mode.
options.c_cflag |= (CLOCAL | CREAD);

/* set the new options. TCSANOW - make the change now.
* Other options include wait for i/o buffers to be clear, or flush them.*/
tcsetattr(serial_fd, TCSANOW, &options);
/*---------------------------------------- CONFIGURE SERIAL PORT ----------------------------------------*/
```


Power
GND
Serial Pin
Analog Pin
Chip Enabled (High:ON, Low: OFF - chip works at the minimum power)
Do not leave ChipPU pin floating
ANT1
U1
U2
U3
FM25Q32
1608ALH
C15
C18
C7
C6
C12
C11
C5
C2
R7
PROGRAM Port
NC
GND
3.3V