Software Testing Plan
Version 1.0

Date: March 26, 2021

Team Name: SongBird

Project Sponsor: Paul Flikkema

Team's Faculty Mentor: Andrew Abraham

Team Members: Kevin Imlay, Daniel Mercado, Yasmin Vega-Nuno, Anqi Wang

Table of Contents

1 Introduction

Birds play an essential role in the health and development of many ecosystems around the world. Pest insect populations are regulated by bird predation, dead animals are disposed of by bird scavenging, and many seeds rely on birds for distribution and priming for sprouting. Despite the extreme importance of birds, the scientific community still has numerous questions surrounding the behavior of birds and the ways that they communicate. Questions such as where birds eat and nest, how they move from area to area, and how they communicate with each other remain unanswered. Possibly more importantly, it is not known how these behaviors are changing in response to sound pollution, contact with humans, and climate change.

To solve this problem, we are developing BiVo, a front-end-back-end system for recording bird vocalizations. BiVo will use the EFM32GG12 Thunderboard for main recording and audio analysis on site, which will send audio segments to the desktop application. The desktop application will interface with Matlab for easy data analysis, and allow the user to record a specified number of audio segments, view a spectrogram of a wav file, and save wav files.

During the development of BiVo we will be creating software tests. Software testing is used to find and remove bugs early on in development and help us determine if our product fixes the problem it is trying to solve. This is achieved through unit testing to determine if the smallest units of code function properly, integration testing to determine if those units fit together without issues, and usability testing to address any flaws in the user interface with the system. Testing flows from the smallest tests of units to the largest test of the usability and acceptance testing of the system along with every step of development to address issues as early as possible. We will not address acceptance testing in this document as it will be done at a later date.

Our plan for testing is to perform unit and integration testing on each module and useability testing for the whole product. For unit testing the sensor, we will write our own custom scripts in C and Python since that is the easiest way to verify values and tests on the sensor side. For the front end, we will use the Python “unittest” module in the standard python library and Matlab’s application testing framework. This will help us automate testing on the units of the GUI and the Python components of the desktop application.

We will then need to perform integration testing to verify that data flows correctly between modules. This will be done with use of the Python “unittest” module as well as custom C scripts and some manual testing to supplement the lack of an embedded testing framework for the sensor.

Finally, usability testing will be conducted to help us identify the deficiencies of our user interface and workflow. We have selected twelve people with various levels of experience in computer use to test our product via the Matlab GUI. These people will be observed to determine qualities such as the ease of use and time taken to figure out the user interface, and will be questioned about their thoughts and what can be improved upon.

2 Unit Testing

Unit testing is the first step in the testing and validation process for a software system. In order to make sure all of our units work properly we will implement unit testing. In general, testing should begin on the smallest indivisible unit of the system to make sure all components work correctly on their own. This is done to break down the debugging process into small and manageable sections and to prevent more complex bugs when the units are later integrated.

In order to make our testing easy and replicable, we will be using testing frameworks and automated testing scripts wherever possible. For testing the desktop application, we will make use of Python’s standard unit testing module “unittest” and Matlab’s app testing framework for its App Builder tool. Both of these provide automated testing with mocking and stubs, which help isolate the testing from other units that are called on. For testing the sensor, custom C and Python scripts will be used to automate testing where possible, but some testing will have to be done manually with Simplicity Studio's debugger.

Due to the novelty of this project, most of the units in our system will have to be tested. The exceptions to these tests are the functionalities that are supplied from Matlab such as displaying the spectrogram and the file explorer because we didn't create those, and the units associated with the general communication modules and audio analysis module, because these have been removed from our final product.

Within the Matlab GUI, the units that will be tested are the capturing of numeric user input and storing selected file information. Within the Python component of the application, we will be testing the units of creating/exporting wav files, moving files between directories, the generation of wav file names, establishing a connection with the board, and collecting data from the connection stream.

Within the sensor, the units that will be tested are initializing the serial communication module, sending and receiving within this module, initializing and recording in the microphone module, and initialization and interrupt handling within the LDMA utility.

2.1. Audio Segment Input in the GUI

The number that the user specifies in the number of four second audio segments will need to be tested. The partition equivalence would be any negative integer, zero, and any positive integer. The boundary condition would be the positive integer 3600 (which results in about an hour of audio). There is not a boundary condition for the negative integers because any negative integer no matter how big or small is invalid. Zero is also considered to be invalid because no recording will ensue. No recording should also result from a negative integer. Any positive integer up to 3600 should result in recording occurring.

2.2. Storing Selected File Information in the GUI

All browse buttons need to receive the path to a wav file in order to compute the spectrogram or play its audio. To do this, the file explorer is opened and the GUI awaits a selection to give the path to a function that stores that file's path into a variable that can later be accessed by the spectrogram and play audio functionality. The function that stores the path needs to be tested. Because the function deals with paths, there will not necessarily be equivalence partition or boundary values. Correct values to test are if we receive a char value from the file explorer that contains a wav file. This indicates that the user selected a file and a file which is considered in the correct format. The invalid values to test is if a double (or empty vector) is received which indicates that the user closed the file explorer window without selecting anything, or if a char value was received but it contains a file that is in any file format besides wav.

2.3. Creating/Exporting Wav Files

One of the inputs to the function that creates the wav files is a byte array. The equivalence partition would be a byte array with zero entries or a byte array with 16000(1 second of audio)-64000 entries(4 seconds of audio) where the boundary conditions are 0 entries and 64000 entries. A byte array with 0 inputs is considered to be invalid because a wav file cannot be created from 0 bytes collected. A byte array with 16000 - 64000 entries is valid to create a wav file from.

2.4. Moving Files Between Directories

There will not be any boundary conditions or equivalence partition for this particular unit. This unit concerns itself with whether the created wav files can be moved to the correct path which should result in the wav file being transported to the Audio directory. The only way to test this functionality is to observe the behavior in a PC, Mac, and Linux environment by running that portion of the script.

2.5. Wav File Name Creation

The testing involved in this unit is similar to the one discussed in 2.4. There will not be any equivalence partition or boundary conditions for this unit. Since this unit looks inside the path that contains the Audio folder and file names to generate a unique file name, the only way to test this is to observe if the function generates a duplicate file name that already exists within the Audio folder on the computer. Existing wav file names are invalid. If the wav file name is unique and does not exist in the Audio folder, then it is considered valid. This portion of the script will have to be run on PC, Mac, and Linux environments.

2.6. Establishing a Connection with the Board

There are no explicit inputs for this function however there are inputs from the class itself for this would be the “COM” port that a user would try to connect to which has an equivalence partition from COM0 to COM256 on windows and 0 to 1023 ports on unix systems. While the boundary conditions are COM0 to COM256 on windows and 0 to 1023 ports on unix systems. There is no boundary condition on negative ports because any negative number would be incorrect in this context there should never be a negative port. The boundary condition on the ports for windows is 0-256 and 0-1023 on unix systems. For this I would ensure that there is actually a connection and handle any connection exceptions.

2.7. Collecting Data from the Connection’s Stream

Collecting data would have no explicit inputs for the function. This function returns whenever the buffer reaches the appropriate size or the timeout limit is reached or an exception occurs if there is a connection error or it is trying to read from an invalid connection. This will test for the appropriate buffer size from 0 to 1020 bytes with 1020 being the maximum buffer size that can be returned with pyserial.

2.8. Initializing the Serial Communication Module

For initializing the serial communication module, there aren't any boundary conditions or equivalence partitions. This is because there is no input to the function and there is only one configuration with the hardware that works. Because of this, the unit test will consist of using the debugger tool in Simplicity Studio to check that the hardware registers are set appropriately.

2.9. Sending with the Serial Communication Module

Sending with the serial communication module takes three inputs, the buffer to send from, the number of bytes to send, and a callback function pointer for when the sending is complete. The callback will be tested in another unit test (2.10 below). A single send operation has an upper limit of sending 2048 bytes and a lower limit of 1 byte. The equivalence partition falls between 1 and 2048, with 1 and 2048 being the boundary conditions. Erroneous inputs to be tested would include 0 and negative numbers, and inputs 2049 and above.

2.10. Receiving with the Serial Communication Module

Receiving with the serial communication module takes in three inputs, the buffer to save into, the amount to save into that buffer, and a callback function pointer for when the sending is complete. The callback will be tested in another unit test (2.10 below). A single receive operation has an upper limit of 2048 bytes and a lower limit of 1 byte. The equivalence partition falls between 1 and 2048, with 1 and 2048 being the boundary conditions. Erroneous inputs to be tested would include 0 and negative numbers, and inputs 2049 and above.

2.11. Serial Communication Module Callback Functions

The completion callback function has only one input (a pointer to that function to run), so there are only three cases that can be tested. These cases are a valid function pointer, a null pointer (which is also valid), and a pointer to something that is other than a function as an erroneous input.

2.12. Initializing the Microphone Module

Similar to the serial communication module initialization, there are no inputs to test, so there are no boundary conditions, erroneous inputs, or equivalence partitions to test. Instead, the unit test will consist of using the debugger tool in Simplicity Studio to check that the hardware registers are set appropriately.

2.13. Recording with the Microphone Module

Recording with the microphone module takes in three inputs, the buffer to record into, the size of that buffer, and a callback function pointer. This callback function will be tested in another unit test (2.13 below). A single record operation has an upper limit of 1024 bytes and a lower limit of 2 byte. The equivalence partition falls between 2 and 1024, with 1 and 1024 being the boundary conditions. Erroneous inputs to be tested would include 0 and negative numbers, and inputs 1025 and above. Additionally, recording will be tested using the debugger tool in Simplicity Studio to check that data is being recorded, and Audacity and Excel will be used to verify that the audio recorded is accurate to what is being recorded.

2.14. Microphone Module Callback Functions

The completion callback function has only one input (a pointer to that function to run), so there are only three cases that can be tested. These cases are a valid function pointer, a null pointer (which is also valid), and a pointer to something that is other than a function as an erroneous input.

2.15. Initializing the LDMA Utility

Similar to both the microphone module and serial communication module, there are no inputs to test, so there are no boundary conditions, erroneous inputs, or equivalence partitions to test. Instead, the unit test will consist of using the debugger tool in Simplicity Studio to check that the hardware registers are set appropriately.

2.16. Allocating Channels with the LDMA Utility

Allocating LDMA channels to modules takes in two parameters, the channel to use and a pointer to the function to call to maintain the appropriate module when a transfer is complete. There exist 12 channels for LDMA transfers (0-11), so the equivalence partition is 0-11. The boundary cases are 0 and 11, and erroneous inputs would be any negative numbers and any number 12 or greater. The callback function will be tested in a different unit (2.17 below) test because it is only set in this function, but not used.

2.17. Interrupt Handling with the LDMA Utility

There are no input parameters for the LDMA interrupt handler, so the full functionality must be tested in an integration test. For the functionality that can be tested, the debugger tool will be used to look at variable values and register values to confirm that it is functioning properly.

2.18. LDMA Utility Callback Functions

The completion callback function has only one input (a pointer to that function to run), so there are only three cases that can be tested. These cases are a valid function pointer, a null pointer (which is also valid), and a pointer to something that is other than a function as an erroneous input.

3 Integration Testing

The next step in the testing and validation process, after all unit testing has passed, is integration testing. Integration testing refers to testing the flow of data between the units verifying that there is not a mishandling of data through this transfer of data. This is important because once we have completed unit tests we will want to make sure the transfer of data between modules is bug free. The focus for our integration test will be on the interaction points between the individual modules created.

After revisions of requirements with our client, we are now left with the following modules in our current implementation of BiVo to test: serial communication and microphones/recording, on the sensor side, and serial communication, data management, and data visualization on the desktop application side.

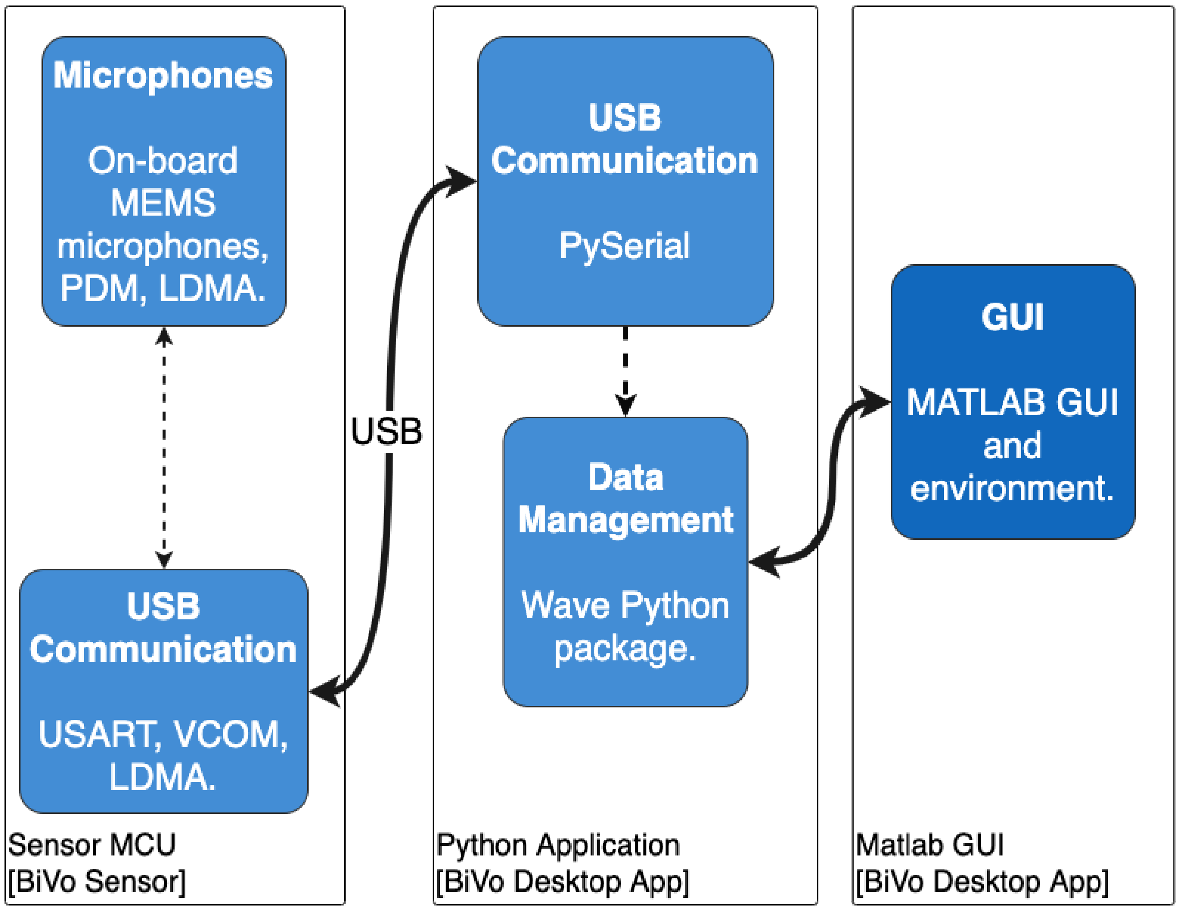

Figure 1. Interactions between the modules in the current implementation of our product. Note: data visualization is inside the GUI.

To begin with, the general data flow starts at the microphone module. The Thunderboard's microphones will record audio as audio samples. Next, the USB communication module on the sensor side will need to send these audio samples to the serial communication module on the desktop application side. The serial communication module's job on the desktop side is to send the audio samples as byte data to the data management module to create playable wav files for the Matlab GUI. Data Visualization lives entirely in Matlab and depends on the wav files generated in data management to either generate a spectrogram, or to play them.

3.1. Serial Communication Between the Sensor and Application

This integration point we will verify the data on both the Sensor side and User Interface by running an echo test to confirm transmission of data is correct in both directions. With this we will ensure that the entirety of data is being transmitted correctly without corruption of loss. This can be written with a custom Python script or using the Python "unittest" module.

3.2. Microphone to Serial Communication Desktop Side

Due to the use of hardware to move data in the sensor, the integration of the microphone module and sensor side serial communication module cannot be observed directly. Instead, the integration will have to be tested by sending the microphone data to the desktop application's serial communication module. This will need to be performed after test 3.1 is tested to eliminate the possibility of bugs with the communication. This will be tested using a custom Python script along with Audacity to confirm the audio recorded and sent is accurate to what was being recorded.

3.3. Serial Communication Desktop Side to Data Management

This integration point will verify that data is correctly sent from the data stream from the board to be compiled into a wav file. To test this we will make use of the Python "unittest" module and Audacity to feed in a byte stream of data from a known audio file and then compare it to the new audio file produced. This ensures that data is not being corrupted and is being saved appropriately into the wav format.

3.4. Between Matlab and Communication Desktop Side

This integration point will verify that Matlab can appropriately call the Python side of the application to collect audio segments, and that the Python side of the application can effectively update Matlab on its progress. Testing for this will involve running the streaming mode from the GUI to get audio segments and observing that the GUI is updated appropriately. This will need to be done manually due to the lack of a testing framework that can observe both the Matlab and Python sides simultaneously.

4 Useability Testing

Usability testing is the second to last step in the testing and validation process for a software system, the last being acceptance testing. This will be done after integration and unit testing so that we can confirm that the product works with little to no bugs. This type of testing is used to evaluate the functionality and ease of use of our product by having end-users go through the workflow of our product by completing tasks. By doing so, we will be able to see what can be improved upon and if the product is usable in its current state.

There are a limited number of actions the user can take to use the BiVo system at this point. Those include plugging the board, opening the GUI, running streaming mode, displaying a spectrogram of an audio segment, and listening to an audio segment. Additionally, the process of flashing the board will be skipped as it requires extensive knowledge and experience in software IDEs, toolchains, Git, and troubleshooting, which are all known to be outside the experience of our test users. The installation and configuring of Matlab will also be skipped because Matlab requires a paid licence that our test users will not have access to. These are known deficiencies of our testing user group, but would hopefully not be deficiencies of our target demographic of scientist end-users.

Due to COVID-19, the amount of users we can test with, as well as the demographic of users we can recruit to test our product, is severely limited. This is because many people are reluctant to agree to meet in person. This creates a situation where we have less users to test BiVo and no users in the target end-user demographic.

In our usability testing plan, we have 12 test users selected on a volunteer basis by contacting roommates, friends, and families in close contact with ourselves. They will perform the 3 main tasks available: record a number of audio segments, view the spectrogram of a wav file produced from recording, and play a wav file produced from recording.

We will conduct our testing in an in-person and moderated fashion. We have chosen to use in-person testing because we need to also provide the hardware and software to run the system, and we have chosen to moderate to help guide the flow of the test. For the three main tasks, we will observe the following qualitative and quantitative attributes:

Recording:

- How confused the user is figuring out how to record audio segments.
- How many questions the user asks.
- How much the user struggles before completing the recording segments task.
- How many clicks does it take the user to record an audio segment
- How long it takes the user to record an audio segment

Viewing Spectrogram:

- How confused the user is in finding where to view a spectrogram.
- How many questions the user asks.
- How many clicks it takes the user to successfully view a spectrogram.
- How much the user struggles before completing the viewing the spectrogram task.
- How long it takes for the user to select a recorded wav file.
- How long it takes for the user to successfully view a spectrogram.

Playing Audio:

- How confused the user is in figuring out how to play an audio file.
- How many questions the user asks.
- How many clicks it takes the user to play an audio file.
- How much the user struggles before completing the playing audio task.
- How long it take the user to play an audio segment
- How long it takes a user to select the file.

Information will be recorded by observing the test users perform these actions. Quantitative attributes of performance like mouse movements, mouse clicks, and well as time to perform can be directly observed. Quantitative attributes can be recorded by noting comments made by the users during the test as well as interviewing the users after the tests complete.

Analysis of these results will look at the average and median of each of the quantitative results, and the qualitative results will be compiled into a list of comparing positive results to negative results. From these we will compare our number of clicks and time taken against the requirements document to determine if we need to simplify the GUI layout, and we will analyze the positive and negative results to find where we could make the process more intuitive as a whole.

5 Conclusion

As mentioned in the beginning, birds play a vital role in the health and development of many ecosystems by regulating pest populating, scavenging, and distributing seeds. Despite this, we still have many questions surrounding their behavior. Our solution is BiVo, a front-end-back-end system for recording, analyzing, and sending bird vocalizations to the PC for further analysis in Matlab. Our testing plan is composed of three phases: unit testing, integration testing, and useability testing. To implement our plan, we will be using Python's "unittest", Matlab's app testing framework, custom C and Python scripts, and Simplicity Studio's debugger. In the unit testing phase, we will observe the output resulting from valid and invalid inputs in our units inside of our modules. In the integration testing phase, we will be testing the portions of our code that link modules together through data transfers. The following are the core links we will be testing: serial communication between the sensor and application, microphone to serial communication (desktop side), serial communication (desktop side) to data management, and between matlab and communication (desktop side). In the useability testing phase, we will be gathering a total of 12 end-users to evaluate the functionality and ease of use of our product by performing three tasks: record 2 audio segments, generate a spectrogram, and play a wav file. Additionally, we will be recording and analyzing relevant data such as the time and number of clicks it took to finish each task and the difficulties the user had. Our three phase testing plan will ensure that we deliver a working and easy to use product with minimal bugs.

Despite the hurdles and roadblocks we have faced on both the front-end and back-end, we have managed to create a functioning product. To date, we have incorporated the ability for the Thunderboard to record audio and stream that audio to the PC. On the front-end, we have incorporated the ability to save the streamed audio as wav files, view the spectrograms of the wav files, and to play recorded wav files. Next on our agenda is to successfully implement audio analysis, increase the audio quality coming from the Thunderboard's microphones, refine the front-end communication side to programmatically find the COM port the Thunderboard is connected to, and for the GUI to programmatically find where the Python scripts are located without having to hard code the paths. Despite the fact that we still have quite a bit to do, we have implemented the product's core functionality and are confident that we will be able to implement the remaining functionality and refinements that are on our agenda.

Great Blue Heron at Kaibab Lake, Williams, AZ. Imlay, K. 2017.