System Verification and Validation Plan for Audio360

Team #6, Six Sense
Omar Alam
Sathurshan Arulmohan
Nirmal Chaudhari
Kalp Shah
Jay Sharma

October 26, 2025

Revision History

Date	Version	Notes
Date 1	1.0	Notes
Date 2	1.1	Notes

[The intention of the VnV plan is to increase confidence in the software. However, this does not mean listing every verification and validation technique that has ever been devised. The VnV plan should also be a **feasible** plan. Execution of the plan should be possible with the time and team available. If the full plan cannot be completed during the time available, it can either be modified to "fake it", or a better solution is to add a section describing what work has been completed and what work is still planned for the future. —SS]

[The VnV plan is typically started after the requirements stage, but before the design stage. This means that the sections related to unit testing cannot initially be completed. The sections will be filled in after the design stage is complete. the final version of the VnV plan should have all sections filled in. —SS]

Contents

1 Symbols, Abbreviations, and Acronyms **iv**

2 General Information **1**
- 2.1 Summary . . . 1
- 2.2 Objectives . . . 1
- 2.3 Challenge Level and Extras . . . 1
- 2.4 Relevant Documentation . . . 2

3 Plan **2**
- 3.1 Verification and Validation Team . . . 2
- 3.2 SRS Verification . . . 2
- 3.3 Design Verification . . . 3
- 3.4 Verification and Validation Plan Verification . . . 3
- 3.5 Implementation Verification . . . 3
- 3.6 Automated Testing and Verification Tools . . . 3
- 3.7 Software Validation . . . 4

4 System Tests **4**
- 4.1 Tests for Functional Requirements . . . 4
  - 4.1.1 Audio Filtering Tests . . . 5
  - 4.1.2 Visualization Controller Tests . . . 9
- 4.2 Tests for Nonfunctional Requirements . . . 11
  - 4.2.1 Audio Filtering . . . 11
  - 4.2.2 Visualization Controller . . . 13
- 4.3 Traceability Between Test Cases and Requirements . . . 15

5 Unit Test Description **15**
- 5.1 Unit Testing Scope . . . 16
- 5.2 Tests for Functional Requirements . . . 16
  - 5.2.1 Module 1 . . . 16
  - 5.2.2 Module 2 . . . 17
- 5.3 Tests for Nonfunctional Requirements . . . 17
  - 5.3.1 Module ? . . . 17
  - 5.3.2 Module ? . . . 18
- 5.4 Traceability Between Test Cases and Modules . . . 18

6 Appendix **19**
6.1 Symbolic Parameters 19
6.2 Usability Survey Questions? 19

List of Tables

1 Functional Requirements and Corresponding Test Sections . . 15
2 Non-Functional Requirements and Corresponding Test Sections 15
[Remove this section if it isn't needed —SS]

List of Figures

[Remove this section if it isn't needed —SS]

1 Symbols, Abbreviations, and Acronyms

symbol	description
T	Test

[symbols, abbreviations, or acronyms — you can simply reference the SRS (Sathurshan et al., 2025) tables, if appropriate —SS]

[Remove this section if it isn't needed —SS]

This document ... [provide an introductory blurb and roadmap of the Verification and Validation plan —SS]

2 General Information

2.1 Summary

[Say what software is being tested. Give its name and a brief overview of its general functions. —SS]

2.2 Objectives

[State what is intended to be accomplished. The objective will be around the qualities that are most important for your project. You might have something like: "build confidence in the software correctness," "demonstrate adequate usability." etc. You won't list all of the qualities, just those that are most important. —SS]

[You should also list the objectives that are out of scope. You don't have the resources to do everything, so what will you be leaving out. For instance, if you are not going to verify the quality of usability, state this. It is also worthwhile to justify why the objectives are left out. —SS]

[The objectives are important because they highlight that you are aware of limitations in your resources for verification and validation. You can't do everything, so what are you going to prioritize? As an example, if your system depends on an external library, you can explicitly state that you will assume that external library has already been verified by its implementation team. —SS]

2.3 Challenge Level and Extras

[State the challenge level (advanced, general, basic) for your project. Your challenge level should exactly match what is included in your problem statement. This should be the challenge level agreed on between you and the course instructor. You can use a pull request to update your challenge level (in TeamComposition.csv or Repos.csv) if your plan changes as a result of the VnV planning exercise. —SS]

[Summarize the extras (if any) that were tackled by this project. Extras can include usability testing, code walkthroughs, user documentation, formal proof, GenderMag personas, Design Thinking, etc. Extras should have already been approved by the course instructor as included in your problem statement. You can use a pull request to update your extras (in TeamComposition.csv or Repos.csv) if your plan changes as a result of the VnV planning exercise. —SS]

2.4 Relevant Documentation

[Reference relevant documentation. This will definitely include your SRS and your other project documents (design documents, like MG, MIS, etc). You can include these even before they are written, since by the time the project is done, they will be written. You can create BibTeX entries for your documents and within those entries include a hyperlink to the documents. —SS]

Sathurshan et al. (2025)

[Don't just list the other documents. You should explain why they are relevant and how they relate to your VnV efforts. —SS]

3 Plan

[Introduce this section. You can provide a roadmap of the sections to come. —SS]

3.1 Verification and Validation Team

[Your teammates. Maybe your supervisor. You should do more than list names. You should say what each person's role is for the project's verification. A table is a good way to summarize this information. —SS]

3.2 SRS Verification

[List any approaches you intend to use for SRS verification. This may include ad hoc feedback from reviewers, like your classmates (like your primary reviewer), or you may plan for something more rigorous/systematic. —SS]

[If you have a supervisor for the project, you shouldn't just say they will read over the SRS. You should explain your structured approach to the review. Will you have a meeting? What will you present? What questions will you ask? Will you give them instructions for a task-based inspection? Will you use your issue tracker? —SS]

[Maybe create an SRS checklist? —SS]

3.3 Design Verification

[Plans for design verification —SS]

[The review will include reviews by your classmates —SS]

[Create a checklists? —SS]

3.4 Verification and Validation Plan Verification

[The verification and validation plan is an artifact that should also be verified. Techniques for this include review and mutation testing. —SS]

[The review will include reviews by your classmates —SS]

[Create a checklists? —SS]

3.5 Implementation Verification

[You should at least point to the tests listed in this document and the unit testing plan. —SS]

[In this section you would also give any details of any plans for static verification of the implementation. Potential techniques include code walkthroughs, code inspection, static analyzers, etc. —SS]

[The final class presentation in CAS 741 could be used as a code walkthrough. There is also a possibility of using the final presentation (in CAS741) for a partial usability survey. —SS]

3.6 Automated Testing and Verification Tools

[What tools are you using for automated testing. Likely a unit testing framework and maybe a profiling tool, like ValGrind. Other possible tools include a static analyzer, make, continuous integration tools, test coverage tools, etc. Explain your plans for summarizing code coverage metrics. Linters are another important class of tools. For the programming language you select,

you should look at the available linters. There may also be tools that verify that coding standards have been respected, like flake9 for Python. —SS]

[If you have already done this in the development plan, you can point to that document. —SS]

[The details of this section will likely evolve as you get closer to the implementation. —SS]

3.7 Software Validation

[If there is any external data that can be used for validation, you should point to it here. If there are no plans for validation, you should state that here. —SS]

[You might want to use review sessions with the stakeholder to check that the requirements document captures the right requirements. Maybe task based inspection? —SS]

[For those capstone teams with an external supervisor, the Rev 0 demo should be used as an opportunity to validate the requirements. You should plan on demonstrating your project to your supervisor shortly after the scheduled Rev 0 demo. The feedback from your supervisor will be very useful for improving your project. —SS]

[For teams without an external supervisor, user testing can serve the same purpose as a Rev 0 demo for the supervisor. —SS]

[This section might reference back to the SRS verification section. —SS]

4 System Tests

This section outlines the tests for verifying and validating the functional and nonfunctional requirements outlined in the SRS (Sathurshan et al., 2025). When done correctly this ensures the system meets the user expectations and performs reliably.

4.1 Tests for Functional Requirements

The sections below outline the tests that will be used to verify the functional requirements in section S.2 of the SRS. Each subsection will focus on how the functional requirements for a specific component will be verified through testing. These components include the Embedded Firmware, Driver Layer,

Audio Filtering, Audio360 Engine, Frequency Analysis, Visualization Controller, Microphone, Output Display and Microcontroller.

4.1.1 Audio Filtering Tests

This section covers the tests for ensuring the system processes audio into a form that can be analyzed by internal components of the system. Each test is associated with a functional requirement defined under section 3.2.3 of the SRS. As such, each test will verify whether the system meets the associated functional requirement.

1. **test-FR-3.1** Converting time-domain audio signals to frequency-domain

   Control: Automatic

   Initial State: The audio filtering module is initialized and ready to process audio input retrieved from an audio file.

   Input: A 3 second audio clip represented in an audio file containing pre-recorded audio data in the time domain sampled at 16 kHz. The audio clip contains 3 sine waves at low (100 Hz), mid (1 kHz), and high (8 kHz) frequency ranges. No filtering or frequency transformation have been applied to the audio data initially.

   Output: The audio filtering module accepts the file with no errors. The resulting frequency domain representation should display 3 spectral peaks at approximately 100 Hz, 1 kHz, and 8 kHz, corresponding to the sine waves.

   Test Case Derivation: The Fourier Transform converts time-domain signals into frequency domain by independently extracting the frequency of various waves in the signals and plotting the peaks at those frequencies after the transformation. In the original audio clip, there are 3 sine waves at 100 Hz, 1 kHz, and 8 kHz. After applying the Fourier Transform, the resulting frequency domain representation should display peaks at those frequencies.

   How test will be performed: The test file will be uploaded as an artifact in the automated testing framework. This test will trigger when a commit is made to any branch in the repository. The audio filtering module will return the frequency domain representation automatically

on the input of the audio file. The frequency-domain output will be inspected to verify the presence of peaks at 100 Hz, 1 kHz and 8 kHz. The test passes if all 3 peaks are present with no unexpected frequencies showing up.

2. **test-FR-3.2** Normalize amplitude of signals

   Control: Automatic

   Initial State: The audio filtering module is initialized and ready to process audio input retrieved from a audio file.

   Input: A 2 second digital audio signal sampled at 16 kHz that alternates between a low-amplitude sine wave and a high-amplitude sine wave with the same frequency. These sine waves will be decimal multiples of a defined max amplitude value. Where the low-sine wave will be 0.2 * max amplitude, and the high sine wave will be 0.8 * max amplitude.

   Output: A normalized output signal that still has both the low amplitude and high amplitude sine waves, but both waves have been scaled to a consistent target amplitude, having a maximum absolute value of 1.0. Note, the frequency of the sine wave should remain unchanged.

   Test Case Derivation: Amplitude normalization scales the amplitude of a signal so its maximum ampltiude is between 0 and 1. If one section is quiet (0.2 * max), and another section is louder (0.8 * max), normalization should scale both sections so their peak amplitudes are between the range 0 and 1.

   How test will be performed:

   The test file will be uploaded as an artifact in the automated testing framework. This test will trigger when a commit is made to any branch in the repository. The audio filtering module will return normalized time-domain signal automatically on the input of the audio file. The normalized time domain output will be inspected to verify the amplitude across both sections of the file are the same now.

3. **test-FR-3.3** Reduced spectral leakage

   Control: Automatic

Initial State: The audio filtering module is initialized and ready to process audio input retrieved from an audio file.

Input: A 1 second sine wave at an arbitary frequency sampled at 16 kHz, whose duration does not contain an integer number of cycle (the cycle is cut off before 1 period is complete). This intentionally causes spectral leakage. This will be passed in with a parameter of whether to apply a windowing function or not. In one case, a window function will be passed in, in the other no function will be passed in.

Output: The windowed output audio should have reduced spectral leakage. This is represented by a sharper and more defined peak at the sine wave's frequency, with reduced side-lobes in the frequency spectrum compared to the output of the non-windowed case.

Test Case Derivation: Spectral leakage occurs when a signal is truncated without windowing, causing discontinuities at the edges of the truncated signal. Applying a windowing function tapers the edges of the signal, reducing the discontinuities, and confining the energy to the main frequency band, preventing leakage into other frequencies from occuring. As such, in the windowed case, the frequency spectrum should show a sharper peak at the sine wave's frequency, with redcued side-lobes compared to the non-windowed case. The leakage will be measured by first computing the peak amplitude K_{peak}, then applying the leakage function. Where M represents the mainlobe half-width in bins, based on the windowing function used.

$$\text{Leakage} = 1 - \frac{\sum_{k=k_{\text{peak}}-M}^{k_{\text{peak}}+M} |X[k]|^2}{\sum_{k} |X[k]|^2}$$

How test will be performed: The test file will be uploaded as an artifact in the automated testing framework. This test will trigger when a commit is made to any branch in the repository. The audio filtering module will return 2 frequency-domain spectrums. One spectrum will be generated without windowing, and the other will be applied with a windowing function. For each spectrum, the amplitude of the mainlobe will be compared with the largest side-lobe amplitude. The test

passes if the side-lobe in the filtered case is lower than the unfiltered case, which indicates reduced spectral leakage.

4. **test-FR-3.4** Hardware acceleration

   Control: Manual

   Initial State: The audio filtering module is deployed on the microcontroller. A local computer without hardware acceleration is available to the team to test the audio filtering component on.

   Input: A 10 second digital audio signal sampled at 16 kHz containing waves with mixed frequencies and amplitudes. The same input will be processed once with the microcontroller, and once on a local development machine without hardware acceleration. Each test will be timed to measure the processing speed.

   Output: Both processing modes should produce equivalent spectrograms for the given audio input. This means for each frequency in the spectrogram, the amplitude defined in the hardware-accelerated mode should match the amplitude in the non-accelerated mode within a defined tolerance of 0.1%. The hardware accelerated run should complete in less time than the non-accelerated run.

   Test Case Derivation: Hardware acceleration uses specialized processing uits to perform expensive operations, like FFT or convolutions more efficiently than general-purpose. Verifying the reduced runtime and equivalent outputs confirms the module deployed on the hardware is functioning correctly.

   How test will be performed: Manually running one confirguration on the microcontroller, and another on the local computer. Execution time will be measured with performance logs. Test function will be written to measure the numerical equivalence of both outputs after processing is completed. Logs will be manully inspected to verify the response time of the hardware accelerated mode is less than the non-accelerated mode.

5. **test-FR-3.5** Flagging anomalies

   Control: Automatic

Initial State: The audio filtering module is initialized and ready to process audio input retrieved from an audio file.

Input: Three separate audio clips represented in audio files. One will have a 1 second sine wave with an amplitude that exceeds 1.0. Since amplitudes above 1.0 will become clipped. Another clip will have a 1 second sine wave, that is replaced by zeros halfway. This will test the lost signal case. The last clip will just have 2 secnds of zero amplitude, measuring the silence case.

Output: For each test case, the component should output the correct anomaly flag. In this case, for the first audio clip, it should output a clipping flag. For the second clip, it should output a lost signal flag. For the last clip, it should output a silence flag.

Test Case Derivation: Clipping occurs when the amplitude of a signal exceeds the maximum representable value (-1.0 to 1.0 for normalized audio). As such, for sine wave with an amplitude above 1.0, clipping will occur. A lost signal is detected when a section of the audio suddenly dropped to zero amplitude, which is the case in the second clip. Silence is detected when the entire audio clip has zero amplitude, which is the case in the last clip.

How test will be performed: Each test file will be uploaded as an artifact in the automated testing framework. There will be a test case for each test file, measuring each of the anomalies mentioned above. THe test cases will trigger when a commit is made to any branch in the repository. The audio filtering module will return 1 output for each test case. Test will be verified by asserting whether the correct anomly is displayed for each audio file in each test case.

4.1.2 Visualization Controller Tests

This section covers the tests for ensuring the correct output is being created and sent from the visualization controller to the output display. Each test is associated with a functional requirement defined under section 3.2.6 of the SRS. As such, each test will verify whether the system meets the associated functional requirement.

1. **test-FR-6.1** Notify direction of audio source

Control: Manual

Initial State: The Visualization Controller module is deployed on the microcontroller and initialized. Drivers for output display are installed in microcontroller, and the microcontroller is connected to the output display.

Input: A mock audio source direction input, represented as the object taken by the Visualization Controller module. The object will include an angle parameter in degrees (0 to 360°), indicating the direction of the audio source relative to the user. This can be an arbitary angle, such as 0°, 90°, 180°, 270°.

Output: Corresponding visual indicator appears on the output display pointing in the same direction as the input angle. The visualization appears within 1 second of inputting the direction (VC-3.2)

Test Case Derivation: When the audio system detects an incoming sound and reports its direction, the visualization controller must translate that information into a user-facing cue so that it may displayed on the output display. In this case, by sending an object that outlines the direction of audio, that direction must be formatted by the Visualization Controller so that it can be rendered on the output display. This confirms that signals are being correctly translated.

How test will be performed: Simulate a directional events by mocking the Visualization Controller's input object with directions at 0°, 90°, 180°, 270°. Capture the output display output by visually seeing if the correct direction is visualized. The test passes if all simulated directions match the expected visual outputs and response time thresholds (read using microcontroller logs) are met.

2. **test-FR-6.2** Notify direction or classification failure

Control: Manual

Initial State: The Visualization Controller module is deployed on the microcontroller and initialized. Drivers for output display are installed in microcontroller, and the microcontroller is connected to the output display.

Input: 2 mock audio source direction input, represented as the object taken by the Visualization Controller module. The first object's

metadata will include a failure flag indicating that the direction of the audio source could not be determined. The second object's metadata will include a failure flag indicating that the classification of the audio source could not be determined.

Output: For the first input object, a visual indicator appears on the output display signifying that the direction of an audio source could not be determined. The second input object should produce a different visual indicator on the output display signifying that the classification of the audio source could not be determined.

Test Case Derivation: When the audio system fails to determine either the direction or classification, it will report that failure in the input object to the Visualization Controller. The Visualization Controller must then translate that failure information into a user-facing cue so that it may be displayed on the output display. This confirms that errors are correctly being processed and presented to the user.

How test will be performed: Simulate failure events by mocking the Visualization Controller's input object with 2 failure flags, one for direction failure, and one for classification in the object metadata. Capture the output display output by visually seeing if the correct failure indicators are visualized on the output display. The test passes if all simulated failure events match the expected visual outputs.

4.2 Tests for Nonfunctional Requirements

This section covers system tests for the non-functional requirements (NFR) listed under section S.2 of the SRS. Each subsection will be focused on the NFR for a specific component will be verified through testing.

4.2.1 Audio Filtering

1. **test-NFR3.1** Accurate frequency-domain translation

   Type: Non-Functional, Dynamic, Automatic

   Initial State: The audio filtering module is initialized and ready to process audio input retrieved from an audio file. Reference implementation for true frequency-domain representation is available for comparison.

Input/Condition: A 1-second sine wave with arbitrary frequency sampled at 16 kHz. Additional composite signals (white noise segments) may be used for robustness testing.

Output/Result: The computed frequnecy-domain representation from the component should differ from the true spectrum by less than 10% error across all frequency bins.

How test will be performed: Upload the audio file and high-precision FFT reference file to the automated testing framework. Confgiure the test to run every time a commit is made to Git. When a commit is made, the test suite will feed the audio file into the audio filtering component. After retrieving the frequency-domain output, calculate the mean relative error between component's output and the reference spectrum using the following formula across all bins. If the mean is less than 10%, the test passes.

$$\text{Error} = \frac{|A_{\text{component}} - A_{\text{true}}|}{A_{\text{true}}} \times 100\%, \quad \forall A \in \text{Spectrum}$$

2. **test-NFR3.2** Handle different input signal sizes

Type: Non-Functional, Dynamic, Manual

Initial State: The audio filtering component is deployed on the microcontroller, and ready to process audio input retrieved from an audio file. Logging has been implemented on the microcontroller to capture time taken for processing.

Input/Condition: Digital audio signals of varying sizes: 512, 1024, 2048 and 4096 frames, all sampled at 16 kHz. Each input contains an arbitary test signal (sine wave with arbitrary frequency).

Output/Result: For each input size, the Audio Filtering component should process all frames without exceeding time constraints defined in NFR1.2.

How test will be performed: Manually upload each audio file to the microcontroller and trigger processing. Execution time will be measured using microcontroller logs. After processing is complete, logs will be manually inspected to verify the processing time for each input size meets the time constraints defined in the SRS.

3. **test-NFR3.3** Accuracy of FFT calculation exceeds 90%

   Type: Non-Functional, Dynamic, Manual

   Initial State: The audio filtering component is deployed on the microcontroller, and ready to process continous audio retrieved from the environment using attatched microphones. Mechanism to output spectrogram data from microcontroller is available for future analysis.

   Input/Condition: 60 second continous audio from the environment sampled at 16 kHz. The audio should contain a mix of frequencies and amplitudes to simulate real-world conditions. This same audio will be processed simulatenously by a high-precision FFT reference implementation on a seperate laptop.

   Output/Result: The spectrogram output from the microcontroller should match the accuracy of the reference implementation with at most 10% relative error across all frequency bins. The following formula can be used to calculate the relative error is shown below.

$$\text{Error} = \frac{|A_{\text{component}} - A_{\text{true}}|}{A_{\text{true}}} \times 100\%, \quad \forall A \in \text{Spectrum}$$

   How test will be performed: Manually record 60 seconds of audio from the environment using microphones attatched to microcontroller. The same audio will be recorded on a seperate laptop for reference processing. After recording, both the microcontroller and laptop will output their respective spectrograms. The spectrograms will be compared by calculating the mean relative error across all frequency bins using the formula above. If the mean error is less than 10%, the test passes

4.2.2 Visualization Controller

1. **test-NFR6.1** Display safety critical information first

   Type: Non-Functional, Dynamic, Manual

   Initial State: The Visualization Controller module is deployed on the microcontroller and initialized. Drivers for output display are installed

in microcontroller, and the microcontroller is connected to the output display.

Input/Condition: 3 mock audio sources, represented as the object taken by the Visualization Controller module. These sources will be sent simulatenously to the module. The object meta will have a parameter that outlines the priority of the audio sourcs. The first object will have the highest priority, the second object will have medium priority and the third object will have the lowest priority.

Output/Result: The output display should only visualize the highest priority audio source first. So in this case, the direction of the first object should be visualized on the output display, and the rest should be ignored.

How test will be performed: Simulate multiple audio sources by mocking the Visualization Controller's input objects with different priority levels. Capture the output display output by visually seeing if only the highest priority direction is visualized on the output display. The test passes if the highest priority direction is the only one visualized.

2. **test-NFR6.2** Present information in a non-intrusive manner

   Type: Non-Functional, Dynamic, Manual

   Initial State: The Visualization Controller module is deployed on the microcontroller and initialized. Drivers for output display are installed in microcontroller, and the microcontroller is connected to the output display.

   Input/Condition: A series of mock audio source direction inputs, represented as the object taken by the Visualization Controller module. The object will include an angle parameter in degrees (0 to 360°), indicating the direction of the audio source relative to the user. These can be an arbitary angles.

   Output/Result: Stakeholders verifies the non-obtrusive nature of the visualizations on the output display. The stakeholder should report that the visualizations do not obstruct their view or cause discomfort during typical usage scenarios.

How test will be performed: Conduct a controlled usability session with at least 5 stakeholders. Record quantiative feedback from stakeholders, each rating the non-obtrusiveness on a scale of 1 to 5 (1 being very obtrusive, 5 being very non-obtrusive). The test passes if the average rating across all stakeholders is at least 4.

4.3 Traceability Between Test Cases and Requirements

Table 1: Functional Requirements and Corresponding Test Sections

Test Section	**Supported Requirement(s)**
Audio Filtering	FR-3.1, FR-3.2, FR-3.3, FR-3.4
Visualization Controller	FR-6.1, FR-6.2

Table 2: Non-Functional Requirements and Corresponding Test Sections

Test Section	**Supported Requirement(s)**
Audio Filtering	NFR-3.1, NFR-3.2, NFR-3.3
Visualization Controller	NFR-6.1, NFR-6.2

5 Unit Test Description

[This section should not be filled in until after the MIS (detailed design document) has been completed. —SS]

[Reference your MIS (detailed design document) and explain your overall philosophy for test case selection. —SS]

[To save space and time, it may be an option to provide less detail in this section. For the unit tests you can potentially layout your testing strategy here. That is, you can explain how tests will be selected for each module. For instance, your test building approach could be test cases for each access program, including one test for normal behaviour and as many tests as needed for edge cases. Rather than create the details of the input and output here, you could point to the unit testing code. For this to work, you code needs to be well-documented, with meaningful names for all of the tests. —SS]

5.1 Unit Testing Scope

[What modules are outside of the scope. If there are modules that are developed by someone else, then you would say here if you aren't planning on verifying them. There may also be modules that are part of your software, but have a lower priority for verification than others. If this is the case, explain your rationale for the ranking of module importance. —SS]

5.2 Tests for Functional Requirements

[Most of the verification will be through automated unit testing. If appropriate specific modules can be verified by a non-testing based technique. That can also be documented in this section. —SS]

5.2.1 Module 1

[Include a blurb here to explain why the subsections below cover the module. References to the MIS would be good. You will want tests from a black box perspective and from a white box perspective. Explain to the reader how the tests were selected. —SS]

1. test-id1

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   Initial State:

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

2. test-id2

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

Initial State:

Input:

Output: [The expected result for the given inputs —SS]

Test Case Derivation: [Justify the expected value given in the Output field —SS]

How test will be performed:

3. ...

5.2.2 Module 2

...

5.3 Tests for Nonfunctional Requirements

[If there is a module that needs to be independently assessed for performance, those test cases can go here. In some projects, planning for nonfunctional tests of units will not be that relevant. —SS]

[These tests may involve collecting performance data from previously mentioned functional tests. —SS]

5.3.1 Module ?

1. test-id1

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   Initial State:

   Input/Condition:

   Output/Result:

   How test will be performed:

2. test-id2

   Type: Functional, Dynamic, Manual, Static etc.

Initial State:

Input:

Output:

How test will be performed:

5.3.2 Module ?

...

5.4 Traceability Between Test Cases and Modules

[Provide evidence that all of the modules have been considered. —SS]

References

Sathurshan, Omar, Kalp, Jay, and Nirmal. System requirements specification. https://github.com/Team6-SixSense/audio360/blob/main/docs/SRS/SRS.pdf, 2025.

6 Appendix

This is where you can place additional information.

6.1 Symbolic Parameters

The definition of the test cases will call for SYMBOLIC_CONSTANTS. Their values are defined in this section for easy maintenance.

6.2 Usability Survey Questions?

[This is a section that would be appropriate for some projects. —SS]

Appendix — Reflection

1. What went well while writing this deliverable?

   Sathurshan: The team had a good understanding of the system as in the areas of the system that have the most critical risk to the project and functionality to the product. As a result, it helped the team know whats tests should be prioritized.

   Nirmal: Having a clear understanding of the project requirements from the SRS made it easier to derive test cases for various components. Furthermore, after working on the POC implementation, I think I had a good understanding of what artifacts can and will be used to test various components. For example, uploading pre-existing test files to automate testing in our pipeline.

2. What pain points did you experience during this deliverable, and how did you resolve them?

   Sathurshan: The team has packed with midterms and assignments from other courses which made working on this deliverable and capstone difficult. There wasn't a good resolution other than getting the team to work on sections of the deliverable when possible.

   Nirmal: Trying to prioritize working on this deliverable with other commitments was very difficult. Especially since this deliverable was smaller compared to the SRS fr example, it made it difficult to push myself to work this in advance, since I thought other things from other courses were more pressing at the time, and that I can probably do this closer to the deadline.

3. What knowledge and skills will the team collectively need to acquire to successfully complete the verification and validation of your project? Examples of possible knowledge and skills include dynamic testing knowledge, static testing knowledge, specific tool usage, Valgrind etc. You should look to identify at least one item for each team member.

   Team respose: The following are the knowledge and skills to perform verification and validation of the project:

   (a) gtest: the main testing tool for writing unit test for source code.

(b) Hardware debugging: There aren't many methods to debug on a microcontroller. Thus we need someone to investigate on how to debug our software on a microcontroller. If not possible, what other ways we can debug our software without the hardware.

(c) Integration testing: Testing the integration of the software on the hardware to verify it has been done correctly.

(d) Validating the product with the user. It is not intuitive at the moment on how we will know that the product addresses the user's problem effectively.

(e) Design verification requires an expert to ensure that the team's initial design is correct to minimize technical debt since there is not a lot of time left in this project.

4. For each of the knowledge areas and skills identified in the previous question, what are at least two approaches to acquiring the knowledge or mastering the skill? Of the identified approaches, which will each team member pursue, and why did they make this choice?

(a) gtest:

(b) Hardware debugging:

(c) Integration testing: Sathurshan will be pursuing this as he has experience of integration testing. He has already acquired partial skills from industry and will acquire more by looking at how public GitHub projects that uses hardware performed integration testing.

(d) Validating the product with the user:

(e) Design verification: Nirmal will be pursuing this since he has experience with design verification from previous internships and research background. He will be looking at best practices for design, using design principles and architecture styles as references to verify whether the correct one has been applied.