



Rave Rings

Light up your night with audio
reactive earrings!

Group 7

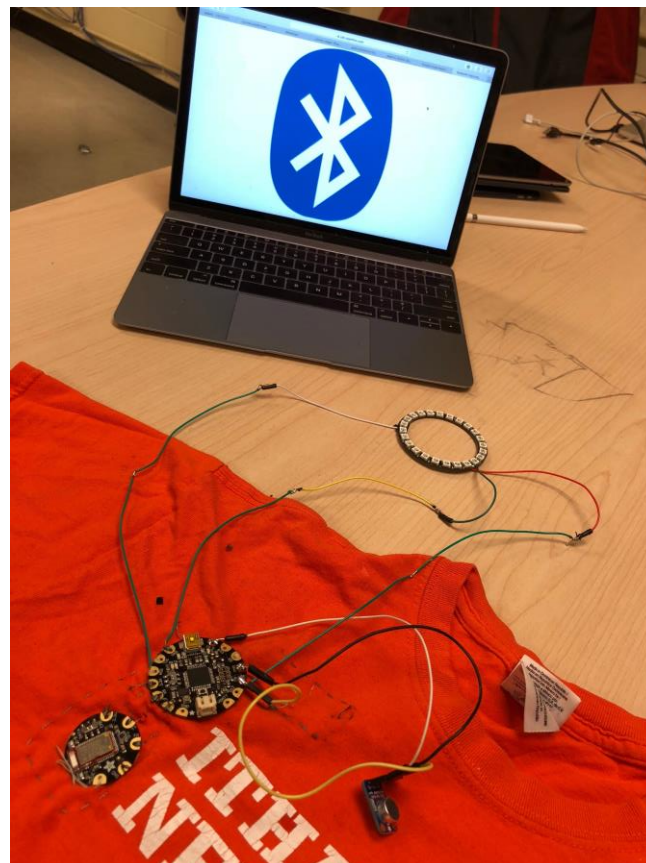
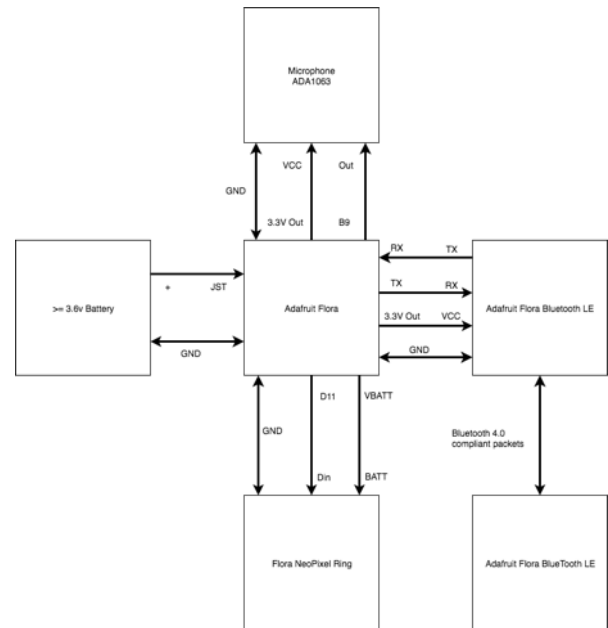
C.W, R.P, N.L, C.F, J.X.Q, A.N

Table of Contents

Rave Rings, what are they?	3
Devices	4
Adafruit Flora	4
Applications.....	4
How it Works.....	4
Interface	4
Power	4
Software	5
Difficulties	5
Bluefruit Flora module.....	6
Applications.....	6
How it Works.....	6
Interface	6
Power	8
Software	8
Microcontroller Platform	9
Difficulties	9
Code	10
NeoPixel Ring	11
Applications.....	11
How it Works.....	11
Interface	11
Power	13
Software	13
Microcontroller Platform	14
Difficulties	14
Code	15
Microphone.....	16
Applications.....	16
How it Works.....	16
Interface	16
Power	18
Software	18
Difficulties	18
Code	19
Computer Running Python Program	20
How it Works.....	20
Interface	20
Software	20
Difficulties	20

Rave Rings, what are they?

Our project is a “Personal Rave Light Show”. Rave Rings use a ring of LEDs, that light up in different colours to match the pitch of sound or music around you. This is accomplished using an analog microphone connected to an Adafruit Flora wearable. The microcontroller on the wearable transmits the raw sound data via a Bluetooth module, to a Bluetooth 4.0 capable host. This host runs a program written in python and performs an FFT (fast-Fourier transformation) to convert the audio voltage information to frequency information. This program will determine which lights will illuminate and what colours they will be. This information will then be sent back to the Flora as an encoded string over Bluetooth. Once decoded this string will inform the Flora which lights it is to illuminate. It will send this information to the NeoPixel strand to light up your night!

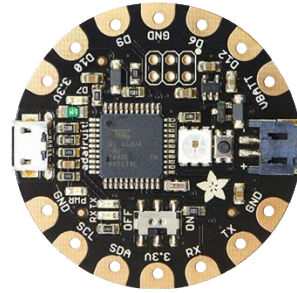


Devices

Adafruit Flora

Applications

Adafruit Floras are used in a wide variety of wearable projects, due to the small form factor they can be worn almost anywhere on the body. They can be used to run fitness trackers, watches that utilize rings of light to tell time, or just about anything you can think of making a wearable device out of.

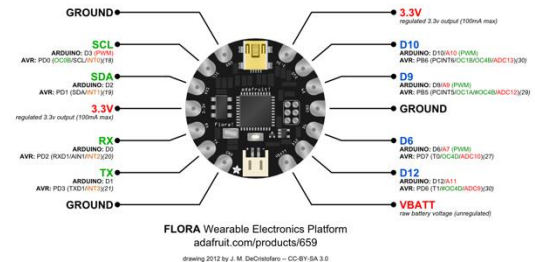


How it Works

The Flora microcontroller is a very complex little piece of equipment. Most of the work goes in to the interfacing that it does and that section will cover in detail the pins, and different connections that it can make. It's built around the Atmega32u4 chip, which has built-in USB support. This allows us to program the device simply with a mini-USB cable, and makes the device very robust. The device has a clock speed of roughly 8 MHz. While underpowered, even for an Arduino, it is more than fast enough to power a wide range of wearable devices.

Interface

There are many distinct interfaces available on the Flora module. First we will cover the mini-USB port. This can provide data and power to the Flora. It allows the device to be programmed. The 6 pin pads in the upper middle of the device allow for SPI interface connections. **SPI (Serial Peripheral Interface bus)** This allows communication both ways using a master-slave architecture. Further this means that the Flora controls the device it is connected to and can receive and send data to it. The other interfaces that are available on the device are the pin pads around the outside of the Flora. There are several power pin pads, as well as data receiver, transmitter, and general purpose data pins. The receiver and transmitter pin pads are used to connect to our Bluetooth module in this project. The other general purpose data pins are used to connect our microphone and NeoPixel display. The Flora has onboard RAM that stores information regarding the running program and data to be sent out over the pins. We will not go in to so much detail in this section as many pieces are covered by the other devices.



Power

The Flora can be used with a wide variety of power sources. It will accept a minimum of 3V, but 3.5 to 5V or higher is ideal. It even has protection circuits built in, in case power is hooked up in the opposite way it should be. The maximum power that should be provided is 9V. It can run on Lilon/LiPoly, LiFe, alkaline or rechargeable NiMh/NiCad batteries of any size.

Software

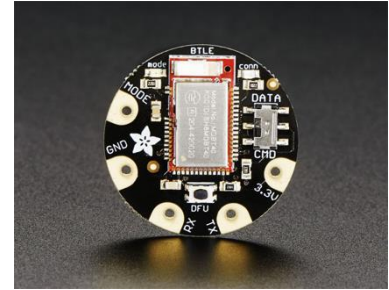
The Flora, like all Arduino compatible devices runs a variant of C++. This makes it easy for high level programmers to experiment with the device and access the full range of its capabilities. While this variant of C++ lacks access to the standard libraries of the language, there are many libraries available for use in the language. Tutorials provided on adafruit.com provide clear and concise documentation to get the device up and running.

Difficulties

We did not experience many difficulties with the Flora itself. Most of the difficulties we had were with the actual connections to the device. The libraries and tutorials that Adafruit provides on their website made using the Flora a breeze!

Bluefruit Flora Bluetooth module

Bluetooth is a low-power wireless connectivity, available in two flavors: Basic Rate/Enhanced Data Rate (BR/EDR) which enables continuous wireless connection and Low Energy (LE) which enables short-burst wireless connection. The Bluefruit Flora module (ADA-2487) uses a Bluetooth LE, which reduces power consumption by compromising the range and data throughput



Applications

The Bluefruit Flora module can be used for a variety of applications. Due to its low power consumption, the Bluefruit LE is ideal for small data transfers, but not for audio streaming. Typical applications for the Bluefruit Flora includes HID keyboard and BLE heart rate monitor.

How it Works

Bluetooth uses the Industrial Scientific and Medical Spectrum (ISM) to transmit data between the two parties. The ISM is a set of frequency allocated for a specific purpose, ranging from 2.400 - 2.485 GHz frequency.

Frequency Hopping Spread Spectrum (FHSS) is the technique used to transmit data, divided into packets, between connected devices. It operates on a range of frequency, with the connected devices sending and receiving data at the same frequency. However, the frequency in which the connected devices operates at is constantly switching at a rate of 1600 times per second. This is to avoid interference. An algorithm, Pseudo Random Algorithm, known only by the connected devices is used to generate random numbers to determine switches in operating frequencies.

Packets are sent as bits through radio waves encoding the 0s and 1s. Following each transmission, the frequency of both devices is changed.

Three components in the Bluetooth is used to accomplish this:

1. Radio Emitter
2. Antennae
3. Baseband Processor

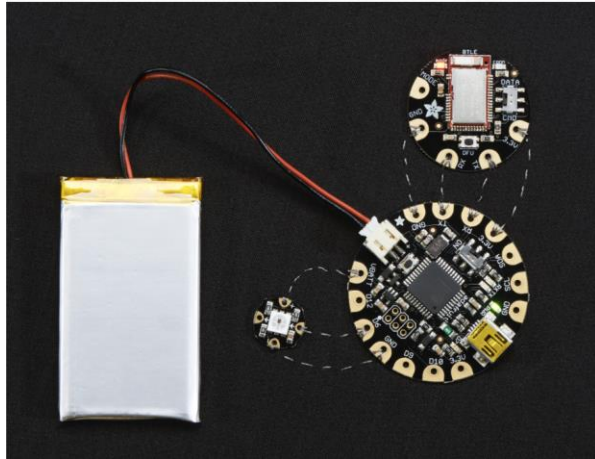
The radio emitter emits the radio waves, the antenna listens and receives the radio waves, and the baseband processor manages and processes the radio waves.

Interface

Bluetooth to Microcontroller

Four wires are sewn connected the Bluefruit to the Adafruit - two for the power pads and two for the data pads. A wire between the 3.3V pinpads of the Bluefruit LE and Adafruit

Flora is for the power supply, and a wire between the GND pinpads of the Bluefruit LE and the Adafruit Flora is for ground. The TX pinpad is responsible for transmitting data while the RX pinpad is responsible for receiving data. Therefore, a wire between the TX pinpad of the Bluefruit LE to the RX pinpad of the Adafruit Flora would transmit data from the Bluefruit to the Adafruit. The diagram below shows how the Bluefruit and Adafruit are connected together.



Bluetooth to Laptop

A Universal Asynchronous Receiver and Transmitter (UART) was used to transmit data between the connected devices. It communicates with serial input and output, and sends and receives data as if there were RX and TX lines between the devices. Bytes of data is transmitted as individual bits sequentially, which is then re-assembled back into bytes when received. To insure data integrity, a parity bit is attached to the data character which is then checked by the receiver for any transmission bit errors.

Since an asynchronous serial communication was used, the clocks of the transmitting and receiving device are not continuously synchronized. Instead, synchronization bits, which are the start and stop bit, are placed at the beginning and end of a packet and a baud rate is determined to specify how fast the data is transmitted. The start bit is signaled by a transition from 1 to 0 for a period that is reciprocal to the baud rate, and the stop bit by holding at 1.

For our project, utilising UART was, in many ways, a practical consideration. The limited pins available on the Flora meant that other modes of data transfer would not be possible as they require more pins to be connected from the microcontroller to the transmitter that govern whether or not a module should read data off the incoming pin or not. Since other parts of our project tied up the majority of the GPIO pins we were left with little choice but to use UART. Fortunately, this simplified things. The UART protocol, or at least Adafruit's implementation of it for BLE was designed to be as transparent to the programmer as possible.

Power

This device is powered by 3.3v and has no special power requirements.

Software

Software libraries, drivers, basic code:

BLE UART.

Adafruit_BLE.h

Adafruit_Bluefruit_LE.h

Special library for the device, and it's functions:

UART to send data to mac

```
writeBLEUart(uint8_t const * buffer, int size);
```

Names and functions of interface signals:

Ble.readline()

Reads data off of the incoming pin and into a buffer for the BLE object.

Ble.waitForOK()

When data is successfully sent, the module changes its RX pin to read "OK". This function stalls as we wait for data to be sent.

Ble.println(data) / Ble.print(data)

Writes the specified data, as a string, to the BLE module. This function is also used to write AT commands to the module. These commands handle things like requesting data to be sent or read or for the device to perform a full factory reset.

Ble.echo(boolean mode)

Enables or disables command echo. If enabled, the module will echo every command it is sent back to the microcontroller.

Ble.verbose(boolean mode)

Switches on/off verbose mode which is used primarily in debugging any issues with the module.

Ble.info()

Requests firmware/hardware information from the BLE module

Ble.factoryReset()

Performs a factory reset. This is important to do when launching any programme which utilises the BLE module. It guarantees that the module reverts to a known safe state before the programme begins.

Microcontroller Platform

The limited pins of the Flora Platform necessitated that we had to use the specialised Flora BLE module which was designed to work having just four pins connected.

Difficulties

1. Wiring: As with many parts of this project, the sewn connections between the Flora and the BLE Module meant that wires would often cross. At best this meant that we got garbage data being sent from/received from the module but, more often than not it resulted in a small plume of smoke rising up letting us know that the circuit had shorted.
2. Code: Since it was our first time working with the module, we did not understand the importance of certain procedures such as putting the module in to verbose mode before doing a factory reset.
3. Understanding Protocols: Before we settled on using UART as our means of data transfer, we spent a great many hours scratching our heads reading the specifications of BLE trying to figure out which means of data transfer were available to us and which of those would best suit our needs.

Code

This code allowed us to test the Bluefruit module in isolation

```
// Section One: Bluetooth //
/*
 * To utilise the bluetooth module we make heavy use of the Adafruit BLE library which makes for easy interfacing with
 * the devices. Further simplifying things is the fact that we make use of the UART protocol within Bluefruit LE. This
 * provides a simple interface where all we need to do is read from the RX pin and write to the TX pin.
 */

// Initialising BLE Object. Letting us initialise our object.
#include <Adafruit_BLE.h>
#include <Adafruit_Bluefruit_LE.h>

Adafruit_BluefruitLE_UART ble(BLUEFRUIT_HWSERIAL_NAME, BLUEFRUIT_UART_MODE_PIN);

// We additionally have to start up the device. In order to do a factory reset
// we must put it in verbose mode. This puts it in a known safe state

// Factory reset the bluetooth module so that everything is in a safe place
// First need to begin verbose mode, or factory reset wont work.
if ( !ble.begin(VERBOSE_MODE) ) {
  error(F("Couldn't find Bluefruit, make sure it's in CoMmanD mode & check wiring?"));
}

Serial.println( F("OK!") );

if (FACTORYRESET_ENABLE) {
  Serial.println("Resetting Bluetooth Module:");
  if ( !ble.factoryReset() ) {
    error(F("Unable to factory reset"));
  }
}

// We can then turn off echo and verbose mode.
// ble.info prints information about the bluetooth device which may be of interest.
ble.echo(false);
ble.verbose(false);
ble.info();

// The final part of setting up a BLE device in the adafruit library is to stall while waiting for
// a connection. In our code it is not as important that we do stall. If no connection exists it will
// do little in the main loop.
while ( ! ble.isConnected() ) {
  Serial.println(F("Stallin like Stalin!"));
  delay(500);
}

// To receive data, one must first print the proper AT command to the BLE device
ble.println("AT+BLEUARTRX");

// One can then read from the devices buffer like so
ble.readline(); // Reads data into the buffer off of the pin
if (strcmp(ble.buffer, "OK") == 0) { // Buffer reads OK if there's nothing to read in.
  // There's no data
}
else {
  // Process incoming data
}
ble.waitForOK(); // Wait for buffer to flush.

// Sending Data: Sending data is possibly even simpler than receiving data.
// We first write the proper command to the device.
ble.println("AT+BLEUARTTX=");

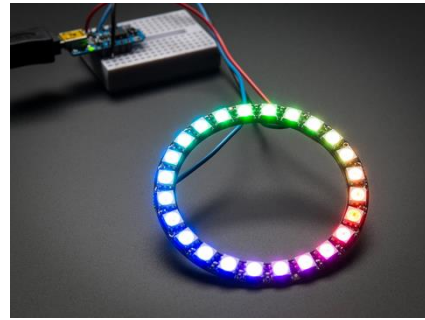
// Any data we want to send we can print to the bluefruit. Anything we send before a "Wait for OK"
// message call or a new line will be sent. This is true as long as the message remains under the
// limit of 256 bytes. Unfortunately, Adafruit's library only allows for strings to be sent over
// UART while we send 128 16 bit numbers (The limit of what we can send), we must do so one at a time
// as each character is 1 byte.
ble.print(MESSAGE); // Basic message sending.

// Below is the loop we use to send messages
for (i = 0; i < MESSAGE_COUNT; i++) {
  ble.print("AT+BLEUARTTX=");
  ble.print(MESSAGE[i]);
  ble.print(","); // Delimiter (important for parsing on python end)
}

// One can then ensure that their message has sent safely by waiting for the outgoing buffer to
// read OK.
if ( ! ble.waitForOK() ) {
  Serial.println(F("Failed to send?"));
}
```

[NeoPixel Ring](#)

A NeoPixel ring is a small electronic module made up of a printed circuit board ring with individual LEDs soldered on. The ring has three wires to connect to power, ground, and data. These rings are a part of the family of NeoPixel devices that Adafruit makes and come in a variety of sizes. Our ring has twenty-four individual LED's and with a diameter of two inches.



[Applications](#)

NeoPixel Rings are used in a wide variety of hobby projects. Due to their simple implementation and attachment, they can be used to light up any project you could imagine. Some projects similar to ours such as audio, or visually reactive programs lighting up the ring in certain ways. They can be used as decoration or indicator lights as well.

[How it Works](#)

The Ring works in a very elegant fashion at the circuitry level. A positive wire, and a ground wire give the ring the power it needs to drive the LEDs, while a single data wire is used to dictate how each pixel is lit individually. Each of these wires are connected to a compatible microcontroller. A timing diagram and circuit description below will go in to more detail regarding how this happens. On a high level, each pixel is sent three colour values which are made up of eight bit, binary integers. This mean each pixel is sent twenty-four binary values that dictate what colour, if any that it will produce. The colour values that are sent are green, red, then blue. Each pixel can be set one after the other with this pattern, starting from the pixel closest to the data connection. Once a final “RES” signal is sent, all the pixels that have been set will produce their colour.

NeoPixels receive data from a fixed-frequency 800 KHz data stream. Each bit of data therefore requires $1/800,000$ sec — 1250 nanoseconds. One pixel requires 24 bits (8 bits each for red, green blue) — 30,000 nanoseconds. After the last pixel's worth of data is issued, the stream must stop for at least 50,000 nanoseconds for the new colors to “latch.”

For our ring of 24 pixels, that's $(24 * 30) + 50$, or 770,000 nanoseconds. $1,000,000,000 / 770,000 = 1298$ updates per second, approximately.

[Interface](#)

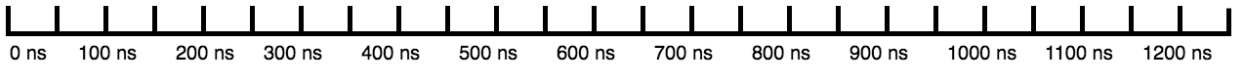
The interface of a NeoPixel ring is a single data wire connected to our Adafruit Flora. The data sent is high or low voltage with sub microsecond timing requirements that indicate whether each binary value is set to a logic one or zero for 24 binary values of each pixel. This timing diagram (waveform diagram, Figure 1) shows the timing ranges and requirements.

RAVE RINGS

0 Code



1 Code



Code	Meaning	Time Frame
T0H	0 code high voltage time	250 - 550ns
T1H	1 code high voltage time	650 - 950ns
T0L	0 code low voltage time	700 - 1000ns
T1L	1 code low voltage time	300 - 600ns
RES	RES / Latch low time	50000ns+

Example For A Single Pixel

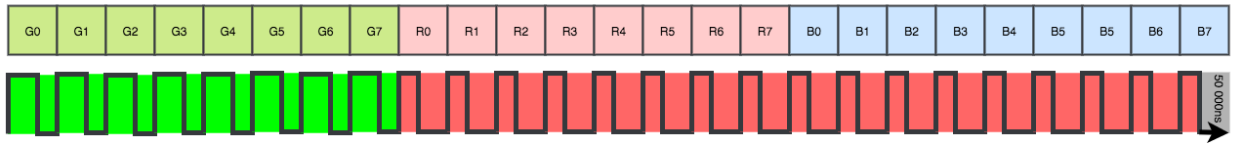


Figure 1 *RES is not to scale as it would not fit properly.

In this diagram we see the difference between a 0 and 1 code indicated by the length of time a high voltage is sent. Each pixel requires 24 of these voltage patterns to be sent for a colour to be indicated. Each subsequent pixel can be set after in the same pattern. Once all the pixels that are meant to be set have been given a voltage pattern, a low voltage is sent for over 50 000ns to indicate that data is finished being sent. This is called a “latch” or “reset” and once sent, the ring will illuminate each pixel with the colour that was sent. The values for each colour (Green, Red, Blue) have a maximum of 256. In Figure 1 example, a single pixel is being set to pure green (256, 0, 0). This is colour coded for easy reading as well. Pure white would be each value at maximum (256, 256, 256) and off would be each value at 0.

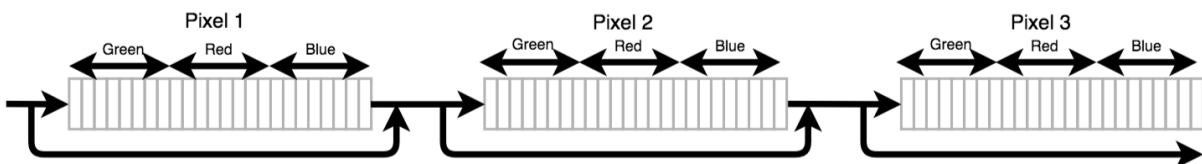


Figure 2

When setting the pixel's colour, it starts with the pixel closest to the data wire of the ring and moves around the ring a single pixel at a time. Each pixel has a 24-bit register embed in it that allows it to continuously drive the colour of the LED after it is set. Before each register is a “data re-shaping circuit” which prevents distortion of the wave forms and ensures each subsequent pixel is receiving the correct values. Once the pixel's register is filled it forwards the

remaining data to the next pixel and so on. Figure 2 shows these registers. Figure 3 shows how the data forwarding works, as each pixel starts to receive all of the remaining data, it fills its register with the first 24 bits and sends the rest on its way. This is known as cascading port transmission and works logically the opposite of a traditional shift register.

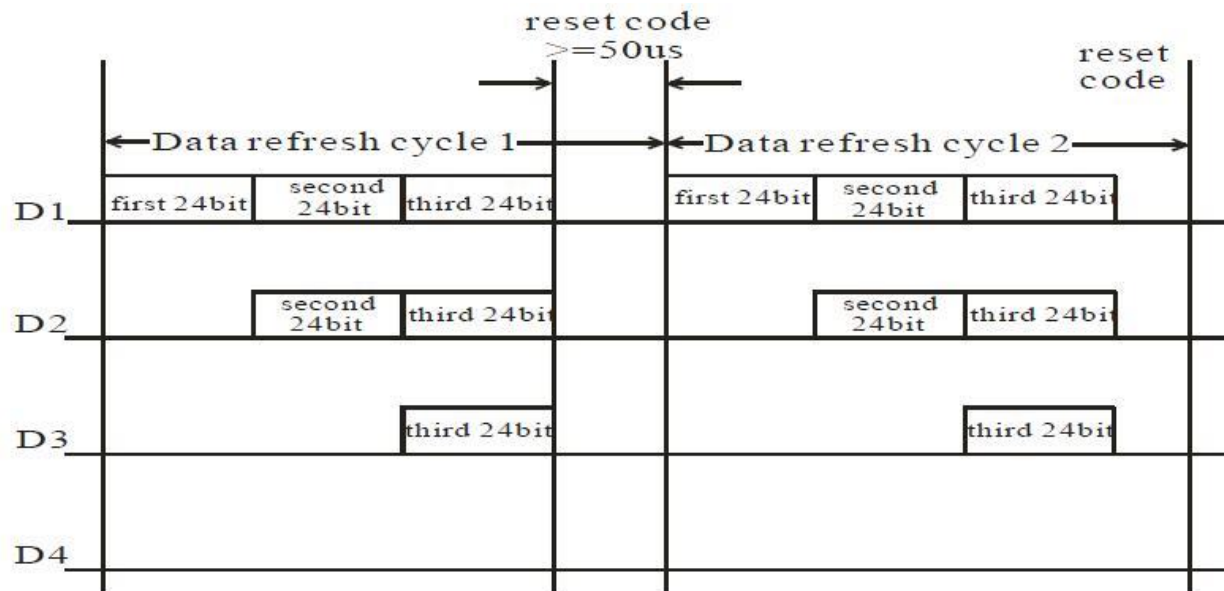


Figure 3

Power

The NeoPixel ring requires a 5-volt power supply. Each LED on the ring can draw up to 50 mA at 5 V when outputting white at full brightness. That means the ring could draw up to a maximum of around 1.2 A. No special requirements further than that are necessary for this ring

Software

The software to run NeoPixels is quite simple as Adafruit provides a library that allows for easy control of any number of neo pixels. The software is written in Arduino C and the library provided by Adafruit is called Adafruit_NeoPixel.h. This library can be implemented using the statement:

```
#include <Adafruit_NeoPixel.h>
```

The functions available for use through this library are as follows:

```
//defines the ring variable for future use
```

```
Adafruit_NeoPixel varName = Adafruit_NeoPixel (number of pixels, pin, definition)
```

```
//initiates the use of the NeoPixels
```

```
varName.begin ()
```

```
//set a specific pixel (starting at 0 up to 23 for our ring) to the colour value of the arguments
```

```
varName.setPixelColour(pixelNum, Green, Red, Blue)
```

```
//displays and previously set colours on the ring  
varName.show()
```

The function `setPixelColour` can be implemented in a for loop to set all the pixels to a colour. This uses the increasing value of the for loop as the pixel number to set. This single function can be used with many different kinds of logic to create interesting patterns on the ring.

Microcontroller Platform

We are using the AdaFruit Flora microcontroller. This microcontroller is the most compatible with the NeoPixel ring in terms of clock speed and has software libraries to interfacing from the microcontroller to the ring.

Difficulties

We encountered a couple different difficulties with the NeoPixel Ring. The worst issue we ran into was how fragile the ring's wiring was. All three wires detached from the ring at some point and needed to be re-soldered. We were wary of the timing of the data being sent and learned a lot about the timing diagram before starting to write the code. We thought that we would run in to issues regarding this, but the library that we used made this point moot as it abstracts all of this away. The documentation of the ring indicates that the timing is actually stricter than it is in reality. One neat thing that we found out was that there is no theoretical maximum length of time that a 1 code must fall under for a single binary of one pixel. In practice this can cause issues though as the embedded systems of the ring perform wave-form re-shaping. This prevents the accumulation of distortion during data forwarding, and if the timing is not correct could cause undefined behaviour.

RAVE RINGS

Code

This is the code that allowed us to test the NeoPixel ring in isolation. It is simply connected to the Adafruit Flora and power. This code allows for setting specific pixels to a colour value.

```
/*
 * Due to the existence of Adafruit's library, interfacing with Neopixels
 * becomes a pretty trivial task. Fortunately, this makes it easy, should one
 * desire to have the pixels shine in vibrant patterns with out too much difficulty.
 */

// As always we begin by importing the library and initialising the object.
// Note: The two constants in the object instantiation are based on the data transfer
// speed to the light and the type of value the pixel expects.
// NUMPIXELS is the amount of pixels, while the PIN is the data pin they are connected to.

#include <Adafruit_NeoPixel.h>
Adafruit_NeoPixel pixels = Adafruit_NeoPixel(NUMPIXELS, PIN, NEO_GRB + NEO_KHZ800);

// To initialise the pixels to an off state, we initialise the library, then run the
// show command. Since no lights are available, it will display black,

pixels.begin();
pixels.show();

// To propagate a colour to an individual pixel one must set the pixel colours
// and then latch them into place. The brilliance of the neopixel means that for our
// application (and the desire to not blind people), we peak any individual colour value
// around 90-100. So long as the ratios are consistent, this provides us a wide range of
// colour!
pixels.setPixelColor(pixelCount, GREENVALUE, REDVALUE, BLUEVALUE);
pixels.show(); // This sends the updated pixel color to the hardware.

// Lastly, when working with neo-pixels it can be helpful to delay the arduino for some time
// to allow the lights a chance to shine.
delay(delayval); // Delay for a period of time (in milliseconds).
```

Microphone

The Electret Microphone Amplifier MX4466 electret microphone amplifier is a board that comes with a microphone attached. The microphone is powerful enough to pick up sound ranging from 20-20k hz frequency.

Applications

This device is best used for projects such as voice changers, audio recording/sampling, and audio-reactive projects that use FFT.



How it Works

Three pins attached to this microphone provide the power and data transmission ability, ground, power, and output. The electret microphone is designed with a capacitor composed of two plates, each holding a fixed electric charge. One plate is fixed while the other is a diaphragm that vibrates as sound waves pass through. The vibration will change the distance between the diaphragm and the fixed plate, which will cause a change in capacitance between the two conductive plates. This change will cause a voltage variance on the back-plate, and in turn, this is picked up by the amplifier, which then outputs it as sound data.

The amplifier module actually includes the fixed plate (also called the pick-up plate, as to pick up sound), which is sealed inside a plastic container. The amplifier module also contains an JFET transistor, which serves as the actual amplifier component that strengthens the voltage difference output. A JFET has three pins, the gate, which is connected to the pickup plate, source, which is connected to the ground, and drain, which is connected to the signal (this is where the output is being measured). The voltage across capacitor can be represented as $V = Q/C$, where Q represents the charge that these plates carry with them, and C being the capacitance. In the microphone setting, this equation can be further modelled, as we can calculate C using the formula $C = \epsilon \cdot A/t$, where ϵ is a constant that correlates to the material property of the medium between capacitors, A being the area of plates, and t being the separation between two plates. When V is changing, the conductivity of the JFET gate is also changing, which leads to the current change on the drain, producing a signal across the drain. The drain is what's being measured.

Interface

Since the Electret Microphone is an analogue device, it sends information back to the Arduino through modulating the level of voltage on the output wire. Upon arriving at the flora, this signal serves as input to an integrated circuit which performs the Analogue to Digital Conversion. It must be said that an Analogue to digital conversion is not without fault. There are two major inhibitors to a flawless transformation

1. The transformation of continuous signals into discrete signals necessitates sampling of the input. The higher the sample rate, the more accurate the digital representation of the original data is. For our project, we used an effective sampling rate of just under 9000 hz. While well under the normal sample rate for music (which is typically at 44KHZ it was more than sufficient enough for our application. Use of a Capacitor called a Sample and Hold will hold the voltage coming in over the wire so that it stays constant for the duration of the conversion

Some issues that impact the sampling rate include:

Aliasing

Aliasing occurs when a function is sampled too infrequently resulting in missed cycles. This has the effect of giving having a given frequency as being lower than it would. For example, a 2hz sine wave sampled at only 1.5 hz would appear as a 500hz sine wave. To avoid aliasing, ADC input must be filtered to remove all frequencies above half of the sampling rate. In our case this means that any frequencies above ~4000 hz are disregarded by the flora.

Oversampling

For efficiency, something of particular importance on low power devices we sample just as much as we need to. This means that the quantisation noise is introduced as a white noise which is spread over the entire band of the converter.

2. Even if we could sample continuously, the amplitude of the analogue signal has to be quantised into the corresponding digital amplitude. There are a couple of factors which dictate how well this process can be performed.

Resolution

Indicates the number of discrete values that can be produced over a range of analogue values. Higher resolution means less resolution error. Since the the number of digital values in a given range are expressed in bits, we assume that there the number of levels is a power of 2. This works in a straightforwardly. An 8 bit resolution can produce 256 intervals. We can think of resolution in an electronic manner. In this paradigm, we can think of resolution being the needed change in voltage to see a corresponding change in voltage. Mathematically: $Res = \frac{VoltageRange}{2^M}$

Voltage Range is just the difference between the High Reference Voltage (for the Flora this is 3.3 volts) and the Ground Voltage. M is the is the number of bits we allow the express this range. For the flora this is a 10 bit range (Permissible values from 1-1028
Thus the voltage resolution is 6.6/1024. We would need only a shift of 6mV to change the output. Unfortunately, this figure is an upper bound on our resolution, not what is obtained in practise. Presence of noise reduces the Effective Number of Bits for our resolution. It is difficult to ascertain from the Flora's spec sheets the precise nature of the ADC. Presence of things like pre amplifiers or other components increase the amount of noise present in the signal.

Quantisation Error

Just the process of quantisation introduces “noise” to the signal. It is an artifact of rounding from the analogue input to a digital output. As one expects, the ideal is to have the smallest amount of noise introduced. The amount of noise introduced is frequently expressed as a ratio called the Signal to Quantisation Noise Ratio. Formally it is:

$$SQNR = (20 * \log_{10}(2^Q))$$

We can approximate this to simply $6.02 * Q$. In our case this means that the SQNR for the Flora should be approximately 60.2 db.

Power

This device can be powered by anywhere from 3 to 5 volts. If connected to a device like the Arduino Uno, which is capable of supplying multiple voltages to devices, an additional connection supplying the reference voltage must be connected to the Data Pin.

Software

The program that receives data from the microphone runs on the Adafruit Flora microcontroller. It is written in C++. No libraries are used to perform the Analogue to Digital conversion, this is built in (with specialised hardware) to the microcontroller. .

Difficulties

When we digitise a sinewave, using a less than optimal clock (we most assuredly are) results in uncertainty in precisely when samples are taken. For Direct Current this error is zero and at low sampling frequencies it is negligible. The higher the error caused by jittering will be.

An additional difficulty comes from the limited range of the microphone. Its effective range is quite small. While sufficient for a proof of concept, were we to proceed with developing this idea into a product, a more powerful microphone would have to be utilised.

Code

The below code does not test the microphone per say but instead establishes all of the data structures necessary for the microphone to be sampled in what is called "Free Run Mode". Should one want to test the output of the microphone, one simply needs to run a for loop printing all of the values in capture.

```
// PART THREE: MICROPHONE //

/*
 * In order to get samples quickly enough, we do the
 * Analogue - to - Digital conversions in what is called
 * "Free Run Mode". This allows for the arduino to
 * with out our direct intervention collect incomming
 * voltages off the wire. When 128 (the number we use for the
 * FFT) are present, it will send an interrupt to the microcontroller
 * and cause it to run a short interrupt service routine. This routine
 * can do anything but we use it to normalise values so that they are in
 * range for the FFT.
 */

// We do not use a library for this but we have to define some constants
#define FFT_N 128
#define MIC_ADC_CHANNEL 6 // Pin mic is connected to. On the flora all digital pins can be analogue pins

// We also need to define a couple variables for working with the audio buffers.

int16_t capture[FFT_N]; // Audio Capture Buffers
volatile byte samplePos = 0; // Position Counter for the Buffer

// To enable freerun mode we need to do a fair bit of setting flags in the Arduino
// as it is not enabled by default. The below code initialises our microphone
// to have a sample rate of 9.6k hz.
// Init ADC free-run mode; f = ( 16MHz/prescaler ) / 13 cycles/conversion
ADMUX = MIC_ADC_CHANNEL; // Channel sel, right-adj, use AREF pin
ADCSRA = _BV(ADEN) | // ADC enable
         _BV(ADSC) | // ADC start
         _BV(ADATE) | // Auto trigger
         _BV(ADIE) | // Interrupt enable
         _BV(ADPS2) | _BV(ADPS1) | _BV(ADPS0); // 128:1 / 13 = 9615 Hz ((SAMPLE RATE))
ADCSRB = 0; // Free run mode, no high MUX bit
DIDR0 = 1 << MIC_ADC_CHANNEL; // Turn off digital input for ADC pin
TIMSK0 = 0; // Timer0 off

sei(); // Enabling interrupts.

// In our main loop we must have the following statement, to stall the loop if there the buffer is not
// full yet. We then reset variables used in the counter and resume sampling.
while(ADCSRA & _BV(ADIE)); // Wait for audio sampling to finish
samplePos = 0; // Reset Sample Counter
ADCSRA |= _BV(ADIE); // Resumes the sampling interrupt

// Lastly there is the Interrupt Service Routine.

ISR(ADC_vect) {
    // Here we do what ever processing we feel is appropriate.
    // Once the processing is done we check if our buffer is full so that we can disable our
    // interrupt.

    // Here we check if our buffer is full. If so we turn off our interrupt.
    if(++samplePos >= FFT_N) { // Sample pos is the sample we have been processing.
        ADCSRA &= ~_BV(ADIE);
    }
}
```

Computer Running Python Program

The python program is responsible for processing sound data that is sent from Bluetooth. The program using the information to decide what colours to light the ring with, and sends the data back to the microcontroller over Bluetooth.

How it Works

Using a Fourier Transform function from Numpy library, the `fft` function in Numpy takes a list of audio data, and returns the corresponding frequency bins. We then find the highest frequency, and use that value as our result for color mapping from frequency to color. The output of the program is three integer values representing the r, g, b values of a color, and this output was write back to the Bluetooth. The program would run continuously unless stopped by keyboard interrupt.

Interface

The interface is designed so that the Arduino program would send one piece of audio data (type integer) at a time, and the python program would read these numbers and store them into a buffer. After buffer size reaches 128, the python program proceeds to analyze this data. We use a library to search for Bluetooth LE modules in the proximity, and is responsible for establishing connection between laptop and the Bluetooth LE module, as well as maintaining that connection between the two end programs.

Software

The library we used for communication between the Bluetooth module and the python program is called “Adafruit Python Bluetooth LE”, It is a Bluetooth LE library that’s developed specifically for the Mac and Linux systems.

Difficulties

We experienced difficulties deciding on a language to use on this end of things. We originally used Java, but due to a lack of documentation for the libraries we wanted to use we moved to Python. Due to using a library we cannot be certain that they are performing the functions correctly. This is due to a lack of knowledge of FFTs. We can semi-validate the output but must rely on the library for this which causes difficulty.

Code

```

# Author: Tony Dwyer
import Adafruit_BluefruitLE
from Adafruit_BluefruitLE.services import UART
import time

# Get the BLE provider for the current platform.
ble = Adafruit_BluefruitLE.get_provider()

# Main function implements the program logic so it can run in a background
# thread. Most platforms require the main thread to handle GUI events and other
# asynchronous events like BLE actions. All of the threading logic is taken care
# of automatically though and you just need to provide a main function that uses
# the BLE provider.
def main():
    # Clear any cached data because both bluez and CoreBluetooth have issues with
    # caching data and it going stale.
    ble.clear_cached_data()

    # Get the first available BLE network adapter and make sure it's powered on.
    adapter = ble.get_default_adapter()
    adapter.power_on()
    print('Using adapter: {}'.format(adapter.name))

    # Disconnect any currently connected UART devices. Good for cleaning up and
    # starting from a fresh state.
    print('Disconnecting any connected UART devices...')
    UART.disconnect_devices()

    # Scan for UART devices.
    print('Searching for UART device...')
    try: ...
    finally: ...

    print('Connecting to device...')
    device.connect() # Will time out after 60 seconds, specify timeout_sec parameter
                    # to change the timeout.

    # Once connected do everything else in a try/finally to make sure the device
    # is disconnected when done.
    try:
        # Wait for service discovery to complete for the UART service. Will
        # time out after 60 seconds (specify timeout_sec parameter to override).
        print('Discovering services...')
        UART.discover(device)

        # Once service discovery is complete create an instance of the service
        # and start interacting with it.
        uart = UART(device)

        # Write a string to the TX characteristic.
        uart.write('Hello world!\r\n')
        print("Sent 'Hello world!' to the device.")

        time.sleep(0.5)

        # Now wait up to one minute to receive data from the device.
        print('Waiting up to 60 seconds to receive data from the device...')
        received = uart.read(timeout_sec=60)
        if received is not None:
            # Received data, print it out.
            print('Received: {}'.format(received))
        else:
            # Timeout waiting for data, None is returned.
            print('Received no data!')

    finally:
        # Make sure device is disconnected on exit.
        device.disconnect()

```

Citations:

newstarleds. "APA107 LED Chip, the Difference with APA107 and APA102C/SK6812." *NEWSTAR LED CO., LIMITED*, Newstarleds, 19 Apr. 2017, newstarleds.wordpress.com/2017/04/19/apa107-led-chip-the-difference-with-apa107-and-apa102csk6812/.

Josh. "NeoPixels Revealed: How to (Not Need to) Generate Precisely Timed Signals." *Josh.com*, Word Press, 13 May 2014, wp.josh.com/2014/05/13/ws2812-neopixels-are-not-so-finicky-once-you-get-to-know-them/.

AdaFruit. "Advanced Coding." *Learn.adafruit*, AdaFruit, 12 Oct. 2017, learn.adafruit.com/adafruit-neopixel-uberguide/advanced-coding.

The Scipy community. "Numpy.fft.fftfreq¶." *Numpy.fft.fftfreq — NumPy v1.13 Manual*, The Scipy Community, 10 June 2017, docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.fft.fftfreq.html.

adafruit. "ELECTRET MICROPHONE AMPLIFIER - MAX4466 WITH ADJUSTABLE GAIN." *Adafruit*, AdaFruit, 7 Mar. 2016, www.adafruit.com/product/1063.

"An Introduction To Delta Sigma Converters." *Beis.De*, 2017, <http://www.beis.de/Elektronik/DeltaSigma/DeltaSigma.html>.

다수의 마이크 모듈, 주파수 변환 라이브러리. "Electret Microphones." *Openmusiclabs*, Open Music Labs, 24 Nov. 2015, www.openmusiclabs.com/learning/sensors/electret-microphones/index.html

"Robot Platform | Knowledge | Analog To Digital Conversion." *Robotplatform.Com*, 2017, http://www.robotplatform.com/knowledge/ADC/adc_tutorial.html.

"ADC - DAC Glossary." 2002, <https://www.maximintegrated.com/en/app-notes/index.mvp/id/641>.

Statement of Review and Approval




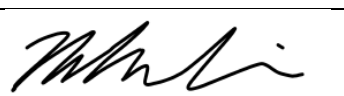

Each team member must sign this form acknowledging that they have reviewed and approve of their team project documentation as submitted. This is important as team members receive individual marks associated with their contribution to the project. Submit this completed form to the course instructor with the final documentation when submitted.

Team #: 7

Team Leader: Rocky Petkov

Date Team Project was uploaded to the course web site for marking: Oct 23rd 2017

Team Member Signatures

Team Member Name (print)	Date	Signature	Approval of Submission (please check for yes or leave blank if no)
Clayton Warren	Oct 23 2017		Yes
Rocky Petkov	Oct 23 2017	Rev. 	Yes
Jack Qiao	Oct 23 2017		Yes
Niki Lin	Oct 23 2017		Yes
Chuck Fu	Oct 23 2017		Yes
Aaron Nahum	Oct 23 2017		