System Verification and Validation Plan for Audio360

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

October 28, 2025

Revision History

Date	Version	Notes
2025-10-27	1.0	Initial write-up

Contents

List of Tables

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 SRS tables, if appropriate —SS]
[Remove this section if it isn't needed —SS]

This document outlines the comprehensive verification and validation plan for Audio360, an assistive device designed to aid individuals who are deaf or hard of hearing by providing real-time visual indications of sound source locations and classifications through smart glasses. The plan ensures that the system meets all specified requirements and delivers reliable, safe functionality for its intended users.
The verification and validation process is structured around three main phases: verification of the Software Requirements Specification (SRS), design verification, and implementation verification. This plan addresses both functional and non-functional requirements through systematic testing approaches, including unit testing, integration testing, and user validation sessions. The roadmap prioritizes safety-critical features first, ensuring that the most important functionality is thoroughly validated before moving to enhanced features.

2 General Information

2.1 Summary

Audio360 is an embedded assistive device that processes real-time audio signals from a microphone array mounted on smart glasses to provide visual feedback about sound source locations and classifications. The system consists of several key components: embedded firmware for real-time audio processing, audio filtering for frequency domain conversion, direction of arrival (DoA) analysis for spatial awareness, sound classification for identifying audio sources, and a visualization controller for displaying information to users. The software being tested includes the complete embedded system running on a microcontroller, with components for audio capture, signal processing, classification, directional analysis algorithms, and visual output generation. The system operates as a closed embedded environment with no external connectivity, ensuring reliability and security for its safety-critical application domain.

2.2 Objectives

The primary objectives of this verification and validation plan are:
Primary Objectives:

- **Software Correctness:** Build confidence that the system correctly implements all functional requirements, particularly safety-critical features such as direction of arrival estimation and sound classification with 90% accuracy.
- **Real-time Performance:** Demonstrate that the system meets all timing constraints, including processing audio at 16 kHz sampling rate and providing visual feedback within 1 second latency.
- **User Safety and Usability:** Validate that the system effectively addresses the needs of individuals who are deaf or hard of hearing through structured user testing sessions with the McMaster Sign Language Club.
- **System Reliability:** Ensure robust error handling and fault tolerance, particularly for memory management and hardware component failures.

Objectives Out of Scope:

- **External Library Verification:** Third-party libraries (such as FFT implementations and machine learning frameworks) are assumed to be verified by their respective development teams. We will focus on testing our integration and usage of these libraries.
- **Long-term Durability Testing:** Extended wear testing and long-term hardware reliability assessment are beyond the scope of this academic project due to time constraints.
- **3D Spatial Localization:** Testing of elevation angle determination is out of scope as the system is designed for 2D horizontal plane analysis only.

This prioritization ensures that critical safety and functionality requirements are thoroughly validated while acknowledging resource limitations and project scope constraints.

2.3 Challenge Level and Extras

Challenge Level: Advanced

This project operates at an advanced challenge level due to the complex integration of real-time signal processing, embedded systems development, and safety-critical applications. The system requires sophisticated audio processing techniques including FFT analysis, direction of arrival estimation, and real-time classification algorithms running on resource-constrained hardware. A particularly challenging aspect is achieving precise microphone synchronization, as any timing discrepancies between microphones will render direction of arrival estimation impossible, effectively making the system's primary feature unachievable.

Extras:

- **Usability Testing:** Comprehensive user validation sessions with members of the McMaster Sign Language Club to ensure the system effectively addresses the needs of individuals who are deaf or hard of hearing. This includes structured interviews and observation sessions to validate both functionality and user experience.
- **Code Walkthroughs:** Systematic peer review processes for critical system components, particularly audio processing algorithms and safety-critical code paths. This ensures code quality and helps identify potential issues early in the development process.
- **User Documentation:** Development of comprehensive user guides and instructional materials to help end-users effectively utilize the system. This includes setup instructions, usage scenarios, and troubleshooting guides.

These extras enhance the project's value by ensuring both technical excellence and practical usability for the target user community.

2.4 Relevant Documentation

The following documentation serves as the foundation for the verification and validation activities:

Primary Requirements Documentation:

- **Software Requirements Specification (SRS):** [1] - The primary source of functional and non-functional requirements that drive all testing activities. Each test case is directly traceable to specific requirements in this document, ensuring comprehensive coverage of all specified functionality.

- **Problem Statement and Goals:** [2] - Provides the high-level objectives and constraints that inform the prioritization of testing activities, particularly the safety-critical nature of the application.

Development Documentation:

- **Development Plan:** [3] - Outlines the development methodology and testing tools, providing context for the verification and validation approach and timeline.
- **Hazard Analysis:** [4] - Identifies potential safety risks and failure modes that must be addressed through specific testing scenarios, particularly for safety-critical components.

User-Focused Documentation:

- **Verification and Validation Report:** [5] - Documents the results of testing activities and provides a record of system validation for future reference and continuous improvement.

These documents collectively provide the complete context needed for effective verification and validation, from high-level requirements through detailed implementation specifications to user-focused validation criteria.

3 Plan

This section will go over the verification and validation team. It will then be followed by the plan to verify the SRS, design, verification and validation plan, and implementation. Finally the section will end off with automated testing and verification tools and software validation.

3.1 Verification and Validation Team

The table below outlines the roles and responsibilities of each team member involved in the verification and validation process. Roles were intentionally assigned to individuals not directly responsible for the corresponding implementation components, ensuring an unbiased evaluation of system functionality. The supervisor also contributes by providing technical oversight and expert validation for signal processing.

Table: Verification and validation team breakdown

Role	**Description**	**Assignee**
Firmware Verification	Develops and executes tests to confirm that the firmware implementation conforms to the requirements outlined in the software specification.	Jay
Visualization Verification + Validation	Develops and executes tests to confirm that the visualization implementation conforms to the requirements outlined in the software specification. Also responsible for engaging with users to validate the usability of the product specific to the visualization.	Nirmal
Audio Classification Verification	Develops and executes tests to confirm that the audio classification module conforms to the requirements outlined in the software specification.	Sathurshan
Directional Analysis Verification	Develops and executes tests to confirm that the directional analysis component conforms to the requirements outlined in the software specification.	Omar
Product Validation	Responsible for engaging with individuals who are hard of hearing to validate that the system effectively addresses their pain points related to situational awareness.	Kalp
Audio Processing Verification	Reviews and assesses the audio processing methodologies implemented. The team will demonstrate and explain these techniques in a supervisor meeting and receive verbal or written feedback.	MVM (Supervisor)

3.2 SRS Verification

The verification of the SRS will follow a structured and systematic process. Each software requirement will be associated with at least one corresponding test case, which will verify whether the implementation satisfies the intended specification. Both unit and integration testing will be conducted to confirm functionality at the component and system levels. To avoid bias, test cases will be developed and executed by a team member who was not directly involved in the implementation of the component. These fall under the firmware verification, visual verification, audio classification verification, directional analysis verification roles.

The team will adopt a new GitHub peer review process to SRS verification. A comment will be added by bot to the PR. The comment will contain a list of reminders for the reviewers to confirm that the software implementation and/or written test cases comply with the requirements defined in the SRS. If a requirement cannot be met, the reviewer will be instructed to request an update in a separate PR linked with a rationale for the change. Below is the text that is included in the bot's comment addressing this topic.

> *Reviewer's Note*
>
> *- Ensure that all implemented features and/or test cases comply with the SRS. If any requirement cannot be met, link a separate PR updating the SRS and explaining the rationale for the change.*

The supervisor verification process will follow a formal meeting based review approach. During these meetings, the team will present core system elements, mainly audio processing methodologies, using mathematical descriptions, prototype demonstrations, and graphed data. The supervisor will be provided with targeted review questions and asked to identify potential weaknesses or missing test cases. Feedback will be documented, and resulting action items will be tracked and resolved through the project's issue tracker.

For validation, the team will engage users who are hard of hearing in structured sessions. These sessions will include observation of product use and semi-structured interviews. The observation aspect aims to allow the team understand the usability of the product while the interview serves as a method to identify validation issues in addressing user's needs related to situational awareness. This will fall under the visual validation and product validation roles.

3.3 Design Verification

The design verification process will include structured peer reviews conducted by the team. Below is the checklist to verify the design.

Software Core Architecture
☐ Does the selected software architecture appropriately support the system's requirements and intended functionality?
☐ Is the software design portable, allowing the software to be easily integrated with different hardware or simulation layer.

Software Design
☐ Is the system decomposed into small, modular components that can be individually tested?
☐ Are encapsulation principles followed, ensuring that data and functions that should be private are private?
☐ Are design assumptions, dependencies, and interfaces clearly defined and documented?
☐ Are software design principles being followed. Check box should fail if there is an another design principle that will be better fitted.

General
☐ Is there a corresponding UML diagram of the design being tracked on git.

Reviewers will document feedback on any checklist criteria that are not satisfied and provide recommendations for improvement. The team will track all feedback using the project's issue tracker.

3.4 Verification and Validation Plan Verification

The verification and validation plan verification process will include structured peer reviews conducted by the Teaching Assistant and Team 13. They will use the checklist from `Checklists/VnV-Checklist.pdf`.

3.5 Implementation Verification

As outlined in the development plan, the primary source code implementation will be developed in C/C++. The compiler used to build the source code will provide warnings of potential bugs. The team will resolve all warnings that is under the team's control, this excludes warnings from imported libraries.

The team will also employ Clang Static Analyzer [6] as a static analyzer tool. The static analyzer will be employed to detect bugs without running the source code on the hardware, and will be ran prior to merging PRs. It will block the PR from merging until all issues identified by the static analyzer has been resolved.
The team will also develop test that verify requirements. These tests are outline in the System Tests section.

3.6 Automated Testing and Verification Tools

Automated testing and verification tools are defined in the following sections from the Development Plan document.

- 10.3: Linter, Static Analyzer and Formatting Tools
- 10.4: Testing Frameworks (section also includes code coverage)
- 10.5: CI/CD (contains automated testing plan in CI)

As the software will be deployed on an embedded device, running unit tests on hardware is infeasible. As a result, all unit tests will be done on developer's local machine without any hardware in loop.

3.7 Software Validation

The Product Validation role, defined in section 3.1: Verification and Validation Team, is responsible for validating the product with the primary stakeholder. The product is composed of both software and hardware with more focus on the software. The validation will be conducted primarily with members of the McMaster Sign Language Club, who may not have the technical expertise to evaluate the requirements from the SRS. To address this, the Product Validation team member will conduct semi-structured interviews as described in section 3.2: SRS Verification. Rev 0 demo will include the results of the user validation, providing an opportunity to gather feedback and improve the software.

4 System Tests

This section outlines the tests for verifying and validating the functional and nonfunctional requirements outlined in the SRS [1]. 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 Embedded Firmware Tests

This section covers the tests for ensuring the embedded firmware correctly manages system tasks, audio synchronization, error handling, and hardware diagnostics. Each test is associated with a functional requirement defined under section 3.2.1 of the SRS. As such, each test will verify whether the system meets the associated functional requirement.

1. **test-FR-1.1** Priority-based task scheduling

   Control: Automatic

   Initial State: The firmware is deployed on the microcontroller with a task scheduler initialized. Multiple tasks with different priority levels are registered in the system.

   Input: Three concurrent tasks submitted to the firmware scheduler: Task A with high priority (priority level 1), Task B with medium priority (priority level 5), and Task C with low priority (priority level

10). Each task logs its execution start time with microsecond precision when it begins execution.

Output: Execution logs show that tasks execute in priority order: Task A starts first, followed by Task B, then Task C. The logged timestamps confirm this execution sequence regardless of submission order.

Test Case Derivation: Priority-based scheduling ensures that critical tasks (audio processing, safety diagnostics) execute before less critical tasks. The scheduler must respect priority assignments to meet real-time constraints. Higher priority tasks (lower numerical values) must preempt or execute before lower priority tasks.

How test will be performed: Deploy test firmware with three mock tasks to the microcontroller. Submit all three tasks simultaneously. Collect execution logs from the microcontroller. Analyze timestamps to verify execution order matches priority order. The test passes if Task A executes before Task B, and Task B executes before Task C in all test runs.

2. **test-FR-1.2** Audio signal synchronization

   Control: Manual

   Initial State: The firmware is deployed on the microcontroller with all four microphones connected and initialized. Audio capture subsystem is ready to receive data.

   Input: An impulse sound (hand clap or tone burst) played in the environment, captured simultaneously by all four microphones. The same acoustic event should be detected by all microphones with known time delays based on geometry.

   Output: Four synchronized audio buffers, each containing the impulse signal. Cross-correlation analysis between channels shows time alignment within margins of 10 microseconds, accounting for physical microphone spacing. All channels have matching sample rates and frame timestamps.

   Test Case Derivation: Accurate direction of arrival estimation requires precise temporal alignment across microphone channels. Misalignment exceeding 10 microseconds will degrade localization accuracy per the system's spatial resolution requirements.

How test will be performed: Generate acoustic impulse at known location. Capture audio on all four channels for 1 second. Export captured buffers to analysis software. Compute cross-correlation between all channel pairs. Calculate maximum time delay between channels. The test passes if all channel pairs show time alignment within 10 microseconds after compensating for physical propagation delays.

3. **test-FR-1.3** Memory error handling

   Control: Automatic

   Initial State: The firmware is running on the microcontroller with memory monitoring enabled. Available heap and stack memory are known and logged.

   Input: A test sequence that attempts to allocate memory buffers of increasing size until allocation would exceed available heap memory. The sequence includes: (1) valid allocation within limits, (2) allocation at memory limit, (3) allocation exceeding memory limit.

   Output: For valid allocations, memory is allocated successfully and functions return success codes. For allocation exceeding limits, the firmware detects the condition, denies the allocation, returns an appropriate error code, and continues operation without crashing. System remains stable and responsive after handling the error.

   Test Case Derivation: Memory exhaustion on embedded systems causes system crashes if not handled properly. The firmware must detect out-of-memory conditions before they cause undefined behavior. Graceful error handling prevents system failures that would leave users without situational awareness.

   How test will be performed: Execute automated test suite that progressively allocates memory buffers. Monitor system logs for allocation attempts and results. Verify that memory allocations within limits succeed. Verify that allocations exceeding limits return error codes without system crash. Confirm system continues responding to commands after error conditions. The test passes if all error conditions are caught and the system remains operational.

4. **test-FR-1.4** Hardware diagnostics

Control: Manual

Initial State: The firmware is deployed with all hardware components (microphones, display, etc.) connected and operational. Diagnostic subsystem is initialized and logging is enabled.

Input: A sequence of hardware fault conditions introduced systematically: (1) disconnect one microphone, (2) disconnect display interface, (3) restore all connections. Normal operation continues between fault injections.

Output: For each fault condition, the diagnostic system detects the hardware error within 1 second and logs appropriate error codes identifying the failed component. When connections are restored, the diagnostic system detects recovery and clears error flags. All detections and recoveries are timestamped in logs.

Test Case Derivation: Continuous hardware monitoring enables the system to detect failures in real-time and inform users about degraded functionality. Quick detection (within 1 second) ensures users are aware of system limitations. Component identification in error logs aids in troubleshooting and repair.

How test will be performed: Deploy firmware to microcontroller with all peripherals connected. Start diagnostic logging. Systematically introduce each fault condition while monitoring logs. Verify each fault is detected within 1 second by examining log timestamps. Verify correct error codes are generated for each fault type. Restore connections and verify recovery detection. The test passes if all faults are detected within timing constraints with correct error identification.

4.1.2 Driver Layer Tests

This section covers the tests for ensuring the driver layer provides correct hardware abstraction, permission handling, error propagation, and data integrity. Each test is associated with a functional requirement defined under section 3.2.2 of the SRS. As such, each test will verify whether the system meets the associated functional requirement.

1. **test-FR-2.1** Hardware interface abstraction

Control: Automatic

Initial State: The driver layer is compiled and running on the microcontroller. Hardware peripherals (GPIO for display, timers, etc.) are connected and initialized. Test application with driver API access is ready to execute.

Input: A test sequence that invokes driver API functions for different hardware operations: (1) read from ADC channel, (2) write to GPIO pin, (3) configure timer, (4) read system clock. Each operation uses the driver's abstracted interface rather than direct hardware register access.

Output: All driver API calls complete successfully and return expected data. ADC reads return valid voltage samples. GPIO writes toggle pin states visible on oscilloscope. Timer configuration produces expected timing behavior. System clock reads return monotonically increasing timestamps. No direct hardware register access is required by the test application.

Test Case Derivation: Hardware abstraction through the driver layer enables higher-level software to interact with peripherals without hardware-specific knowledge. This abstraction makes the software portable and maintainable. The driver must provide complete functionality for all hardware operations required by application components.

How test will be performed: Compile test application that uses only driver API calls (no direct register access). Deploy to microcontroller. Execute test sequence exercising each driver function. Verify all operations complete successfully through return codes. Use oscilloscope to verify GPIO toggles. Use serial logging to verify ADC readings are within expected ranges. The test passes if all driver operations execute correctly and provide expected functionality.

2. **test-FR-2.2** Permission-based hardware access

   Control: Automatic

   Initial State: The driver layer is running with a permission system configured. Two software modules are registered: Module A with high privilege (full hardware access), Module B with restricted privilege (limited hardware access).

Input: Test requests from both modules attempting to access hardware resources: (1) Module A requests ADC access (allowed by permissions), (2) Module B requests ADC access (denied by permissions), (3) Module A requests GPIO write (allowed), (4) Module B requests GPIO write (denied).

Output: Module A's requests complete successfully with hardware operations executed. Module B's requests are rejected by the driver layer, returning permission denied error codes without executing hardware operations. System logs show permission checks for all requests.

Test Case Derivation: Permission-based access control prevents unauthorized or unintended hardware operations that could compromise system safety or stability. The driver must enforce access policies to maintain system integrity. Only authorized components should control critical hardware like ADCs and display interfaces.

How test will be performed: Configure driver with permission rules for test modules. Execute test requests from both modules in sequence. Verify Module A's requests succeed by checking return codes. Verify Module B's requests are denied by checking for error codes. Examine system logs to confirm permission checks occurred. The test passes if all access decisions match configured permissions.

3. **test-FR-2.3** Error code propagation

   Control: Automatic

   Initial State: The driver layer is running on the microcontroller. Hardware is in a state where specific error conditions can be triggered (e.g., ADC not initialized, invalid GPIO pin specified).

   Input: A sequence of driver API calls designed to trigger various error conditions: (1) read from uninitialized ADC channel, (2) write to invalid GPIO pin number, (3) configure timer with out-of-range parameters, (4) read from disconnected peripheral.

   Output: For each error condition, the driver returns a specific error code identifying the failure type (e.g., ERROR_NOT_INITIALIZED, ERROR_INVALID_PIN, ERROR_INVALID_PARAMETER, ERROR_HARDWARE_DISCO
   Error codes are returned immediately without hanging or crashing.

Higher-level software receives these codes and can take appropriate action.

Test Case Derivation: Proper error reporting enables higher-level software to detect and handle hardware failures gracefully. Error codes must be specific enough to identify the failure cause. Immediate return of error codes (without blocking) maintains real-time system responsiveness.

How test will be performed: Execute test sequence that triggers each error condition. Capture return codes from driver API calls. Verify each return code matches the expected error type. Measure time from API call to return to confirm immediate propagation (less than 10 milliseconds). The test passes if all error conditions return correct, specific error codes within timing constraints.

4. **test-FR-2.4** Data integrity maintenance

   Control: Automatic

   Initial State: The driver layer is managing multiple memory regions for audio buffers, each actively being written by ADC DMA (Direct Memory Access) and read by audio processing software. System is configured for continuous audio capture.

   Input: Continuous audio input for 30 seconds, captured into circular buffers managed by the driver layer. Test includes concurrent read and write operations to simulate real-time processing conditions.

   Output: Audio data in buffers maintains integrity throughout the test period. No buffer corruption occurs (verified by checksum or known test pattern). No data races between writers (DMA) and readers (audio processing) cause invalid data. Buffer state transitions are atomic and consistent.

   Test Case Derivation: Data integrity is critical in audio processing systems where corrupted samples lead to incorrect analysis results. The driver must protect shared memory regions from concurrent access issues. Proper buffer management prevents overruns and data races that would degrade system reliability.

   How test will be performed: Configure driver to manage audio buffers with continuous DMA writes. Start audio capture with simultaneous reading by test application. Inject known test patterns into audio stream at marked intervals. Verify test patterns are correctly received without corruption. Monitor for buffer overflow or underflow conditions. The test passes if all test patterns are received intact and no memory corruption is detected over the 30-second duration.

4.1.3 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.

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

   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 main-lobe 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 configuration on the microcontroller, and another on the local computer. Execution time will be measured with a profiler. A test function will be written to measure the numerical equivalence of both outputs after processing is completed. Profiler measurements will be manully inspected to verify that 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.

4.1.4 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.

4.1.5 Audio360 Engine Tests

This section covers the tests for determining the accuracy of the Audio360 Engine in estimating the direction of incoming audio sources and their classification. Error handling of the Audio360 Engine will also be evaluated. Each test is associated with a functional requirement defined under section 3.2.4 of the SRS.

1. **Test-FR-4.1** Frequency and Error input capability testing.

   Control: Automatic

   Initial State: The Audio360 Engine module and its submodules have successfully initialized and are ready to process audio from the audio filtering module or microphone array. There are no errors flagged to the Audio360 Engine module.

   Subtest 1: Frequency Input Capability

   Input: A data stream of frequency-domain audio data sampled at 16 kHz by an array of 4 microphones. The audio data contains sine waves at low (100 Hz), mid (1 kHz), and high (8 kHz) frequency ranges. No error flags are set in the input data.

   Output: The Audio360 Engine module accepts the input data with no errors. There should be no error flags set in the output of the Audio360 Engine module. There should also be a direction and classification output based on the input audio frequency domain data.

   Subtest 2: Error Input Capability

   Control: Automatic

   Input: A stream of random frequency-domain audio data sampled with a 16 kHz sample rate. The input data will have error flags set, indicating either clipping, lost signal, or silence.

   Output: The Audio360 Engine module accepts the input data with error flags. The output does not contain a valid direction or classification. The output error flags should reflect the input error flags.

   Test Case Derivation: The Audio360 Engine module must be able to handle both valid frequency-domain audio data and audio data with error flags. Upon receiving error flags, the module is expected to fail gracefully without crashing, and propagate the error if necessary. The outputs in both subtests define the expected behavior of the module under different input conditions.

How test will be performed: Saved audio recordings from the microphone array will be used as artifacts in the automated testing framework. Random audio data with error flags will be generated at runtime. These tests will trigger when a pull request is made to any branch. Inputs will be passed to the Audio360 Engine module, and outputs will be checked against the expected outputs.

2. **Test-FR-4.2** Dependent component notification

   Control: Automatic

   Initial State: The Audio360 Engine module is deployed on the microcontroller and initialized and has received new audio data from the audio filtering module without error flags.

   Input: Mock audio data from the audio filtering module with no error flags.

   Output: All components in the system are expected to log important events for debugging purposes. The Audio360 Engine module starts the processing pipeline by notifying the dependent components to begin processing the new audio data. The sent notification and received acknowledgment from dependent components should be logged in the debugging logs for verification of pipeline integrity.

   Test Case Derivation: The Audio360 Engine module must notify its dependent components to start processing new audio data once it has received the data from the audio filtering module. This forms the basis of a streaming audio processing pipeline.

   How test will be performed: The test will be performed automatically by simulating the input of new audio data from the audio filtering module. The dependent components will be monitored through debugging logs to verify that they receive the notification and start processing the new audio data in the correct order.

3. **Test-FR-4.3** Pipeline flow management.

   Control: Automatic

   Initial State: The Audio360 Engine module has received new audio data from the audio filtering module without error flags and has notified its dependent components to start processing the new audio data.

Input: Mock audio data from the audio filtering module with no error flags.

Output: The Audio360 Engine module manages the flow of data through the processing pipeline. It ensures that each component processes the data in the correct order and that the output of one component is correctly passed as input to the next component. The flow of data through the pipeline should be logged for verification. Every dependent component in the pipeline is executed correctly and in order.

Test Case Derivation: The Audio360 Engine module must manage the flow of data through the processing pipeline. Since individual dependent components are not expected to be aware of other components in the pipeline, the Audio360 Engine module must ensure that data is passed correctly between components, ensuring data integrity. The module also needs to consider error states of each module to ensure that the pipeline can handle errors gracefully.

How test will be performed: The test will be performed automatically by simulating the input of new audio data from the audio filtering module. The flow of data through the pipeline will be monitored through debugging logs to verify that each component processes the data in the correct order and that the output of one component is correctly passed as input to the next component.

4. **Test-FR-4.4** Error based pipeline suppression.

   Control: Automatic

   Initial State: The Audio360 Engine module has received new audio data from the audio filtering module with error flags set.

   Input: Mock audio data from the audio filtering module with error flags set, indicating either clipping, lost signal, or silence.

   Output: The Audio360 Engine module detects the error flags in the input data and suppresses the processing pipeline. The module should log the error and propagate the error flags to dependent components without attempting to process the audio data.

Test Case Derivation: The Audio360 Engine module must be able to handle error flags in the input data. Upon receiving error flags, the module is expected to fail gracefully without crashing, and propagate the error if necessary. The processing pipeline should be suppressed to prevent unnecessary processing of invalid data.

How test will be performed: The test will be performed automatically by simulating the input of new audio data from the audio filtering module with error flags set. The processing pipeline will be monitored through debugging logs to verify that the pipeline is suppressed and that error flags are handled appropriately. Once the error has been resolved by clearing the error flags, the pipeline should resume normal operation.

4.1.6 Frequency Analysis Tests

This section covers the tests for ensuring the Frequency Analysis component which is responsible for analyzing frequency-domain audio data to estimate the direction of incoming audio sources and their classification. Each test is associated with a functional requirement defined under section 3.2.5 of the SRS.

1. **Test-FR-5.1** Audio Classification

   Control: Automatic

   Initial State: The Frequency Analysis module is initialized and ready to process frequency- domain audio data retrieved from the Audio360 Engine module.

   Input: A set of frequency-domain audio data samples representing different audio classes. The classification of each sample is known beforehand for verification purposes.

   Output: The Frequency Analysis module processes each input sample and outputs the predicted audio class. The predicted class should match the known class for each sample with a class level accuracy of at least 90%.

   Test Case Derivation: The Frequency Analysis module is responsible for classifying audio sources based on their frequency-domain characteristics. By providing known samples, the module's classification accuracy can be evaluated.

How test will be performed: The test will be performed automatically by feeding the frequency domain data of the known audio samples to the Frequency Analysis module. The predicted classes will be compared against the known classes to calculate the classification accuracy. The test passes if the accuracy meets or exceeds the 90% threshold per class.

Accuracy is calculated as follows:

$$\text{Accuracy}_{\text{class}} = \frac{\text{Number of Correct Predictions For Class}}{\text{Total Number of Samples for Class}} \times 100\%$$

The overall accuracy across all classes is calculated as follows:

$$\text{Overall Accuracy} = \frac{\Sigma_{\text{class} \in \text{tests}} \text{Accuracy}_{\text{class}}}{Number of Classes}$$

2. **Test-FR-5.2** Direction Estimation

   Control: Automatic

   Initial State: The Frequency Analysis module is initialized and ready to process frequency- domain audio data retrieved from the Audio360 Engine module for a four microphone array.

   Input: A set of frequency-domain audio data samples representing audio sources originating from known directions relative to the microphone array.

   Output: The Frequency Analysis module processes input samples from the four microphone array and outputs the estimated direction of each audio source. The estimated direction should be within +/- 45 degrees of the known direction for each sample.

   Test Case Derivation: The Frequency Analysis module is responsible for estimating the direction of audio sources based on the frequency-domain characteristics captured by the microphone array. By providing samples from known directions, the module's direction estimation accuracy can be evaluated.

How test will be performed: The test will be performed automatically by feeding the frequency domain data of the known direction samples to the Frequency Analysis module. The estimated directions will be compared against the known directions to calculate the estimation accuracy. All estimated directions must be within the +/- 45 degree threshold of the known directions. In addition, the Mean Absolute Error (MAE) will be calculated to quantify the average direction estimation error across all samples. The MAE will need to be below 22.5 degrees for the test to pass using the formula:

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N} |\text{Estimated Direction}_i - \text{Known Direction}_i|$$

where N is the total number of samples.

3. **Test-FR-5.3** Radian Units.

   Control: Automatic

   Initial State: The Frequency Analysis module is initialized and ready to process frequency- domain audio data retrieved from the Audio360 Engine module.

   Input: A set of random frequency-domain audio data samples.

   Output: The Frequency Analysis module processes input samples from the four microphone array and outputs the estimated direction of each audio source in radians.

   Test Case Derivation: As per functional requirement FR5.3, the Frequency Analysis module must output direction estimates in radians. This test verifies that the module adheres to this requirement.

   How test will be performed: The test will be performed automatically by feeding the frequency domain data of the known direction samples to the Frequency Analysis module. The output directions will be checked to ensure they are expressed in radians. This will be done by verifying that the output values fall within the range $[0, 2\pi]$ radians.

4. **Test-FR-5.4** Low confidence notification.

Control: Automatic

Initial State: The Frequency Analysis module is initialized and ready to process frequency- domain audio data retrieved from the Audio360 Engine module.

Input: A set of frequency-domain audio data samples that are randomly generated, making it difficult to accurately classify the audio source. Known sample data will also be included for control purposes to determine if the module can differentiate between low and high confidence samples.

Output: The Frequency Analysis module processes each input sample and outputs the predicted classification along with a confidence score. For ambiguous samples, the confidence score should be below a predefined threshold and the module should flag the output as low confidence.

Test Case Derivation: The Frequency Analysis module must be able to handle ambiguous or noisy audio data and indicate when its predictions are uncertain. This test verifies that the module correctly flags low-confidence outputs. The low confidence output should be logged for verification and visible to the user through the visualization controller.

How test will be performed: The test will be performed automatically by feeding random frequency-domain audio data samples **and** known samples to the Frequency Analysis module. The predicted classifications and confidence scores will be evaluated to verify that low-confidence outputs are correctly flagged. The test passes if all ambiguous samples are flagged as low confidence and the known samples are classified with high confidence. The confidence threshold will be calibrated based on real world testing.

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 Audio360 Engine Tests

1. **Test-NFR4.1** No memory race conditions

   Type: Non-Functional, Static, Automatic

   Initial State: The Audio360 Engine module is deployed on the microcontroller and initialized.

   Input/Condition: The Audio360 Engine module receives simulated audio data from the audio filtering module.

   Output/Result: The Audio360 Engine module processes the incoming audio data without modifying the data in the original buffer used by the audio filtering module.

   How test will be performed: The test will be performed automatically using the Clang toolchain to analyze the function calls and memory accesses (read/write) within the Audio360 Engine module. The tool will check for any instances where the Audio360 Engine module writes to memory locations that are passed as read-only inputs. If no such instances are found, the test passes, confirming that there are are no memory race conditions. This is important to ensure since the buffers used by the audio filtering module are most likely serviced by interrupts.

2. **Test-NFR4.2** Real-time processing

   Type: Non-Functional, Automatic

   Initial State: The Audio360 Engine module is deployed on the microcontroller and initialized.

   Input/Condition: A continuous stream of frequency-domain audio data sampled at 16 kHz from the audio filtering module.

   Output/Result: The Audio360 Engine module processes each incoming audio data frame before the next frame arrives, maintaining real-time processing.

   How test will be performed: The test will be performed automatically by simulating a continuous stream of frequency-domain audio data from the audio filtering module. The Audio360 Engine module will log the processing time for each frame. The logs will be analyzed to verify that the processing time for each frame does not exceed the time interval between frames (1/16,000 seconds). If all frames are processed within this time constraint, the test passes real-time processing constraints. This test will have to be performed on the microcontroller to ensure accurate timing measurements.

3. **Test-NFR4.3** Microphone audio data and all derivative data is discarded after processing.

   Type: Non-Functional, Static, Manual

   Initial State: The Audio360 Engine module is initialized and ready to process frequency-domain audio data retrieved from the audio filtering module.

   Input/Condition: A set of random frequency-domain audio data samples.

   Output/Result: After processing each input sample, the Audio360 Engine module or any of its dependent components do not retain any copies of the original microphone data or any derivative data in memory.

   How test will be performed: The test will be performed manually by inspecting the source code of the Audio360 Engine module and its dependent components. The code will be reviewed to ensure that there are no persistent data structures or variables that retain copies of the original microphone data or any derivative data.

4.2.2 Embedded Firmware

This section covers the tests for the non-functional requirements related to the embedded firmware as defined in section 3.2.1 of the SRS.

1. **test-NFR1.1** Monotonic frame sequence processing

   Type: Non-Functional, Dynamic, Automatic

   Initial State: The firmware is deployed on the microcontroller with audio capture active. Multiple audio frames are being generated continuously from microphone input. Frame timestamps and sequence numbers are logged.

Input/Condition: Continuous audio input for 60 seconds with audio frames arriving at the firmware scheduler. Each frame has a timestamp indicating its capture time. Frames may arrive with slight jitter but maintain chronological order based on capture time.

Output/Result: Processing logs show that frames are processed in strict chronological order based on their capture timestamps. No newer frame is processed before an older frame. Earliest frames consistently receive highest priority in the task queue. All 60 seconds of audio are processed without frame reordering.

How test will be performed: Deploy firmware to microcontroller with frame logging enabled. Capture 60 seconds of continuous audio. Export frame processing logs. Analyze log sequence to verify timestamps are monotonically increasing (each processed frame has a timestamp greater than or equal to the previous frame). The test passes if no out-of-order processing is detected across the entire 60-second capture period.

2. **test-NFR1.2** Real-time processing speed

   Type: Non-Functional, Dynamic, Manual

   Initial State: The firmware is deployed on the microcontroller with all audio processing components active. Microphones are configured to sample at 16 kHz. Performance monitoring is enabled to track processing frame rates.

   Input/Condition: Continuous audio input captured at 16 kHz for 5 minutes. This generates approximately 13,230,000 samples (44100 samples/second $\times$ 300 seconds) that must be processed in real-time. Processing includes audio capture, synchronization, and preparation for downstream components.

   Output/Result: Performance logs show that the firmware processes all audio frames without falling behind the input rate. Processing rate consistently exceeds 16 kHz (e.g., processes 44100 samples in less than 1 second). No buffer overflows occur. No frames are dropped due to processing delays.

How test will be performed: Deploy firmware to microcontroller and start continuous audio capture. Monitor processing rate through performance counters and logs over 5-minute duration. Calculate effective processing rate (samples processed per second). Check for buffer overflow indicators in logs. The test passes if processing rate remains above 16 kHz throughout the test and no overflows occur.

4.2.3 Driver Layer

This section covers the tests for the non-functional requirements related to the driver layer as defined in section 3.2.2 of the SRS.

1. **test-NFR2.1** Immediate error propagation

   Type: Non-Functional, Dynamic, Automatic

   Initial State: The driver layer is running on the microcontroller. Hardware peripherals are in various states (some operational, some generating errors). High-precision timing measurement is configured to track error propagation delay.

   Input/Condition: A sequence of N in the range of [50 , 1000] driver API calls that intentionally trigger various error conditions: uninitialized hardware access, invalid parameters, timeout conditions, and hardware disconnections. Each call is timestamped at invocation and when error code is returned.

   Output/Result: For all N error-triggering calls, the driver returns error codes within 10 milliseconds of the call. No error is delayed or queued for later reporting. The median error propagation time is less than 1 millisecond. All errors are reported to the firmware layer immediately without buffering.

   How test will be performed: Execute automated test suite with N error-inducing driver calls. Measure time from function entry to error code return using microcontroller cycle counters or high-precision timers. Calculate propagation delay for each call. Generate statistics (minimum, median, maximum, 99th percentile). The test passes if maximum delay is less than 100 milliseconds and median delay is less than 10 milliseconds.

4.2.4 Frequency Analysis Tests

1. **Test-NFR5.1** Simultaneous classification and direction estimation.

   Type: Non-Functional, Automatic

   Initial State: The Frequency Analysis module is initialized and ready to process frequency- domain audio data when notified by the Audio360 Engine module.

   Input/Condition: A frequency-domain audio data sample containing different audio classes originating from known directions relative to the microphone array. Different classes and directions of audio sources will be embedded in the same input sample.

   Output/Result: The Frequency Analysis module processes each input sample and outputs the predicted audio class and estimated direction for up to 3 simultaneous audio sources. The predicted classes and estimated directions should match the known classes and directions for each source within the defined accuracy thresholds defined in Frequency Analysis Tests.

   How test will be performed: The test will be performed automatically by feeding the frequency domain data of the known audio sample with multiple sources to the Frequency Analysis module. The predicted classes and estimated directions will be compared against the known classes and directions to calculate the accuracy.

2. **Test-NFR5.2** Single source classification accuracy.

   Type: Non-Functional, Automatic

   Initial State: The Frequency Analysis module is initialized and ready to process frequency- domain audio data retrieved from the Audio360 Engine module.

   Input/Condition: A set of frequency-domain audio data samples representing different known audio classes. Each sample contains a single isolated audio source.

   Output/Result: The Frequency Analysis module processes each input sample and outputs the predicted audio class. The predicted class should match the known class for each sample with a class level accuracy defined in Frequency Analysis Tests.

How test will be performed: The test will be performed automatically by feeding the frequency domain data of the known audio samples to the Frequency Analysis module. The predicted classes will be compared against the known classes to calculate the classification accuracy.

3. **Test-NFR5.3** Single source direction estimation accuracy.

   Type: Non-Functional, Automatic

   Initial State: The Frequency Analysis module is initialized and ready to process frequency- domain audio data retrieved from the Audio360 Engine module.

   Input/Condition: A set of frequency-domain audio data samples representing audio sources originating from known directions relative to the microphone array. Each sample contains a single isolated audio source.

   Output/Result: The Frequency Analysis module processes input samples from the four microphone array and outputs the estimated direction of each audio source. The estimated direction should be within the threshold defined in Frequency Analysis Tests.

   How test will be performed: The test will be performed automatically by feeding the frequency domain data of the known direction samples to the Frequency Analysis module. The estimated directions will be compared against the known directions to calculate the directional estimation accuracy.

4.2.5 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

4.2.6 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: Functional Requirements and Corresponding Test Sections

Test Section	**Supported Requirement(s)**
Embedded Firmware	FR-1.1, FR-1.2, FR-1.3, FR-1.4
Driver Layer	FR-2.1, FR-2.2, FR-2.3, FR-2.4
Audio Filtering	FR-3.1, FR-3.2, FR-3.3, FR-3.4, FR-3.5
Visualization Controller	FR-6.1, FR-6.2

Table: Non-Functional Requirements and Corresponding Test Sections

Test Section	**Supported Requirement(s)**
Embedded Firmware	NFR-1.1, NFR-1.2
Driver Layer	NFR-2.1
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]

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

The following 10 essential survey questions will be used during user validation sessions with members of the McMaster Sign Language Club who are hard of hearing to assess the effectiveness and usability of Audio360Audio360.

1. **Background:** What is your level of hearing loss? (Complete deafness / Severe / Moderate / Mild)

   Rationale: This question directly relates to Goal G.1.5 (user-friendly interaction) by ensuring the system is tested across a range of hearing loss levels. Understanding the user's specific hearing profile helps validate that the system meets the needs of the primary stakeholder group (individuals who are deaf or hard of hearing) as defined in the SRS.

2. **Visual Clarity:** How would you rate the clarity of the directional indicators? (1-5 scale: 1 = Very unclear, 5 = Very clear)

   Rationale: This question directly validates Goal G.1.4 (visual display) and Goal G.2 (direction of arrival analysis). Clear directional indicators are essential for users to understand where sounds are coming from, which is a core requirement for maintaining situational awareness and safety.

3. **Response Time:** Does the system respond quickly enough when sounds are detected? (Yes / No / Sometimes)

   Rationale: This question validates the real-time performance requirements from Goal G.1.2 and Goal G.3. The system must provide near real-time feedback to be effective for safety-critical applications, as delayed responses could lead to missed safety cues.

4. **Distraction Level:** Are the visual alerts distracting from your normal activities? (1-5 scale: 1 = Very distracting, 5 = Not distracting at all)

   Rationale: This question relates to Goal G.4 (non-obstructive display) and Goal G.1.6 (user comfort). The system must enhance situational awareness without interfering with normal activities, ensuring it integrates seamlessly into daily life rather than creating additional barriers.

5. **Sound Understanding:** Can you easily understand what type of sound the system is detecting? (Yes / No / Sometimes)

   Rationale: This question directly validates Goal G.1.3 (sound classification) and Goal G.1.5 (user-friendly interaction). Clear sound classification is essential for users to understand their environment and respond appropriately to different types of audio cues.

6. **Confidence:** How confident do you feel about your situational awareness while using this system? (1-5 scale: 1 = Not confident, 5 = Very confident)

   Rationale: This question measures the overall effectiveness of the system in achieving its primary objective of improving situational awareness for individuals who are deaf or hard of hearing. It validates that the combination of all system goals (G.1.1-G.1.6) successfully addresses the core problem of missed audio cues and safety risks.

7. **Safety:** Would you feel safer using this system in real-world scenarios? (Yes / No / Unsure)

   Rationale: This question directly addresses the safety-critical nature of the application mentioned in the SRS context. It validates that the system successfully mitigates the safety risks associated with missed audio cues (car approaching, alarms, etc.) that were identified as the primary motivation for the project.

8. **Comfort:** How comfortable is the smart glasses for extended wear? (1-5 scale: 1 = Very uncomfortable, 5 = Very comfortable)

   Rationale: This question directly validates Goal G.1.6 (user comfort for extended wear). Comfort is essential for daily use and adoption of the assistive technology, ensuring users can wear the system for extended periods without discomfort or fatigue.

9. **Recommendation:** Would you recommend this system to other individuals who are deaf or hard of hearing? (Yes / No / Maybe)

   Rationale: This question provides an overall assessment of user satisfaction and perceived value, indicating whether the system successfully meets the needs of the primary stakeholder group. A positive recommendation suggests the system effectively addresses the identified gaps in existing assistive technologies.

10. **Overall Feedback:** What was the most helpful aspect of using this system, and what improvements would you suggest?

    Rationale: This open-ended question allows users to provide qualitative feedback that can inform future iterations and improvements. It helps validate that the system's features align with user needs and identifies areas for enhancement to better achieve the project goals.

These questions will be administered through semi-structured interviews and observation sessions, providing both quantitative assessment and qualitative feedback to evaluate the system's effectiveness for the target user community.

References

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

[2] ——, "Problem statement and goals," https://github.com/Team6-SixSense/audio360/blob/main/docs/ProblemStatementAndGoals/ProblemStatement.pdf, 2025.

[3] ——, "Development plan," https://github.com/Team6-SixSense/audio360/blob/main/docs/DevelopmentPlan/DevelopmentPlan.pdf, 2025.

[4] ——, "Hazard analysis," https://github.com/Team6-SixSense/audio360/blob/main/docs/HazardAnalysis/HazardAnalysis.pdf, 2025.

[5] ——, "Verification and validation report," https://github.com/Team6-SixSense/audio360/blob/main/docs/VnVReport/VnVReport.pdf, 2025.

[6] LLVM, "Clang static analyzer," The LLVM Project, 2025. [Online]. Available: https://clang-analyzer.llvm.org/

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.

   Jay: The team's previous work on the SRS really helped because we already had a clear picture of what each component needed to do. This made it straightforward to figure out what to test and how to test it.

   Kalp: What worked really well for this document was actually our previous well written SRS document. Since we had already gone through the process of writing the SRS document, we were able to hit the ground running with the VnV plan. We were able to use the same structure and format for the VnV plan as we did for the SRS document, which made it easier to write. Referencing the SRS document was also really easy to do since it was well written, organized, and discussed.

   Omar: The team has done significant background research into the problem and the software architecture / methodologies that we will be dealing with. This made it straightforward to describe the various testing instruments that we will be using to verify and validate our system.

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.

Jay: Balancing this deliverable with other course work was really challenging. I kept putting it off thinking I could do it later, but then other assignments kept piling up. I resolved this by setting specific time blocks to work on this deliverable.

Kalp: The only pain point that I experienced wasn't even related to the document itself, but rather the fact that we had a lot of content to cover in the document, but with a lot of other course work as well at this time of year. Having to focus on the upcoming PoC implementation, as well as dealing with the midterm season made it quite difficult to focus on the VnV plan.

Omar: Defining specific test without sounding very reptitive throughout the document was a challenge. To resolve this, I made sure to carefully read through each test case and ensure that they were unique and specific to the requirement they were testing.

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: Omar will be pursuing this experience as he has experience with gtest from previous projects. He will be looking at how to write effective unit tests using gtest, as well as looking at best practices for writing unit tests for embedded systems.

   (b) Hardware debugging: Kalp will be pursuing this experience as his lack of experience with hardware debugging was felt as a technical blocker in his sklil set. He will be looking at how to debug hardware on a microcontroller for the project, but also be reading into documentation and practicing good hardware debugging practices on his own personal projects.

   (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: Jay will be pursuing this since he has experience with user research from his design background. He will work with the McMaster Sign Language Club to conduct structured interviews and usability testing sessions

   (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.