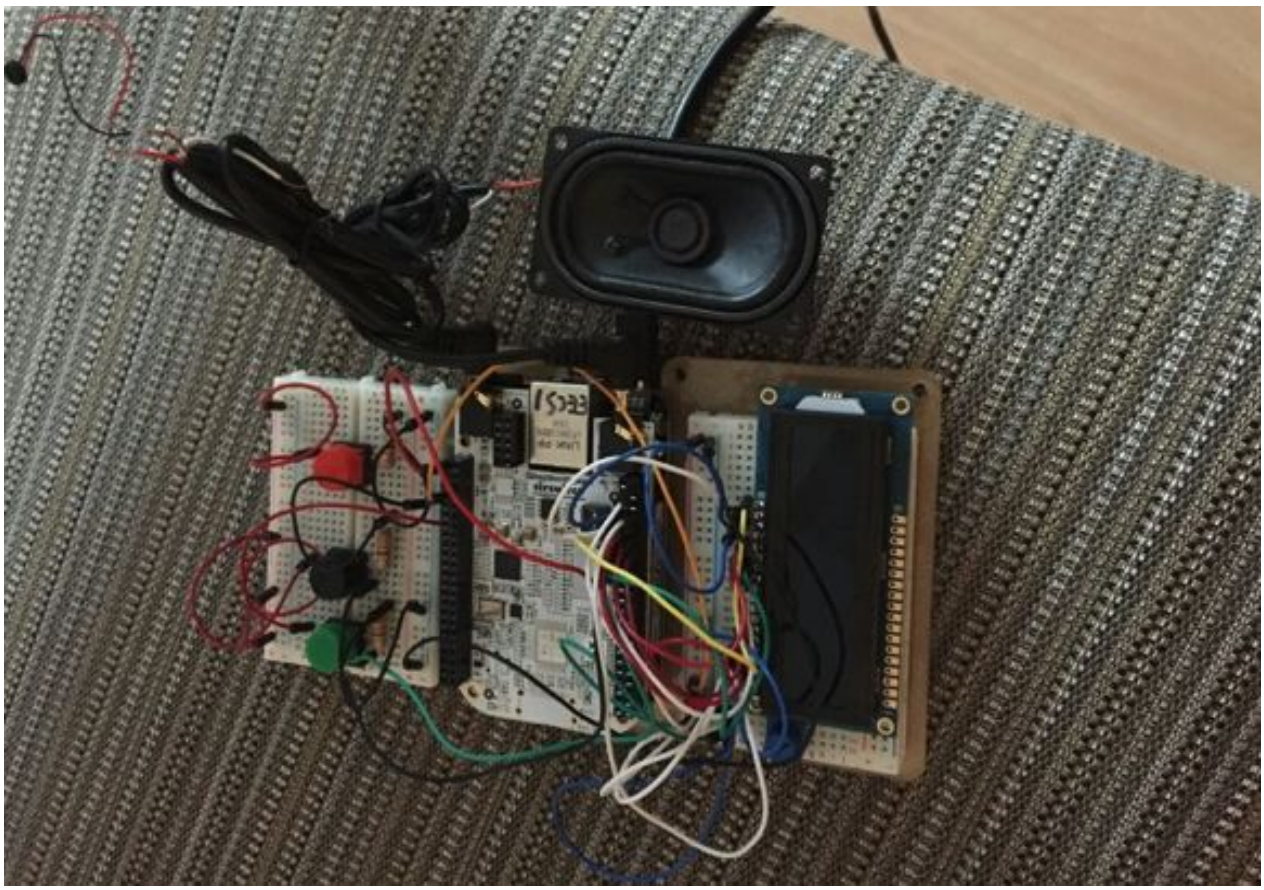
BED TUNES

EECE 4376 CLASS PROJECT

TECHNICAL MANUAL



DESIGNED BY: ILHAM HASNUL, BEN COOK,

ANTHONY SAWYER, QUENTON STEVENSON

Introduction

Bed Tunes is a project to combine the functionality of a music instrument tuner and a playback device that is easy to use and interactive. The device has two different modes, a tuner mode and a recorder mode. In the tuner mode, the device listens to it surroundings for a tune being played. Once the device detects a tune, it correlates the tune with a note on the frequency spectrum and displays the result to the user via a LCD screen. In recorder mode, the device can either record audio and store the audio data in a binary file the device's memory, or playback an already existing audio data in the binary file. The device is initially programed to tune a guitar in tuner mode. However, the device's program can be changed in a relatively simple manner to tune any musical device as long as the audio frequency to musical notes correlation is known.

List of Parts

- BeagleBone Black
- RGB backlight negative LCD 16x2 - RGB on black (Adafruit Product ID: 399) + 10k Ohm potentiometer and strip of header pins
- BeagleBone Black Audio Cape (Mfg Part No: BB-BONE-AUDI-01)
- Visaton 8048 Speaker, 20 kHz, 8OHM, 4W (Newark Part No.: 40P1184)
- Wired Miniature Electret Microphone (Adafruit Product ID: 1935)
- 3 Push Buttons (COM-10302 ROHS)
- 3 Through Hole Resistor, Carbon Film, MCF Series,10 kohm, 250 mW, ± 5%, 250 V, Axial Leaded (Newark Part No.: 38K0328)
- Right-Angle 3.5mm Stereo Plug to Pigtail Cable
- Connecting Wires

- Breadboard

This particular model of the Beaglebone Black Audio Cape was chosen as it has many functionalities that enhance the final design of the device. First, the Audio Cape has been specially designed for use with the Beaglebone Black, thus no compatibility issues will be encountered with the use of the audio cape. Beside that, the audio cape also has two audio jacks to receive input signal from a microphone and send an output signal to a speaker. The audio cape also has an analog-to-digital converter to convert the input signal from the microphone to a digital signal that can be read through a GPIO pin on the Beaglebone Black. The audio cape also has an digital-to-analog converter to convert digital signals from a GPIO pin from the Beaglebone Black to an analog signal that can be sent to the microphone. Lastly, the audio cape is able to sample the audio data at a sampling rate of 96 kHz, which can be used to reproduce a good playback of the audio data.

The Adafruit LCD screen is chosen as there already exists an open sourced Python driver to interface the LCD screen and BeagleBone Black. The Adafruit LCD screen also has two rows of LCD that is adequate to show information of what note is being heird in the tuner state.

A folder of datasheets for the parts is also in the CD submitted. However, the Adafruit website where the microphone is bought explicitly says that the microphone does not have a datasheet, but the datasheet submitted is a microphone is similar to the Adafruit microphone.

State Machine

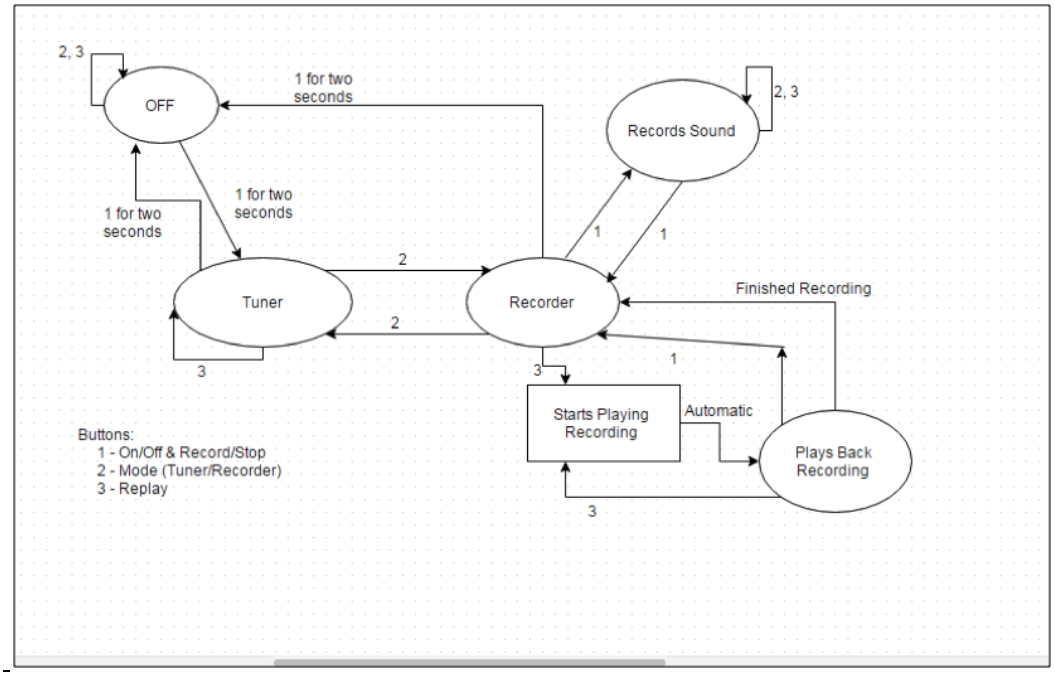


Figure 1. State Diagram Displaying Control Logic of Guitar Tuner + Recorder

In designing the tuner and recorder device, it was imperative to demonstrate the functionalities employed and how they each function related to one another. The illustrated deterministic finite state machine in **Figure 1** is used as an abstraction to outline the design algorithm implemented for our Bed Tunes device. In this diagram, the bubbles represent the states and the output action, while the arrows represent state transitions. The arrow labels indicate the input value corresponding to the transition. Additionally, squared bubbles were used to show actions/results between states. The tuner and recorder device consists of five unique states: OFF, TUNER, RECORDER, RECORD SOUND and PLAYBACK. The inputs present are specific button clicks, Record/Stop, Mode and Replay, with a predetermined allotted time that the button must be held.

The OFF state represents the device in a powered down condition. In this situation, the device returns nothing to the user. Any control or memory on the stack lies dormant on the system until the record/stop button is held for one second, which sets the device in the TUNER state.

The TUNER state applies the pitch detection algorithm to an input signal and displays the relative pitch of the musical notes. It is important to note that holding the Record/Play button with a one second delay suspends the TUNER state and boots off the device. However, pressing the Mode input changes the context of the program. Without any delay, the TUNER state is switched to the RECORDER case.

In the RECORDER case, the device enables the microphone for recording. If the user selects the Mode key, the RECORDER state switches context back to the TUNER state. By pressing the Record/Stop button, the user begins the RECORD SOUND state of the program. RECORD SOUND overwrites any data previously recorded and begins to save the incoming audio signal in a binary file. The Mode and Replay buttons have no effect in this given state. Additionally in the RECORDER state, if the Replay button is pressed down, the PLAYBACK state commences. The PLAYBACK condition is responsible for reading in the data from the binary file. If the file is empty, no playback will be available to the user, and places the program back in the RECORDER state. Lastly, holding the Record/Stop button for two seconds initiates the OFF state.

Principles of Operation

The tuner operates strictly according to the states described in the state machine diagram above. The state is kept track of by a integer variable. Through conditional variables and checks, specific threads are synchronized to run only when the tuner is in the corresponding state and, if necessary, certain buttons are pressed. Each thread is continuously looping and always checks if the guitar tuner is in the corresponding state. If not, it will lock until the guitar tuner reaches the state. State 1 is the tuner state. A reading thread and pitch detection thread run in this state. To ensure proper synchronization, two arrays and associated mutexes are provided for reading and calculating pitch. While an array is being fed digital values, it cannot be accessed for pitch detection and vice versa. Reading and pitch detection will be further expanded upon below. State 2 is the recording state. A recorder thread and playback thread are run in this state. This will also be expanded below. State 3 is the recorder state. This is an idle state. From this state the program can uniquely access the recorder or recording state. And lastly, state 4 is the off state. This is reachable by nearly every other state.

The main thread monitors the buttons and changes the guitar tuner state accordingly. When first given power, the guitar tuner is off. It's ON button must be pressed for 2 seconds to turn it on. The guitar tuner begins in tuner mode which can be changed by the user. The button monitoring and state changing section is encompassed by a while loop that checks whether the guitar tuner should be on or off (ON Button). When off, the guitar tuner will continually check if the on button has been pressed for more than 2 seconds. Once on, the tuner will continuously poll buttons. The assumption that only one button will be pressed at a time affords the program to track how long the on button has been pressed each time it is polled and found to be pressed.

The autocorrelation method is the pitch detection method utilized by the guitar tuner. In the tuner mode, sound is sampled at a rate of 22,050 Hz. The algorithm is set to run at 40 Hz, so each array it acts on will have a total of 552 samples. The algorithm will copy the array, multiply it by itself, then shift everything down by 1 sample. The result is maximized when the shift, known as the lag period, is equal to the signal's period. That's when the signal is best correlated with itself and produces the highest number. There are some problems which occur that have been mitigated through optimizations. To account for the decrease of overlapping samples, each multiplication is normalized by the sum of squares of the beginning and end of its overlapping section. The algorithm is also optimized to only check sample lags that correlate to desired frequencies. The highest desired frequency to detect is 360 Hz, an F above guitar's an E string. The period of this signal corresponds to a 55 samples of lag, including a buffer. The lowest desired frequency to detect is at 63 Hz and which has a period corresponding to 360 samples with padding. Given this, the algorithm only needs to check autocorrelation values for sample lags between 55 - 360 samples instead of all 552 samples. Once all these calculations are made, the algorithm determines the highest value. This is mostly likely the desired value. However, this sample lag could correspond a harmonic or octave of the actual pitch. To check for this, the maximum number of possible harmonics are found and their corresponding lags. The array is checked at each of these locations for other peaks. If all multiples are found to be within 90% of the original peak, this indicates that a harmonic or octave was most likely found. The lowest multiple is then selected as the proper sample lag. Then pitch frequency can be calculated by dividing the sampling rate by the sample lag.

Another important part of the design is how the program reads and writes data for the use of the audio cape. The audio cape used runs by default at a 96 kHz sampling rate with 16 bits bit depth when sampling audio data. The audio cape then uses the I²S protocol to communicate to the Beaglebone Black through two of the GPIO pins of the Beaglebone Black. Pin 25 of Port 9 is used to by the audio cape to send audio data to the Beaglebone Black while Pin 28 of Port 9 is used to send audio data from the Beaglebone Black to the audio cape. From the I²S protocol, the bit rate of the audio data sent on both ports is 1.536 MHz. Because of the very high bit rate, the timing of reading and writing the bits through the pins is crucial to the correctness of the program.

When reading in audio data from the audio cape, the program continuously poll the input from the pin until an edge in the signal occurs (whether from low to high or high to low) or a full sample has been read. If an edge occurs, the program thens calculate the amount of time that has passed between the last edge to figure out the amount of bits that has been sent and fills up the data array with 1s or 0s as required. If the program is set to have a different audio sampling rate (can be changed from the macros at the start of the program) than the default sampling rate of the audio cape, the program then uses the nanosleep operation to wait in between samples.

When sending audio data to the audio cape, the program calculates the time in between the bits sent to the audio cape. The program then sets the pin to either high or low based on the data stored, and uses the nanosleep operation to wait in between the bits sent.

Circuit Diagrams

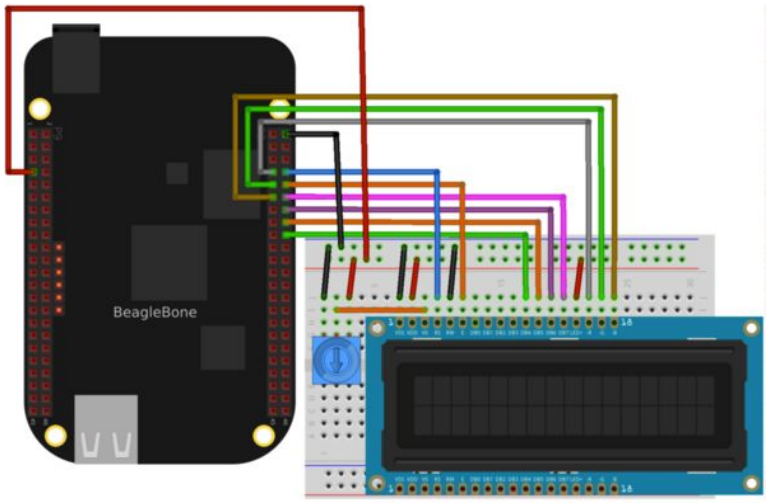


Figure 2. Circuit Diagram for BeagleBone Black and LCD screen

The following instructions outline how to connect the BeagleBone Black to the Adafruit LCD screen correctly:

Connect BeagleBone Black 5V power pin P9_7 to the power rail of the breadboard. From the power rail connect one outer lead of the potentiometer, LCD pin 2 (VDD), and LCD pin 15 (LED+). Connect BeagleBone Black ground pin P8_2 to the ground rail of the breadboard. From the ground rail connect the other outer lead of the potentiometer, LCD pin 1 (VSS), and LCD pin 5 (R/W). Connect the middle lead of the potentiometer to LCD pin 3 (V0/contrast). Connect BeagleBone Black pin P8_8 to LCD pin 4 (RS). Connect BeagleBone Black pin P8_10 to LCD pin 6 (E/clock enable). Connect BeagleBone Black pin P8_18 to LCD pin 11 (DB4). Connect BeagleBone Black pin P8_16 to LCD pin 12 (DB5). Connect BeagleBone Black pin P8_14 to LCD pin 13 (DB6). Connect BeagleBone Black pin P8_12 to LCD pin 14 (DB7). Connect

BeagleBone Black pin P8_7 to LCD pin 16 (-R/red). Connect BeagleBone Black pin P8_9 to LCD pin 17 (-G/green). Connect BeagleBone Black pin P8_11 to LCD pin 18 (-B/blue).

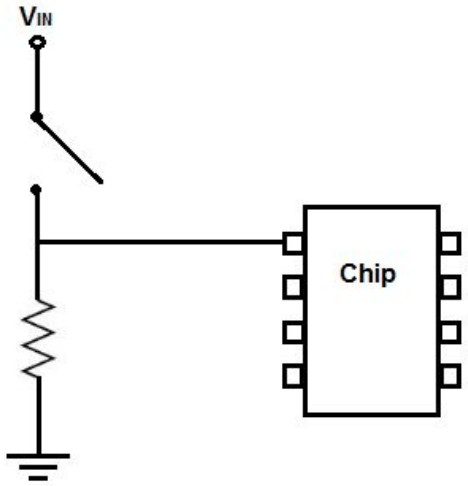


Figure 3. Pull Down Resistor Configuration

The buttons were attached to the BeagleBone Black using the pull down resistor configuration. The red button (stop button) is connected to Pin 13 of Port 8, while the black button (mode button) is connected to Pin 15 of Port 8, and the green button (replay button) is connected to Pin 13 of Port 8.

LCD Screen

Given the Adafruit RGB backlight negative character LCD screen selected for this project, it was not necessary to develop extensive driver software. Fortunately, Adafruit has provided a driver for the Beaglebone Black to interface with their hardware. Certain functions were necessary to program for our device to operate appropriately such as: enabling the display, setting the cursor to an explicit location, enabling cursor blinking, writing messages to the screen and clearing the LCD screen's content. These functions were key to our implementation and can

be found in the **Adafruit_Python_CharLCD** library. The only dilemma amid this library was the programming language employed. The entire library was written in Python unlike all of our code written in C. Therefore, a Python extension with C code was written to handle the exchange of data from C to Python.

The code needed to write the wrapper module is another C file containing a few special Python functions. Conventionally, the names of C extensions begin with an underscore so let's call our module _retrieveNote and write it in a file _retrieveNote.c (not to be confused with the retrieveNote.c file that was wrote previously).

In order to be able to access the C functions and types in the Python API, the first thing that was needed was the importation of the Python header, numpy array header and our retrieveNote function located in the _retrieveNote.c file.

```
1  #include <Python.h>
2  #include <numpy/arrayobject.h>
3  #include "retrieveNote.c"
```

Then, it was necessary to write the docstrings for our module and the function that our program is wrapping:

```
5  static char module_docstring[] =
6  "This module provides an interface for printing to LCD using C.";
7  static char* retrieveNote_docstring[] =
8  "Determine the correct pitch and appropiate frequency of a played note.";
```

Additionally, it's necessary to declare the function itself.

```
10 static PyObject *retrieveNote_retrieveNote(PyObject *self, PyObject *args);
```

At this point, this is the first time that we encounter any code Python-specific. The type **PyObject** refers to all Python types. Any communication between the Python interpreter and C code requires being passed down by **PyObjects** so any function needed to be able to call from Python must return a PyObject. A **PyObject** is just a struct with a reference count and a pointer

to the data encapsulated within the entity. This can be as simple as a double, or as complicated as a completely functional Python class.

The name that given to the function (retrieveNote_ retrieveNote) is a matter of convention. From Python, we're going to call the function with the command _retrieveNote.retrieveNote where_retrieveNote is the name of the module and retrieveNote is the name of the function. Since C doesn't have any concept of namespaces, the convention is to name C functions following the form.

The arguments for the function are pretty standard. In our case, the self object points to the module itself and the args object is a Python tuple of input arguments. It's possible to accept keyword arguments by including a third PyObject. However, that was not necessary for our application.

Now, it's important to discuss the contents of module. In this case, there is only going to be one function (called retrieveNote) so the method definition looks like:

```
12 static PyMethodDef module_methods[] =
13 {
14     {"retrieveNote", retrieveNote_retrieveNote, METH_VARARGS, retrieveNote_docstring},
15     {NULL, NULL, 0, NULL}
16 };
```

More functions can be added by adding more lines similar to the second one. This second line contains all the info necessary for the interpreter to link a Python call to the correct C function. The first string is the name of the function as it will be called from Python. The second object is the C function to link with, and the last argument is the docstring for the function. The third argument **METH_VARARGS** means that the function only accepts positional arguments.

The final step in initializing the new C module is to write an init{name} function. This function must be called init_retrieveNote where _retrieveNote is the name of the module.

```
PyMODINIT_FUNC init_retrieveNote(void)
{
    PyObject *m = Py_InitModule3("_retrieveNote", module_methods, module_docstring);

    if (m == NULL)
        return;

    // Load numpy functionality.
    import_array();
}
```

It’s fairly self-explanatory by this point to see what occurs here. Although, it’s important to note that if you want to use any of the functionality defined by numpy, it’s obligatory to include the call to import_array() (a function defined in thenumpy/arrayobject.h header).

Up to this point, we've written only approximately 26 lines of C code to set up a C extension module. All of these steps are common amid any modules written for extending Python functions in C, however, the details become somewhat less general as the wrapper is constructed.

Now, it's time to write the retrieveNote_retrieveNote function that was declared above. The args tuple will contain four doubles (the pitch frequency, highbound frequency of a given pitch range, lowbound frequency of the given pitch range and middle frequency of a given pitch range) and one numpy array for the notes that we are determining from an input signal. Below is an image showing the entire function, which is dissected line-by-line:

```c
static PyObject *retrieveNote_retrieveNote(PyObject *self, PyObject *args)
{
    double pitch, highBound, lowBound, mid;
    PyObject *note;

    // Parse the input tuple
    if (!PyArg_ParseTuple(args, "dddd0", &pitch, &highBound, &lowBound, &mid,
                          &note))
        return NULL;

    // Interpret the input objects as numpy arrays.
    PyObject *note_array = PyArray_FROM_OTF(note, NPY_STRING, NPY_IN_ARRAY);

    // If that didn't work, throw an exception.
    if (note == NULL)
    {
        Py_XDECREF(note);
        return NULL;
    }

    // Call the external C function to compute the pitch.
    retrieveNote();

    // Clean up.
    Py_DECREF(note);

    // Build the output tuple
    PyObject *ret = Py_BuildValue("dddd0", pitch, highBound, lowBound, mid, note);

    return ret;
}
```

Initially, the function must parse the input tuple using the PyArg_ParseTuple function. This function takes the tuple, a format and the list of pointers to the objects that are designated as input values. This format is comparable to the sscanf function in C, except the format characters are slightly different. In our example, d indicates that the argument should be cast as a C double and O is just a catchall for **PyObjects**. There isn't a specific format character for numpy arrays, therefore, we have to parse them as raw **PyObjects** and decipher them afterwards. If **PyArg_ParseTuple** fails, it will return **NULL**, which is the C-API technique for promulgating exceptions. That means that we should also return **NULL** instantly if parsing the tuple fails.

The next few lines show how to load numpy arrays from the raw objects. The **PyArray_FROM_OTF** function is a fairly general method for converting an arbitrary Python object into a numpy array that can be used in a standard C function. It is important to note that this creation method only returns a copy of the object if needed. The flags **NPY_STRING** and

NPY_IN_ARRAY ensure that the returned array object will be represented as contiguous arrays of C strings.

If a **PyObject** is returned from a function, given how memory management works in Python, it might be necessary to run **Py_INCREF** on it to increment the reference count. And when a new object is created within the Python function that you don't want to return, it would be necessary to run **Py_DECREF** on it before the function returns (even if the execution failed). This ensures a memory leak doesn't occur. With this in mind, is paramount that the function written is prudent in keeping track of memory usage.

The objects returned by **PyArg_ParseTuple** do not have their reference count incremented (the calling function still "owns" them), so it's not necessary to decrement the reference count of the *note object before returning. Conversely, **PyArray_FROM_OTF** does return an object with a +1 reference count. This means that it must call **Py_DECREF** with the *note_array objects as the first argument before returning from this function. If **PyArray_FROM_OTF** can't coerce the input object into a form digestible as a **numpy** array, it will return **NULL**. This can be seen on line 46 where **Py_XDECREF. Py_XDECREF** checks to ensure that the object isn't a **NULL** pointer before trying to decrement the reference count.

On line 50, the C function is finally called that was intended to wrap. Finally, **Py_BuildValue** is utilized to create the output tuple. If **Py_ParseTuple** has syntax similar to **sscanf**, then **Py_BuildValue** is the analog of **sprintf** with the same format characters as **Py_ParseTuple**. Here, we don't need to **Py_INCREF** the return object since **Py_BuildValue** does that for us. Given we have multiple values needed to be available in Python, we must return more than one variable. Luckily, Python supports this functionality by using tuples.

The last thing regarding the Python extension code was to handle compilation and linkage, so this module can be called form Python. From heralded best way to do this is by using the built-in Python distribution utilities. Conventionally, the build script is called setup.py and the file is actually tremendously simple:

```
from distutils.core
import setup, Extension
import numpy.distutils.misc_util

setup
(
      ext_modules=[Extension("_retrieveNote", ["_retrieveNote.c", "retrieveNote.c"])],
      include_dirs=numpy.distutils.misc_util.get_numpy_include_dirs(),
)
```

Running this small python script will compile and link the source code and create a shared object called_retrieveNote.so in the same directory. Then, from the Python shell or script, simply importing the module allows use of the retrieveNote function:

```
import _retrieveNote
import Adafruit_CharLCD as LCD
```