Software Formalization

Year: 2017 Semester: Spring Team: 12 Project: Guitutar
Creation Date: 22 February 2017 **Last Modified:** 22 February 2017
Author: Brian Rieder **Email: brieder@purdue.edu**

Assignment Evaluation:

Item	Score (0-5)	Weight	Points	Notes
Assignment-Specific Items				
Third Party Software	5	x2	10	
Description of Components	5	X3	15	
Testing Plan	4.5	x3	13.5	
Software Component Diagram	5	x4	20	I love me them Mealy-Moore state diagrams!
Writing-Specific Items				
Spelling and Grammar	5	x2	10	
Formatting and Citations	5	x1	5	
Figures and Graphs	5	x2	10	
Technical Writing Style	5	x3	15	
Total Score	98.5/100			

5: Excellent 4: Good 3: Acceptable 2: Poor 1: Very Poor 0: Not attempted

General Comments:
I'm having trouble figuring out if there's anything you're missing, and I find this report to be so thorough that I'm really reaching here. Excellent work, and I mean it. You have a really great handle on all the necessities of your software and it seems that every time I think you need a function for something, I find it in the assignment, already well thought through. Great work.

The comments I gave are for the assignment's improvement, no doubt, but you are already well on your way!

1.0 Utilization of Third Party Software

Within the context of Guitutar's main package, the only code that is intended to be utilized is code constructed by the team and the code generated by the MPLAB Harmony Integrated Software Framework - no additional third party software is planned to be employed. As such, the only code used within the application of the software for the guitar itself will be custom-made with additional module support through Harmony. The primary driving logic running the project that drives the operation of the device will be written by the team, but the UART and GPIO drivers have been pre-generated by Harmony. MPLAB Harmony is an integrated firmware development platform [1] that the team is utilizing to ease the development process through driver generation, use of peripheral libraries, and use of a configurable emulation of object oriented design. Harmony generated code for GPIO port control and for use to implement Bluetooth, such as USART and the SPP Bluetooth Stack Library, are being included within the project for the body of the main device to receive inputs and communicate to the remote interface. Microchip provides a standard license for usage of its libraries within an application that is included in the Microchip Libraries for Applications (MLA) free of charge that also cover the included functionality of Harmony outside of usage of RTOS that falls outside of the scope of this project [1][2].

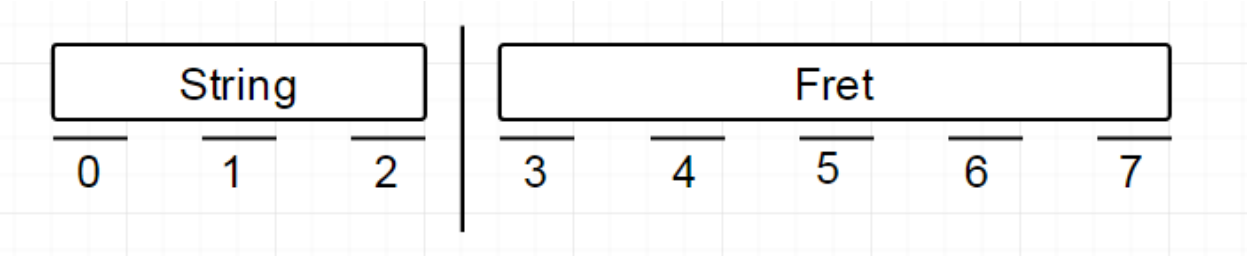
In addition to the physical Guitutar guitar, the team is developing an Android application to drive the operation of the device through the aforementioned user interface. In order to construct this application, it is a requirement of the software to utilize the Android API and developer packages such as android.bluetooth, android.bluetooth.le, android.os.storage, and other interface packages. These packages will be used to generate a user interface to display to the user, access predefined song choices that are stored within the app, and to send inputs to the primary device to drive the program flow and determine what notes need to be pressed. Usage of android.os.storage will be for access to classes that represent the system storage service [3] in order to store the songs that will be relayed to the guitar. The android.bluetooth and android.bluetooth.le packages will both be used to scan for and connect to the device as well as maintain serial data transmission connections through the Bluetooth standard [4][5]. Additional application libraries will be used for the generation and display of the user interface and graphics, but are not listed here for conciseness as well as them having the purpose of simply interfacing with the user and not facing the primary Guitutar device is any way. The preferred license for development using Android under the Android Open Source Project is the Apache 2.0 software license [6]. As this project is small scale and not an industrial application, all development can fall under this license if the product is not to be shipped on a device as additional software. Given that the project is a single time installation on the desired host device and is not being used for marketing, usage of the Android API is permitted without issue.

2.0 Description of Software Components

The software for Guitutar breaks down into several major components within both the physical Guitutar device itself and the user control interface application being run remotely from a phone. Both the application and the device will run project-specific interfacing software to communicate via Bluetooth, the application using the Android API and the device using a UART driver to transmit and receive serial information. The device itself must also utilize GPIO to register inputs from the user on the string-fret switch matrix and drive the program flow shown in Figure 2.1.

2.1 Note Encoding Structure

As was described in the Software Overview document, it is a requirement of the design to specify an encoding for notes to be transmitted by the application via Bluetooth, communicated to the primary device via a serial bus, and then displayed by the device to the user. In order to do so, the note encoding algorithm has been defined as a byte in which the first three bits indicate the string and the last five bits indicate the fret as shown below:



In order to recognize this, a single byte data type must be used to access the individual bit fields of the note. As such, this design will incorporate a union containing a string field and a fret field:

- ***union*** **Note**
  - ***uint8_t*** **note_byte** - A full byte representation of the note that can be set given the output from the serial bus. In addition to easier value-setting and aiding debugging, having an entire byte here allows us to break it apart within the struct immediately following.
  - ***struct*** (wrapper for the string and the fret, contained within the union)
    - ***uint8_t*** **string : 3** - The leading three bits of the byte that represent the string. This serves as the X component of the switch coordinate system that runs along the fret.
    - ***uint8_t*** **fret : 5** - The trailing five bits of the byte that represent the fret. This serves as the Y component of the switch coordinate system running along the truss rod.

Having this standard is essential for all elements to function in real time and to interpret what data coming off the serial bus to the primary device for display. Placing the note within the union allows bits to be sliced off the bit itself to not only access the full byte, but to also have a breakdown to string and fret level for self-documenting code that doesn't need bit-slicing methods.

2.2 Song Structure

With the note encoding structure specified in Section 2.1 above, the Android application will send note data on a serial bus to the microcontroller. For the microcontroller to be able to properly parse string of note bytes being received, a methodology for storing and computing the information is required. A structure will be defined to store the information for the song as such:

- *struct* **Song**
  - *int* **size** - The number of notes in the song. This is used to allocate a variable number of bytes for note storage.
  - *int* **bpm** - The beats per minute (BPM) that the song should be played at. This will be used in calculating how much delay in time should be given between displaying notes in a song.
  - *Note** **notes** - A pointer to the first note of the song used to access the note information. The notes will be stored consecutively therefore each note can be individually indexed.

When transferring the song data from the Android application to the microcontroller, the size of the song will be sent as the first two bytes and stored into the structure. The memory for notes will then be allocated at that time using C's malloc function. Next the byte for bpm will be transmitted and stored into the struct. Finally, each byte of the song's notes will be sent and received one by one while keeping track of the number of bytes that have been read. Once the number of bytes read matches the size of the song, a sanity check will be run making sure the last note is the end byte (11100000). If this is found not to be the case, then it is known that there has been an error loading the song. This will be done within a function with the definitions below:

int **load_song**(*Song_t** song)

- Parameters
  - *Song_t** **song** - The location to store the information read from the device communication
- Local Variables
  - *int* **index** - A index used to count the number of notes read into the song.
- The return value will be an error/success indicator used notify the system if it should continue or retry the process.

This function will make use of the functions made for communication specified in the section below (Section 2.3).

2.3 Application/Device Communication Interface (Bluetooth and USART)

To create an interface between the application and the Guitutar primary device, Bluetooth is intended to be used as the wireless communication interface. To implement Bluetooth on the application device, an Android device with wireless Bluetooth capabilities must be used: for development and demonstration, it was determined that a phone running stock Android should be used to prevent variables, so a Huawei Nexus 6P has been chosen as the development device. For the primary device to receive the information, a serial interface must be established between it and the paired Bluetooth controller - for this purpose, UART has been chosen to the implemented serial interface that drives communication between the device and the application.

The USART driver will be present on the primary device and is created via a driver generated by MPLAB Harmony. To initialize and interface with USART, the primary device must use several functions as described below:

- USART Initialization
  - `SYS_MODULE_OBJ DRV_USART_Initialize(const SYS_MODULE_INDEX index, const SYS_MODULE_INIT * const init)`
    - The USART driver initialization is configured through the DRV_USART_INIT data structure that is passed into this function. *This function must be called for the USART driver to be operable.*
  - `DRV_HANDLE DRV_USART_Open(const SYS_MODULE_INDEX index, const DRV_TO_INTENT ioIntent)`
    - To use the USART driver, the application must open the driver using this function. After it has been called, it determines the functionality of the Write and Read function based on the intent to read/write and whether operations are blocking or non-blocking.
    - The returned handle is valid until the DRV_USART_Close routine is called.
- USART Communication
  - `uint8_t DRV_USART_ReadByte(const DRV_HANDLE handle)`
    - Using the previously defined USART handle, reads a single byte from the serial line.
    - A usage example from Microchip is shown below [7]:

```
DRV_HANDLE      myUSARTHandle;    // Returned from DRV_USART_Open
char            myBuffer[MY_BUFFER_SIZE];
unsigned int    numBytes;

numBytes = 0;
do
{
    if( DRV_USART_TRANSFER_STATUS_RECEIVER_DATA_PRESENT & DRV_USART_TransferStatus(myUSARTHandle) )
    {
        myBuffer[numBytes++] = DRV_USART_ReadByte(myUSARTHandle);
    }

    // Do something else...

} while( numBytes < MY_BUFFER_SIZE);
```

- *void* **DRV_USART_WriteByte**(*const DRV_HANDLE* handle, *const uint8_t* byte)
  - Using the previously defined USART handle, writes a single byte on the serial line.
  - A usage example from Microchip is shown below [7]:

```
DRV_HANDLE      myUSARTHandle;    // Returned from DRV_USART_Open
char            myBuffer[MY_BUFFER_SIZE];
unsigned int    numBytes;

// Preinitialize myBuffer with MY_BUFFER_SIZE bytes of valid data.

numBytes = 0;
while( numBytes < MY_BUFFER_SIZE );
{
    if( !(DRV_USART_TRANSFER_STATUS_TRANSMIT_FULL & DRV_USART_TransferStatus(myUSARTHandle)) )
    {
        DRV_USART_WriteByte(myUSARTHandle, myBuffer[numBytes++]);
    }

    // Do something else...
}
```

- USART De-initialization
  - *void* **DRV_USART_Close**(*const DRV_HANDLE* handle)
    - Invalidates the used device handle and closes the connection. The only parameter passed is the initial return value of the Open routine.
    - There is no necessity in verifying that the routine completed as the driver will abort any ongoing operations when this routine is called [7].

The primary usage of USART will manifest itself within the ReadByte function, as the device will predominantly be reading bytes from the remote application to construct the song and to receive configuration data regarding tempo and execution.

The Bluetooth driver being used for the Android application is part of the Android API and contains a substantial amount of functionality within the *android.bluetooth* package previously detailed in the third party software section. The classes necessary for the use of Bluetooth within

the application are shown below:

- **`BluetoothAdapter`**
  - Used as a representation of the local Bluetooth adapter. This will be the adapter physically located on the phone being used to communicate with the Guitutar primary device [8].
  - Allows for the discovery and pairing of other Bluetooth devices, specifically the primary device.
    - Due to the security of the Bluetooth interface, only the primary device will connect to the phone via this app and will disallow any possible security concerns driven by opening the connection.
- **`BluetoothDevice`**
  - Used as a representation of a remote Bluetooth device. This will be the Bluetooth radio located within Guitutar's primary packaging [8].
  - Serves as the secondary device for communication via a Bluetooth socket.
- **`BluetoothSocket`**
  - The pseudo-network socket that serves as the connection point between two Bluetooth devices that permits the flow of data between the devices [8].
  - This is the primary I/O interface class that will function between Guitutar and the Android application.

In addition to these classes, the primary classes to be used with the BluetoothSocket to exchange data are the following:

- **`InputStream`**
  - Abstract class that represents an input stream of bytes [9].
  - The relevant methods, excluding a trivial constructor, are as follows:
    - *`int`* **`available`**`()`
    - *`int`* `read()`
      - Reads the next byte of data from the input stream.
      - *`int`* **`read`**`(`*`byte[]`*` b, `*`int`*` off, `*`int`*` len)`
        - Reads `len` bytes of data and stores them into an array of bytes `b` starting at index `off`.
        - The number of bytes read is returned as an integer.
      - *`int`* **`read`**`(`*`byte[]`*` b)`
        - Reads `b.length` number of bytes and stores them into an array of bytes `b` starting at index 0.
        - Has the same effect as **`read`**`(b, 0, b.length)`.
    - *`long`* **`skip`**`(`*`long`*` n)`
      - Skips over and discards n bytes of data from this input stream.
      - While this may not be used explicitly within the project, it serves the purpose of throwing out data if deemed necessary and supports debugging.

- **`OutputStream`**
  - Abstract class that represents and output stream of bytes. An output stream accepts output bytes and sends them to some sink [10].
  - The relevant methods, excluding a trivial constructor, are as follows:
    - *`void`* **`flush`**`()`
      - Flushes the output stream and forces any buffered output bytes to be written out.
    - *`void`* **`write`**`(`*`byte[]`*` b)`
      - Writes b.length bytes from array b to the output stream.
    - *`void`* **`write`**`(`*`byte[]`*` b, `*`int`*` off, `*`int`*` len)`
      - Writes len bytes from the specified byte array starting at offset off to this output stream.
    - *`void`* **`write`**`(`*`int`*` b)`
      - Writes the specified byte b to the output stream.
      - The byte to be written is the eight low-order bits of the argument b, whereas the 24 high-order bits of b are ignored.

The above classes are native to Java and allow direct byte transmission to transmit at the same data granularity and speed as the USART driver.

2.4 String-Fret Switch Matrix GPIO

In order for the Guitutar's primary device to receive input as to the correctness of a played note, a switch interface was deemed to be required. As such, a switch matrix was designed such that when pressed a signal would be sent back to the microprocessor to a GPIO input port that registered that a specific fret was pressed.

While the signal coming back from the switch matrix indicates that a fret has been pressed, a software solution to reduce the number of traces by a factor of six is to alternate the activation of each string in the matrix. As such, each string will only be receiving power 1/6th of the time, a continually running process within a software while loop. This is simply a function involving six single output ports that flips a bit positively every sixth cycle.

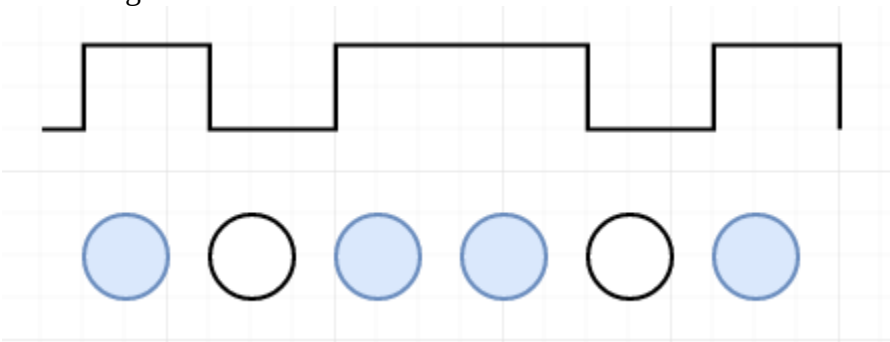
In order to operate with GPIO, another MPLAB Harmony library will be used in order to interface with the GPIO peripheral: the Harmony Ports Peripheral Library. Within this library are the following functions that will be implemented within this design in order to receive user inputs and to power strings:

- User Input Interfacing
  - The following two functions will be used to receive the inputs from the switch matrix on the relevant I/O ports:
  - *`bool`* **`PLIB_PORTS_PinGet`**`(`*`PORTS_MODULE_ID`*` index, `*`PORTS_CHANNEL`*` channel, `*`PORTS_BIT_POS`*` bitPos)`

        - Reads a single boolean value from the bit on port `channel` in the specified `bitPos`.
    - *PORTS_DATA_TYPE* **PLIB_PORTS_Read**(*PORTS_MODULE_ID* index, *PORTS_CHANNEL* channel)
        - Reads the entirety of the data port specified within channel.
        - This is effectively a bulk version of PinGet for an alternate implementation.
- String Power Alternation
    - The following two functions will be used to alternate the powering of the strings in order to effectively multiplex the input ports:
    - *void* **PLIB_PORTS_PinWrite**(*PORTS_MODULE_ID* index, *PORTS_CHANNEL* channel, *PORTS_BIT_POS* bitPos, *bool* value)
        - Writes `value` to bit `bitPos` on the specified port `channel`.
    - *void* **PLIB_PORTS_Write**(*PORTS_MODULE_ID* index, *PORTS_CHANNEL* channel, *PORTS_DATA_TYPE* value)
        - Writes `value`, a combination of bits, to the specified port `channel`.
        - This is effectively a bulk version of PinWrite for an alternate implementation.

2.5 LED Matrix Display Interface

In order to display the expected notes to the user, an array of LEDs has been configured such that they are receiving inputs transitively through fret-dedicated I/O pins from the microcontroller. Each row of six LEDs associated with a fret is driven by a serial-to-parallel shift register whose serial input is derived from the microcontroller's dedicated I/O pin. As such, to display a note or chord, a sequential signal is sent to drive the shift register's outputs through a cyclic six-part wave as shown in the signal below:



Similar~~ly~~ to the string-fret switch matrix, the LED matrix receives all of its inputs generic I/O ports, but, however, does not return anything back to the microcontroller and solely acts as a display interface. Just as with the switch matrix, the Harmony Ports Peripheral Library and its functions will be used to drive the LED array. Within this library are the following functions that

will be implemented within this design in order to transmit display bits:

- *void* **PLIB_PORTS_PinWrite**(*PORTS_MODULE_ID* index, *PORTS_CHANNEL* channel, *PORTS_BIT_POS* bitPos, *bool* value)
  - Writes `value` to bit `bitPos` on the specified port `channel`.
- *void* **PLIB_PORTS_Write**(*PORTS_MODULE_ID* index, *PORTS_CHANNEL* channel, *PORTS_DATA_TYPE* value)
  - Writes `value`, a combination of bits, to the specified port `channel`.
  - This is effectively a bulk version of PinWrite for an alternate implementation.

2.6 Guitutar Device Control Flow

A state machine must be constructed in order to drive the flow of program on the primary device which will determine how all other aspects of the design are operating. As is shown in Appendix 2, a state machine has been created to detail the sequencing of logic within the operation of the device. To realize all of the previously mentioned functionality, the following functions are to be implemented with all references to states being those within Appendix 2:

- *void* **wait_for_song_select**()
  - *Relevant States:* Select Song
  - Executes a wait loop until input is received via UART from the Bluetooth controller that indicates that a song has been chosen.
- *Song_t* **download_song**()
  - *Relevant States:* Select Song
  - After a song has been chosen, it must be downloaded via Bluetooth from the Android device to the primary Guitutar device. This is done via the communication interface detailed in Section 2.3 until the “stop” note detailed in Section 2.2 has been received.
- *Mode* **select_mode**()
  - *Relevant States:* Mode Select
  - After the song has been downloaded, the user must select a mode for the device to operate in. At the time of this document, there are only two modes planned for implementation, LEARNING and REALTIME, but the design is such that there can be expansion in the future.
  - Mode enumeration: In order to allow expansion to other modes, it was implemented as an enumeration rather than a boolean and was chosen not to be simply an integer for the sake of self-documentation and readable code. Shown below is the current enumeration.

```
enum Mode {
      LEARNING,
      REALTIME
};
```

- *void* **display_note**(Note_t note)
  - *Relevant States:* Display Next Note (LEARNING/REALTIME)
  - As the song is parsed through by the device, a single note will be displayed at a time. This function serves the purpose of setting the GPIO outputs to the shift registers properly to display the note.
  - As a single note is passed in, the majority of the tasking here is handled by the persistent nature of the GPIO driver, so there does not need to be a loop broken by a correct/fully delayed note value.
  - In the event that a note with a string value of 3'b111 is read, this note will not be displayed, but will rather be interpreted with two special cases:
    - If the note's fret value is 5'b00000, this is the "song finished" note, so execution is completed and the device moves back to song selection.
    - For any other fret value in REALTIME mode, the note's fret value then becomes the number of 16th notes that the device must wait before reading the next note.
    - In LEARNING mode, these delay values are ignored and the next note is polled.
- *void* **wait_for_input**()
  - *Relevant States:* Wait for Input
  - While in the LEARNING mode, the device will wait for a user to enter an input with the switch matrix that is triggered to be read by the piezoelectric disc sensor on the body of the guitar.
  - The transition out of this state will be a positive trigger from the piezoelectric sensor that indicates that a note was played.
- *void* **delay**(*int* time)
  - *Relevant States:* Delay
  - While in the REALTIME mode, it is necessary to read a delay value that is derived from the last five bits of a delay note of the form 8'b111XXXXX. The delay value is passed in as an integer that is multiple of sixteenth notes.
  - To accommodate delays longer than 31 sixteenth notes, consecutive delays will be treated as composite and combined to make a longer rest.
- *bool* **analyze_note**(*Note_t* played_note)
  - *Relevant States:* Analyze Note (LEARNING/REALTIME)
  - When the piezoelectric disc indicates that a note is played, that note is to be analyzed regardless of mode. This function serves to determine whether the note that the user played was correct or incorrect and to return an indicative boolean.
  - In the LEARNING mode, note analysis determines transitioning to the next note of staying on the current note until it is correct.
  - In the REALTIME mode, note analysis is primarily collected to determine statistics regarding the number of times that the user has played correct notes at the end of the song

3.0 Testing Plan

3.1 Note and Song Decoding

Vitality of Successful Verification (0-10): 7

The ability to cast an input to the note and song data structures is vital for the performance of the device, but can only be tested on a basic level once the communication interface has been put into place. Once the product is able to read bytes of data, it is still dependent on the functionality of the LED matrix display interface to show that it is functional. As the decoding aspect of the software is so heavily dependent on the other components being completed, final testing of note and song decoding can only be tested independently from the control flow of the device. The following tests are the tests that will be performed to ensure functionality:

1. Predefined pattern test: Ensure that predefined bit patterns can be decoded and stored properly. This is the only test that can be performed independently of all functions through the use of the MPLAB debugger.
2. LED tandem test: Ensure that predefined bit patterns can be decoded and displayed across a row of LEDs. This shows interconnectivity of the LED matrix and the decoder.
3. UART tandem test: Ensure that a serial stream coming from UART can be properly decoded and displayed across LEDs.

Once the third test is complete, decoding is shown to be entirely functional independent of the flow logic that drives the program on the main device, thus showing the ability to function in the final design.

3.2 Application/Device Communication Interface (Bluetooth and USART)

Vitality of Successful Verification (0-10): 8

One of the largest portions of the entire project, the remote-to-device communication interface ranks only below the required user interface functions in importance. Without the communication interface, there is no way to transfer songs from the application to the device in any mode or to control the tempo in “Real Time” mode. The following test will be performed to ensure proper functionality:

1. Bluetooth connectivity test: Ensure that both the Android device and the primary device are able to recognize each other and pair via Bluetooth. While a basic test, this can be shown basically through the Android device’s launcher’s user interface.
2. Bluetooth data reception test: Ensure that the Android device transmitting via Bluetooth is *visible* from the perspective of the primary device. This is a basic does-it-turn-on test, where the output of the Bluetooth controller should indicate some value being received - not necessarily the correct value.
3. UART reading test: Ensure that the Android device transmitting via Bluetooth is *visible and correct* from the perspective of the primary device.
4. UART writing test: Ensure that the primary device transmitting via Bluetooth is *visible and correct* from the perspective of the Android device.

If all of the above tests work, it has been shown that a full connection has been made between the primary device and that the connection is functional in both directions, thus showing the ability to function in the final design.

3.3 String-Fret Switch Matrix GPIO

Vitality of Successful Verification (0-10): 10

The functionality of the switch matrix is vital for the usage of the device - if no inputs can be taken in, there is no way to progress through a song and there is no way to verify correctness. The following tests will be performed to ensure proper functionality:

1. Basic input test: Ensure by powering an LED that when an an input is brought high on a port from a basic switch that the LED illuminates.
2. Toggle test: Ensure that an LED can be toggled off and on when an input is brought high on a port from a basic switch.
3. Multiple LED test: Ensure with six LEDs that pressing a button can sequence which LED is powered on by setting only the bit of the next LED.
4. Multiple switch test: Ensure that given six switches and six LEDs that progression through the LED sequence can only occur when the correct and expected switch is pressed.

The above tests functioning demonstrates the ability to verify an input and sequence LEDs based on inputs, thus showing the ability to function in the final design.

3.4 LED Matrix Display Interface

Vitality of Successful Verification (0-10): 9

The function of the LED matrix is secondarily vital, as a display interface must be given to the user. The LED matrix is not rated at a 10, however, because in the event that the matrix ends up functioning incorrectly, the design permits an alternative display interface. The following tests will be performed to ensure proper functionality:

1. Basic illumination test: Ensure that an LED illuminates when the output from the connected port is brought high.
2. Surface mount test: Ensure that the design-specific surface mounted LEDs illuminate when the output from the connected port is brought high.
3. Shift register test: Ensure that the design specific shift register can receive a bit pattern and illuminate LEDs accordingly (i.e., bit pattern 010110 should illuminate the LEDs on the A, G, and B strings)
4. Power consumption test: Ensure that the shift register can adequately provide power when given the bit pattern 111111, thereby illuminating all of the LEDs.
5. Multiple row test: Ensure control over multiple rows of LEDs from two pins on the microcontroller in tandem.

The above tests show that each row can be controlled independently and completely, thus showing the ability to function in the final design.

3.5 Guitutar Device Control Flow

Vitality of Successful Verification (0-10): 6

Ultimately, the device must function as an entire package, which cannot happen without a primary driver of the sequence of the programming. As the control flow cannot happen without all of the other devices functioning, the control flow is the lowest priority testing component during the development process.

1. Song selection test: Ensure that selection of a song from the Android device results in the downloading of the song via Bluetooth to the primary device.
2. Mode selection: Ensure that selection of a mode results in the display of a note and execution within the selected mode.
3. LEARNING mode test: Ensure that execution when LEARNING mode is selected does not transition until a note is played. When the song is finished, the guitar should no longer accept input and wait for the user to select a song on the Android device.
4. REALTIME mode test: Ensure that execution when REALTIME mode is selected transitions autonomously and does not wait for the user. When the song is finished, the guitar should no longer accept input and wait for the user to select a song on the Android device.

The control flow has several tests to ensure its functionality due to the fact that there are a number of corner cases and state transitions that need to be tested exhaustively. Once all state transitions are complete and the system works as a whole, the final design is shown to be completely functional.

4.0 Sources Cited:

[1] Microchip Technology, Inc. (n.d.). *MPLAB Harmony* [Online]. Available: http://www.microchip.com/mplab/mplab-harmony

[2] Microchip Technology, Inc. (n.d.) *Microchip Libraries for Applications* [Online]. Available: http://www.microchip.com/mplab/microchip-libraries-for-applications

[3] Google, Inc. (n.d.). *Android Developer Reference: android.os.storage* [Online]. Available: https://developer.android.com/reference/android/os/storage/package-summary.html

[4] Google, Inc. (n.d.). *Android Developer Reference: android.bluetooth* [Online]. Available: https://developer.android.com/reference/android/bluetooth/package-summary.html

[5] Google, Inc. (n.d.). *Android Developer Reference: android.bluetooth.le* [Online]. Available: https://developer.android.com/reference/android/bluetooth/le/package-summary.html

[6] Google, Inc. (n.d.) *Android Licenses* [Online]. Available: https://source.android.com/source/licenses.html

[7] Microchip Technology, Inc. (n.d.) *MPLAB Harmony Driver Libraries* [Online]. Available:

http://ww1.microchip.com/downloads/en/DeviceDoc/MPLAB%20Harmony%20Driver%20Libraries_v110.pdf
[8] Google, Inc. (n.d.) *Bluetooth API Guide* [Online]. Available: https://developer.android.com/guide/topics/connectivity/bluetooth.html
[9] Google, Inc. (n.d.) *Java I/O Class: InputStream* [Online]. Available: https://developer.android.com/reference/java/io/InputStream.html
[10] Google, Inc. (n.d.) *Java I/O Class: OutputStream* [Online]. Available: https://developer.android.com/reference/java/io/OutputStream.html

Appendix 1: Software Component Diagram

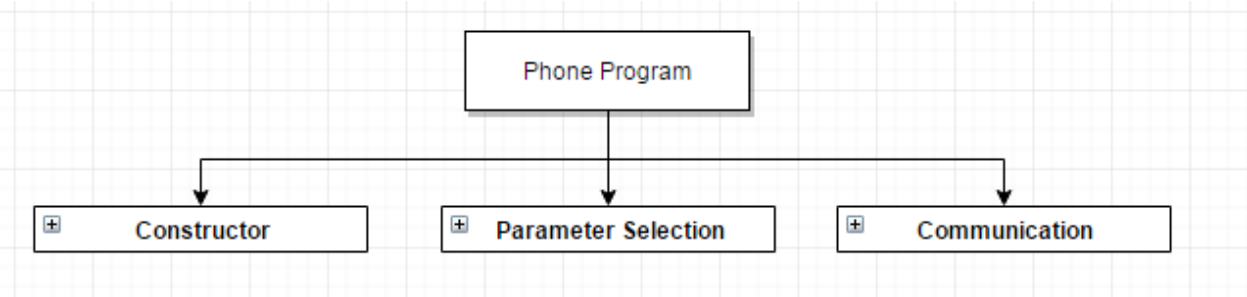


Figure 1.1 - Phone Software Overview

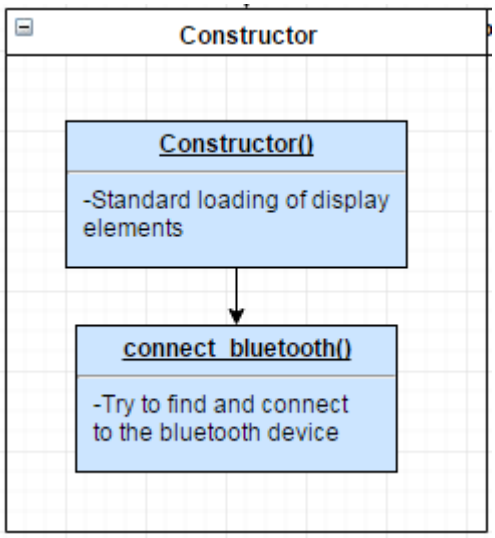


Figure 1.2 - Phone Constructor/Initialization

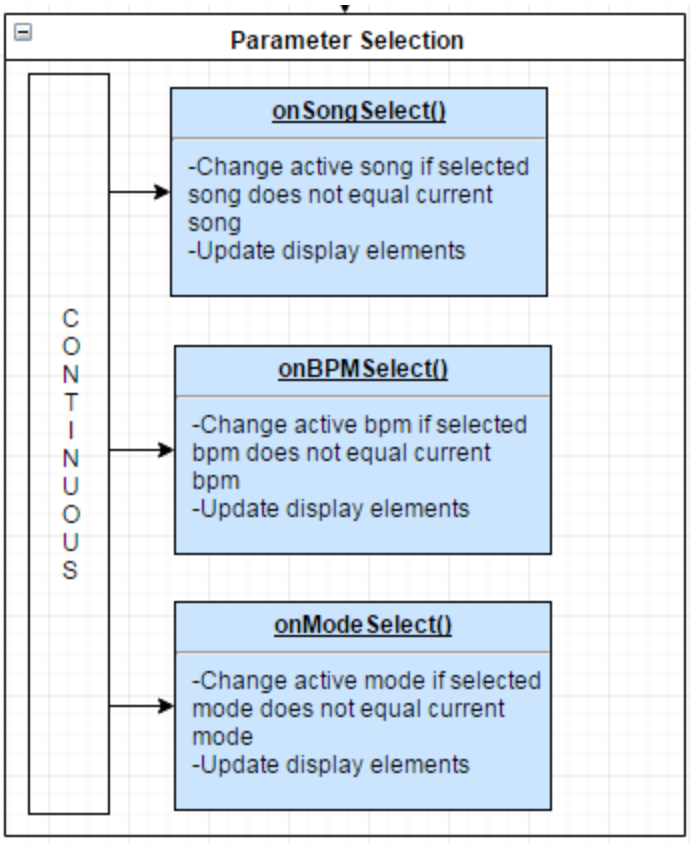


Figure 1.3 - Phone Parameter Selection

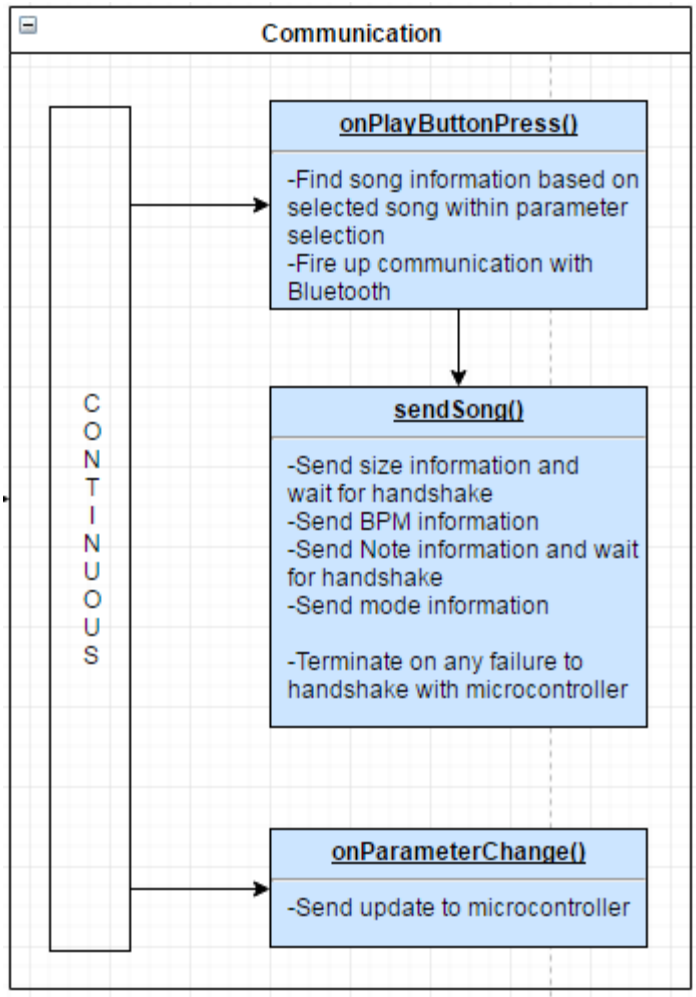


Figure 1.4 - Phone Communication

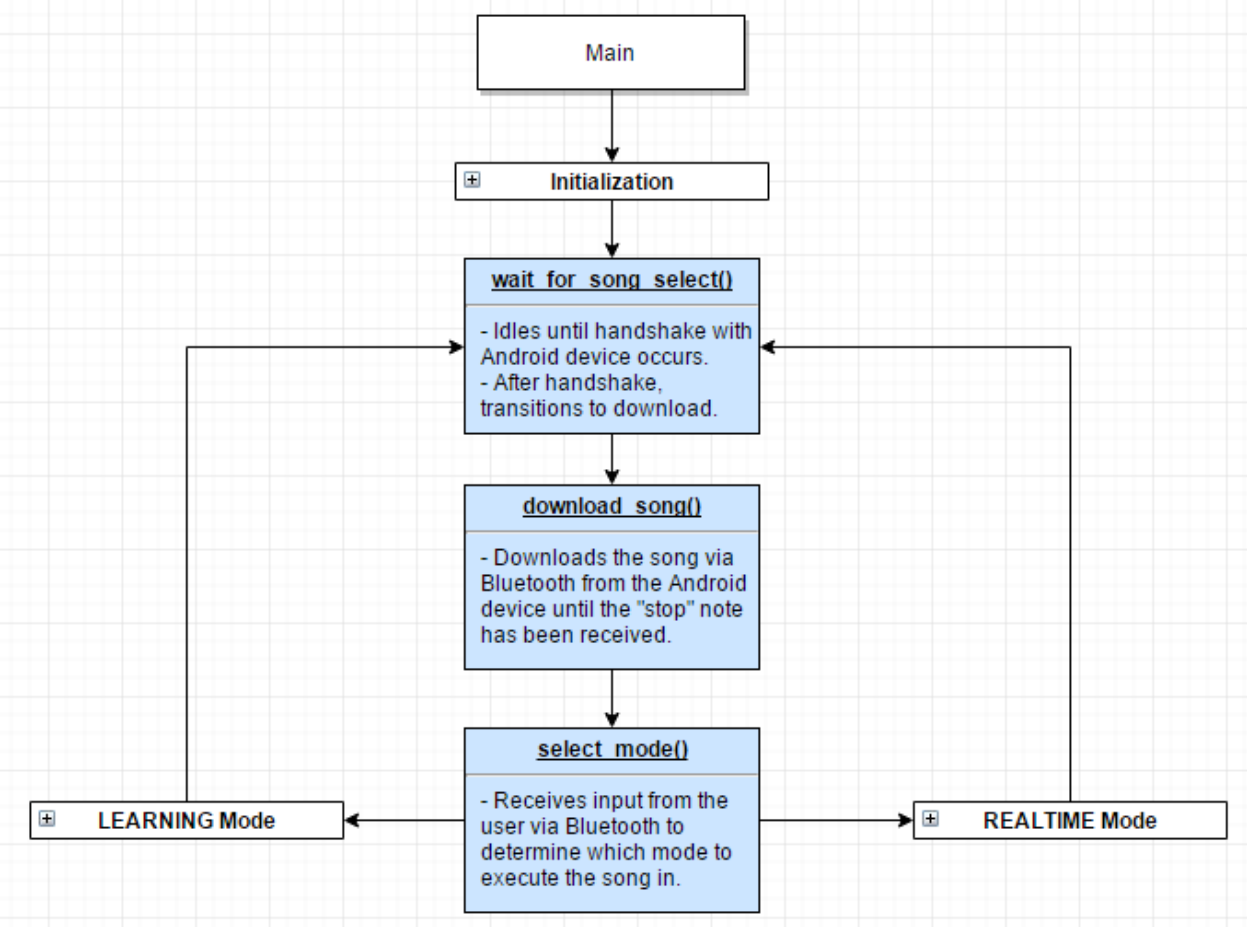


Figure 1.5 - Mricro Software Overview

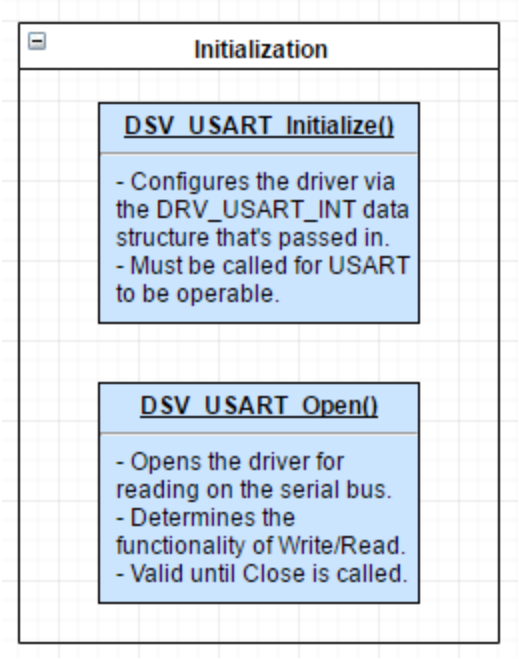


Figure 1.6 - Micro Initialization

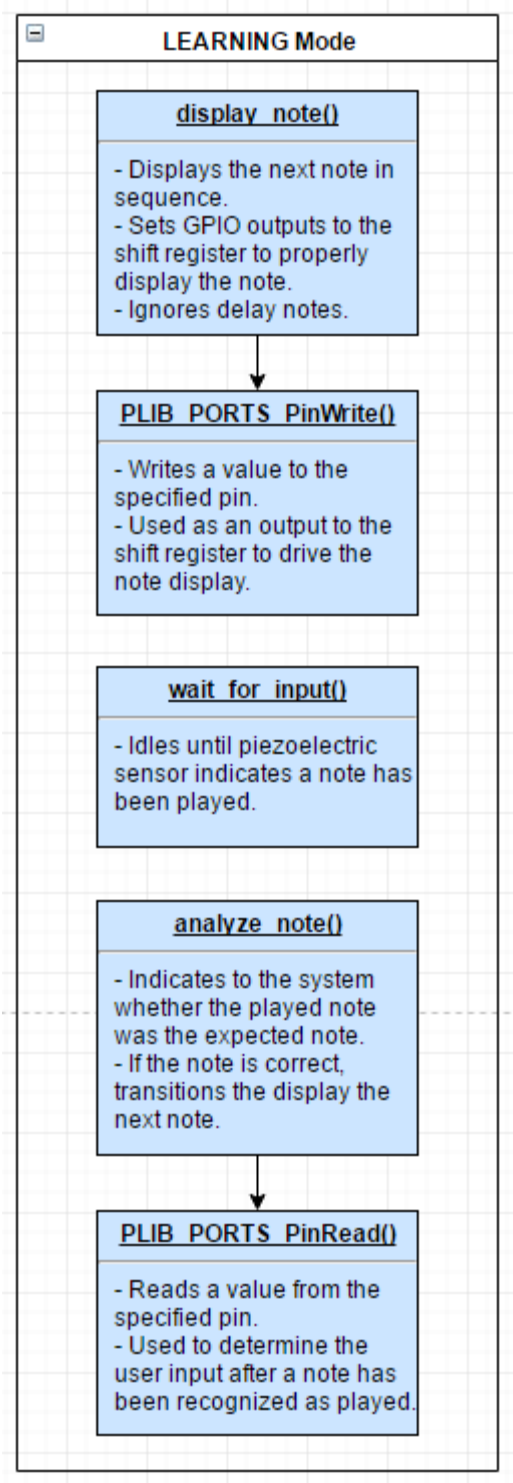


Figure 1.7 - Micro Learning Mode

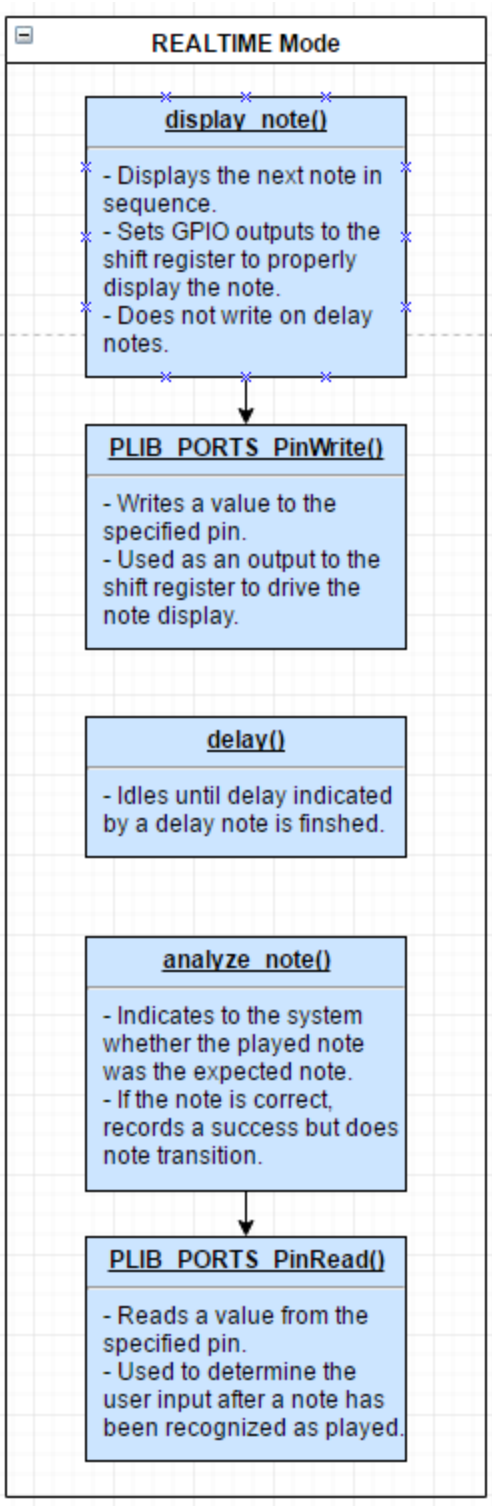

REALTIME Mode
display_note()
- Displays the next note in sequence.
- Sets GPIO outputs to the shift register to properly display the note.
- Does not write on delay notes.
PLIB_PORTS_PinWrite()
- Writes a value to the specified pin.
- Used as an output to the shift register to drive the note display.
delay()
- Idles until delay indicated by a delay note is finshed.
analyze_note()
- Indicates to the system whether the played note was the expected note.
- If the note is correct, records a success but does note transition.
PLIB_PORTS_PinRead()
- Reads a value from the specified pin.
- Used to determine the user input after a note has been recognized as played.

Figure 1.8 - Micro Real Time Mode

Appendix 2: State Machine Diagram

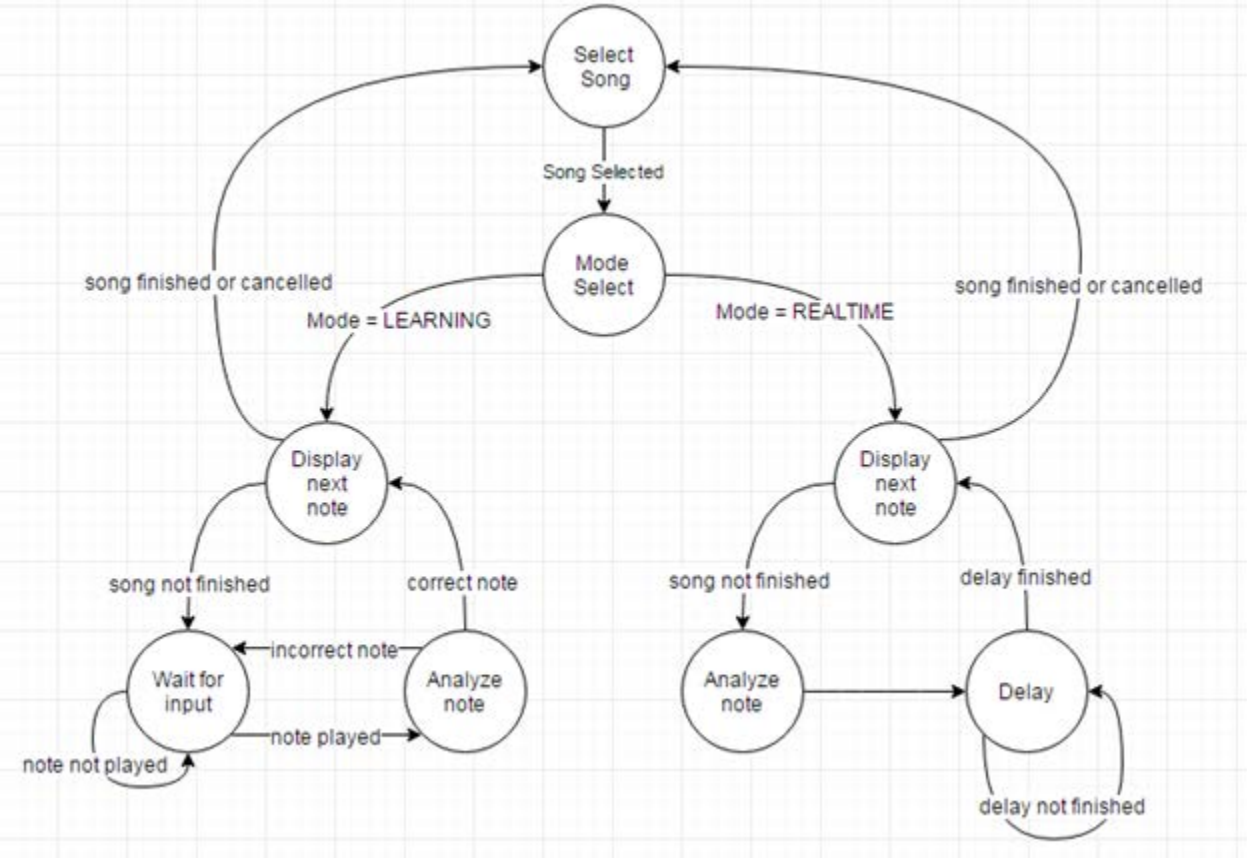


Figure 2.1- Primary Device Program Flow