Module Interface Specification for Audio360

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

November 17, 2025

1 Revision History

Date	Version	Notes
2025-11-13	1.0	Initial write-up

2 Symbols, Abbreviations and Acronyms

symbol	**description**
Audio360	360 Audio analysis system on smart glasses
DOA	Direction of Arrival
FR	Functional Requirement
FFT	Fast Fourier Transform
ICA	Independent Component Analysis
M	Module
MG	Module Guide
MIS	Module Interface Specification
NFR	Non-functional Requirement
R	Requirement
SPI	Serial Peripheral Interface
SRS	Software Requirements Specification
USART	Universal Synchronous/Asynchronous Receiver/Transmitter

Table 1: Symbols, abbreviations and acronyms used in the MIS document.

See SRS Documentation at Symbols, Abbreviations, and Acronyms for a complete table used in Audio360.

Contents

1 Revision History i

2 Symbols, Abbreviations and Acronyms ii

3 Introduction 1

4 Notation 1

5 Module Decomposition 2

6 Data Types 3
- 6.1 Generics 3
  - 6.1.1 bool 3
  - 6.1.2 int8 3
  - 6.1.3 int16 3
  - 6.1.4 int32 3
  - 6.1.5 int64 3
  - 6.1.6 uint8 3
  - 6.1.7 uint16 3
  - 6.1.8 uint32 3
  - 6.1.9 uint64 3
  - 6.1.10 float32 3
  - 6.1.11 float64 3
  - 6.1.12 string 4
- 6.2 Enums 4
  - 6.2.1 Audio360State 4
  - 6.2.2 Audio360Status 4
  - 6.2.3 FaultState 4
  - 6.2.4 MicInputStatus 4
  - 6.2.5 AudioAnomalyStatus 4
- 6.3 Data Structures 5
  - 6.3.1 FrequencyDomain 5
  - 6.3.2 CALLBACK_MIC_DATA 5
  - 6.3.3 MIC_INPUT_CALLBACK 5
  - 6.3.4 Dictionary 5
  - 6.3.5 List 5

7 MIS of Microphone Input Module - M1 6
- 7.1 Module 6
- 7.2 Uses 6
- 7.3 Syntax 6
  - 7.3.1 Exported Constants 6

7.3.2 Exported Access Programs 6
7.4 Semantics 6
7.4.1 State Variables 6
7.4.2 Environment Variables 7
7.4.3 Assumptions 7
7.4.4 Access Routine Semantics 7
7.4.5 Local Functions 8

8 MIS of USART Communication Module - M2 9
8.1 Module 9
8.2 Uses 9
8.3 Syntax 9
8.3.1 Exported Constants 9
8.3.2 Exported Access Programs 9
8.4 Semantics 9
8.4.1 State Variables 9
8.4.2 Environment Variables 9
8.4.3 Assumptions 9
8.4.4 Access Routine Semantics 10
8.5 Local Functions 10

9 MIS of SD Card Interface Module - M3 11
9.1 Module 11
9.2 Uses 11
9.3 Syntax 11
9.3.1 Exported Constants 11
9.3.2 Exported Access Programs 11
9.4 Semantics 11
9.4.1 State Variables 11
9.4.2 Environment Variables 11
9.4.3 Assumptions 11
9.4.4 Access Routine Semantics 11
9.5 Local Functions 12

10 MIS of Microphone Diagnostic Module - M4 13
10.1 Module 13
10.2 Uses 13
10.3 Syntax 13
10.3.1 Exported Constants 13
10.3.2 Exported Access Programs 13
10.4 Semantics 13
10.4.1 State Variables 13
10.4.2 Environment Variables 13

10.4.3 Assumptions 13
10.4.4 Access Routine Semantics 14
10.5 Local Functions 14

11 MIS of Audio Generation Module - M5 15
11.1 Module 15
11.2 Uses 15
11.3 Syntax 15
11.3.1 Exported Constants 15
11.3.2 Exported Access Programs 15
11.4 Semantics 15
11.4.1 State Variables 15
11.4.2 Environment Variables 15
11.4.3 Assumptions 15
11.4.4 Access Routine Semantics 15
11.4.5 Local Functions 16

12 MIS of Direction of Arrival (DOA) Processor Module - M6 17
12.1 Module 17
12.2 Uses 17
12.3 Syntax 17
12.3.1 Exported Constants 17
12.3.2 Exported Access Programs 17
12.4 Semantics 17
12.4.1 State Variables 17
12.4.2 Environment Variables 17
12.4.3 Assumptions 17
12.4.4 Access Routine Semantics 18
12.4.5 Local Functions 18

13 MIS of Audio360 Engine - M7 19
13.1 Module 19
13.2 Uses 19
13.3 Syntax 19
13.3.1 Exported Constants 19
13.3.2 Exported Access Programs 19
13.4 Semantics 19
13.4.1 State Variables 19
13.4.2 Environment Variables 19
13.4.3 Assumptions 19
13.4.4 Access Routine Semantics 20
13.4.5 Local Functions 20

14 MIS of Audio Sampling Module - M8 21
14.1 Module . . . 21
14.2 Uses . . . 21
14.3 Syntax . . . 21
14.3.1 Exported Constants . . . 21
14.3.2 Exported Access Programs . . . 21
14.4 Semantics . . . 21
14.4.1 State Variables . . . 21
14.4.2 Environment Variables . . . 21
14.4.3 Assumptions . . . 21
14.4.4 Access Routine Semantics . . . 22
14.4.5 Local Functions . . . 22

15 MIS of Audio Spectral Leakage Prevention Module - M9 23
15.1 Module . . . 23
15.2 Uses . . . 23
15.3 Syntax . . . 23
15.3.1 Exported Constants . . . 23
15.3.2 Exported Access Programs . . . 23
15.4 Semantics . . . 23
15.4.1 State Variables . . . 23
15.4.2 Environment Variables . . . 23
15.4.3 Assumptions . . . 23
15.4.4 Access Routine Semantics . . . 23
15.4.5 Local Functions . . . 24

16 MIS of Audio Normalizer Module - M10 25
16.1 Module . . . 25
16.2 Uses . . . 25
16.3 Syntax . . . 25
16.3.1 Exported Constants . . . 25
16.3.2 Exported Access Programs . . . 25
16.4 Semantics . . . 25
16.4.1 State Variables . . . 25
16.4.2 Environment Variables . . . 25
16.4.3 Assumptions . . . 25
16.4.4 Access Routine Semantics . . . 25
16.4.5 Local Functions . . . 26

17 MIS of Audio Anomaly Detection Module - M11 27
17.1 Module . . . 27
17.2 Uses . . . 27
17.3 Syntax . . . 27

17.3.1 Exported Constants 27
17.3.2 Exported Access Programs 27
17.4 Semantics 27
17.4.1 State Variables 27
17.4.2 Environment Variables 27
17.5 Assumptions 27
17.5.1 Access Routine Semantics 28
17.6 Local Functions 28

18 MIS of Fast Fourier Transform - M12 **29**
18.1 Module 29
18.2 Uses 29
18.3 Syntax 29
18.3.1 Exported Constants 29
18.3.2 Exported Access Programs 29
18.4 Semantics 29
18.4.1 State Variables 29
18.4.2 Environment Variables 29
18.4.3 Assumptions 29
18.4.4 Access Routine Semantics 29
18.4.5 Local Functions 30

19 MIS of Logging Module - M13 **31**
19.1 Module 31
19.2 Uses 31
19.3 Syntax 31
19.3.1 Exported Constants 31
19.3.2 Exported Access Programs 31
19.4 Semantics 31
19.4.1 State Variables 31
19.4.2 Environment Variables 31
19.4.3 Assumptions 31
19.4.4 Access Routine Semantics 31
19.4.5 Local Functions 32

20 MIS of Fault Manager Module - M14 **33**
20.1 Module 33
20.2 Uses 33
20.3 Syntax 33
20.3.1 Exported Constants 33
20.3.2 Exported Access Programs 33
20.4 Semantics 33
20.4.1 State Variables 33

20.4.2 Environment Variables . . . 33
20.4.3 Assumptions . . . 33
20.4.4 Access Routine Semantics . . . 34
20.4.5 Local Functions . . . 34

21 MIS of Mel Filter Module - M15 35
21.1 Module . . . 35
21.2 Uses . . . 35
21.3 Syntax . . . 35
21.3.1 Exported Constants . . . 35
21.3.2 Exported Access Programs . . . 35
21.4 Semantics . . . 35
21.4.1 State Variables . . . 35
21.4.2 Environment Variables . . . 35
21.4.3 Assumptions . . . 35
21.4.4 Access Routine Semantics . . . 36
21.4.5 Local Functions . . . 36

22 MIS of Discrete Cosine Transform Module - M16 37
22.1 Module . . . 37
22.2 Uses . . . 37
22.3 Syntax . . . 37
22.3.1 Exported Constants . . . 37
22.3.2 Exported Access Programs . . . 37
22.4 Semantics . . . 37
22.4.1 State Variables . . . 37
22.4.2 Environment Variables . . . 37
22.4.3 Assumptions . . . 37
22.4.4 Access Routine Semantics . . . 38
22.4.5 Local Functions . . . 38

23 MIS of Principle Component Analysis Module - M17 39
23.1 Module . . . 39
23.2 Uses . . . 39
23.3 Syntax . . . 39
23.3.1 Exported Constants . . . 39
23.3.2 Exported Access Programs . . . 39
23.4 Semantics . . . 39
23.4.1 State Variables . . . 39
23.4.2 Environment Variables . . . 39
23.4.3 Assumptions . . . 39
23.4.4 Access Routine Semantics . . . 40
23.4.5 Local Functions . . . 40

24 MIS of Linear Discriminant Analysis Module - M18 **41**
24.1 Module . . . 41
24.2 Uses . . . 41
24.3 Syntax . . . 41
24.3.1 Exported Constants . . . 41
24.3.2 Exported Access Programs . . . 41
24.4 Semantics . . . 41
24.4.1 State Variables . . . 41
24.4.2 Environment Variables . . . 41
24.4.3 Assumptions . . . 41
24.4.4 Access Routine Semantics . . . 42
24.4.5 Local Functions . . . 42

25 MIS of Classification Module - M19 **43**
25.1 Module . . . 43
25.2 Uses . . . 43
25.3 Syntax . . . 43
25.3.1 Exported Constants . . . 43
25.3.2 Exported Access Programs . . . 43
25.4 Semantics . . . 43
25.4.1 State Variables . . . 43
25.4.2 Environment Variables . . . 44
25.4.3 Assumptions . . . 44
25.4.4 Access Routine Semantics . . . 44
25.4.5 Local Functions . . . 44

26 MIS of Visualization Module - M20 **45**
26.1 Module . . . 45
26.2 Uses . . . 45
26.3 Syntax . . . 45
26.3.1 Exported Constants . . . 45
26.3.2 Exported Access Programs . . . 45
26.4 Semantics . . . 45
26.4.1 State Variables . . . 45
26.4.2 Environment Variables . . . 46
26.4.3 Assumptions . . . 46
26.4.4 Access Routine Semantics . . . 46
26.4.5 Local Functions . . . 47

27 MIS of Independent Component Analysis (ICA) Module - M21 **48**
27.1 Module . . . 48
27.2 Uses . . . 48
27.3 Syntax . . . 48

27.3.1 Exported Constants 48
27.3.2 Exported Access Programs 48
27.4 Semantics 48
27.4.1 State Variables 48
27.4.2 Environment Variables 48
27.4.3 Assumptions 49
27.4.4 Access Routine Semantics 49
27.4.5 Local Functions 49

3 Introduction

The following document details the Module Interface Specifications for Audio360, an assistive device system for smart glasses that provides real-time visual indications of sound source locations and classifications to aid individuals who are deaf or hard of hearing.

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at https://github.com/Team6-SixSense/audio360.

4 Notation

The structure of the MIS for modules comes from [1], with the addition that template modules have been adapted from [2]. The mathematical notation comes from Chapter 3 of [1]. For instance, the symbol := is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1 | c_2 \Rightarrow r_2 | ... | c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by Audio360.

Data Type	**Notation**	**Description**
character	char	a single symbol or digit
integer	$\mathbb{Z}$	a number without a fractional component in (-∞, ∞)
natural number	$\mathbb{N}$	a number without a fractional component in [1, ∞)
real	$\mathbb{R}$	any number in (-∞, ∞)
array/list	x[]	Ordered list of type x.
negate	$\neg$	Logical math NOT.
and	$\wedge$	Logical math AND
or	$\vee$	Logical math OR
implies	$\implies$	Logical math implies
assignment	:=	For A := B. B is assigned to A.
for all	$\forall$	Referencing all items. For example: $(\forall\ variable \mid condition : statement)$

The specification of Audio360 uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, Audio360 uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

5 Module Decomposition

The following table is taken directly from the Module Guide document for this project.

Level 1	**Level 2**
Hardware-Hiding Module	Microphone Input Module USART Communication Module SD Card Module SD Card Interface Module
Behaviour-Hiding Module	Audio360 Engine Module Visualization Module
Software Decision Module	Audio Generation Module DOA Processor Module Audio Sampling Module Audio Spectral Leakage Prevention Module Audio Normalizer Module Audio Anomaly Detection Module Fast Fourier Transform Module Logging Module Fault Manager Module Mel Filter Module Discrete Cosine Transform Module Principle Component Analysis Module Linear Discriminant Analysis Module Classification Module Independent Component Analysis

Table 2: Module Hierarchy

6 Data Types

6.1 Generics

6.1.1 bool

Logical 0 or 1.

6.1.2 int8

8 bits signed integer ($\mathbb{Z}$).

6.1.3 int16

16 bits signed integer ($\mathbb{Z}$).

6.1.4 int32

32 bits signed integer ($\mathbb{Z}$).

6.1.5 int64

64 bits signed integer ($\mathbb{Z}$).

6.1.6 uint8

8 bits unsigned integer ($\mathbb{N}$).

6.1.7 uint16

16 bits unsigned integer ($\mathbb{N}$).

6.1.8 uint32

32 bits unsigned integer ($\mathbb{N}$).

6.1.9 uint64

64 bits unsigned integer ($\mathbb{N}$).

6.1.10 float32

32 bits floating point ($\mathbb{R}$).

6.1.11 float64

64 bits floating point ($\mathbb{R}$).

6.1.12 string

Character stream.

6.2 Enums

6.2.1 Audio360State

1. AudioClassificationProcess = 0: State when audio classification is running.
2. DirectionalAnalysisProcess = 1: State when directional analysis is running.
3. OutputProcess = 2: State when output processing is running.

6.2.2 Audio360Status

1. Uninitialized = 0: Audio360 Engine is not initialized.
2. Initialized = 1: Audio360 Engine is initialized, but not ready.
3. Ready = 2: Audio360 Engine ready for requests.
4. Running = 3: Audio360 Engine is running. Can not accept new requests.
5. Error = 4: Audio360 Engine is stuck at an unhandled error.

6.2.3 FaultState

1. NoFault = 0: No fault state.
2. MicrophoneXFault = 1: Fault with microphone X.
3. AudioClassificationFault = 2: Fault with audio classification.
4. directionalAnalsysisFault = 3: Fault with the directional analysis.

6.2.4 MicInputStatus

1. Idle = 0: Microphone input is idled.
2. Error = 1: Error in receiving microphone input.
3. Received = 2: Received microphone input.

6.2.5 AudioAnomalyStatus

1. None = 0: No audio anomalies.
2. Silent = 1: Received audio is silent.
3. Loud = 2: Received audio is loud.

6.3 Data Structures

6.3.1 FrequencyDomain

1. N [uint16]: The number of data points.
2. frequency [float32[]]: The frequency represented in Hz.
3. real [float32[]]: The real component of the frequency contribution.
4. img [float32[]]: The imaginary component of the frequency contribution.
5. magnitude [float32[]]: The magnitude of the frequency component.

6.3.2 CALLBACK_MIC_DATA

1. func_callback : ([MIC_INPUT_BUFFER_SIZE] of int16) $\rightarrow$ $\{0, 1\}$ Defines the callback function type for microphone data subscription.

6.3.3 MIC_INPUT_CALLBACK

1. pair of (id: $\mathbb{N}$, callback: CALLBACK_MIC_DATA)

6.3.4 Dictionary

1. For any type T_1, T_2 from generic data types, T_1 (key) maps to T_2 (value). Key := *set of* T_1.

6.3.5 List

1. For any type T from generic data types, a list is a collection of ordered objects of type T.

7 MIS of Microphone Input Module - M1

7.1 Module

This module handles the task of interfacing and receiving audio sample data from the microphone array connected to the microcontroller and providing a buffered stream to modules fetching the stream.

7.2 Uses

1. Serial Audio Interface (SAI) Driver - STM32 HAL Library [3]
2. Direct Memory Access (DMA) Driver - STM32 HAL Library [3]
3. Microphone Diagnostic Module
4. CALLBACK_MIC_DATA
5. SAI_HandleTypeDef - STM32 HAL Library [3]

7.3 Syntax

7.3.1 Exported Constants

- MIC_INPUT_BUFFER_SIZE = 4096. Traces to NFR3.2.
- MIC_INPUT_SAMPLE_RATE = 16000. Traces to NFR1.2.
- NUM_MICROPHONES = N_{mic}

7.3.2 Exported Access Programs

Name	In	Out	Exceptions
initMicInput	-	uint8	MicInputInitFailure
subscribeMicInput	callback: CALLBACK_MIC_DATA	subscriptionCB: MIC_INPUT_CALLBACK	MicInputSubscriptionFailure
unsubscribeMicInput	subscriptionID: $\mathbb{N}$	-	MicInputUnsubscriptionFailure
micInput_DMA_Rx	handle: SAI_HandleTypeDef*	-	-
micInput_diagnostic_error	errorCode: $\mathbb{N}$	-	-

7.4 Semantics

7.4.1 State Variables

- micInputStatus: MicInputStatus

- subscriptions: MIC_INPUT_CALLBACK[]
- nextSubscriptionID: $\mathbb{N}$

7.4.2 Environment Variables

- audioBuffer: sequence [MIC_INPUT_BUFFER_SIZE $\times$ (MIC_INPUT_SAMPLE_RATE $\times$ 2) $\times$ NUM_MICROPHONES] of $\mathbb{N}$

7.4.3 Assumptions

- Callback functions provided in subscribeMicInput are valid function pointers.

7.4.4 Access Routine Semantics

initMicInput():

- transition: micInputStatus := MIC_INPUT_IDLE $\wedge$ nextSubscriptionID := 0 $\wedge$ subscriptions := $\langle\ \rangle$ $\wedge$ ($\forall i \in \mathbb{N} | 0 \leq i <$ |audioBuffer|: audioBuffer[i] := 0) $\wedge$ Initialize SAI and DMA drivers $\wedge$ subscribeDiagnostics(micInput_diagnostic_error)
- output: micInputStatus
- exception: exc := MicInputInitFailure if SAI or DMA initialization fails.

subscribeMicInput(callback):

- transition: nextSubscriptionID := nextSubscriptionID + 1 $\wedge$ subscriptions := subscriptions $\|$ $\langle$ pair of (id: nextSubscriptionID, callback: callback) $\rangle$
- output: subscriptionCB := pair of (id: nextSubscriptionID, callback: callback)
- exception: exc := MicInputSubscriptionFailure if subscription fails for any reason.

unsubscribeMicInput(subscriptionID):

- transition: isValidCallback(subscriptionID) $\Rightarrow$
  subscriptions := (subscriptions - {cb: MIC_INPUT_CALLBACK | cb $\in$ subscriptions: cb.id $\equiv$ subscriptionID})
- output: None
- exception: exc := $\neg$ isValidCallback(subscriptionID) $\Rightarrow$ MicInputUnsubscriptionFailure

micInput_DMA_Rx(handle):

- transition: (handle.ErrorCode $\equiv$ HAL_SAI_ERROR_NONE) $\Rightarrow$ micInputStatus :=MIC_INPUT_RECEIVED $\wedge$ ($\forall$ cb: MIC_INPUT_CALLBACK $|$ cb $\in$ subscriptions : cb.callback(audioBuffer))
- output: None
- exception: None

micInput_diagnostic_error(errorCode):

- transition: (errorCode $\not\equiv$ 0) $\Rightarrow$ micInputStatus := MIC_INPUT_ERROR
- output: None
- exception: None

7.4.5 Local Functions

- isValidCallback: $\mathbb{N} \rightarrow$ boolean
  isValidCallback(id) $\equiv$ ($\exists$ cb : MIC_INPUT_CALLBACK $|$ cb $\in$ subscriptions : $cb.id \equiv id$)

8 MIS of USART Communication Module - M2

8.1 Module

This module handles USART communication between the microcontroller and an external device.

8.2 Uses

1. USART Driver - STM32 HAL Library [3]
2. USART_Init() - STM32 HAL Library [3]
3. UART_Transmit() - STM32 HAL Library [3]

8.3 Syntax

8.3.1 Exported Constants

- USART_BAUDRATE = 115200
- USART_TIMEOUT = 1000
- USART_BUFFER_SIZE = 256

8.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
initUSART	-	uint8	USARTInitFailure
transmitData	data: sequence of uint8	-	USARTTxFailure
receiveData	length: $\mathbb{N}$	data: sequence of uint8	USARTRxFailure

8.4 Semantics

8.4.1 State Variables

None.

8.4.2 Environment Variables

- usartBuffer: sequence [USART_BUFFER_SIZE] of uint8

8.4.3 Assumptions

None.

8.4.4 Access Routine Semantics

initUSART():

- transition: $(\forall i \in \mathbb{N} | 0 \leq i < |\text{usartBuffer}|: \text{usartBuffer}[i] := 0) \wedge$ USART_Init(USART_BAUD_RATE, ...).
- output: (Successful Initialization) $\Rightarrow$ 0 $\wedge$ $\neg$ (Successful Initialization) $\Rightarrow$ 1
- exception: exc := USARTInitFailure if USART initialization fails.

transmitData(data):

- transition: $(|data| \leq$ USART_BUFFER_SIZE) $\Rightarrow$ copy_array_data(data,usartBuffer, $|data| \leq) \wedge (|data| >$ USART_BUFFER_SIZE) $\Rightarrow$ copy_array_data(data,usartBuffer, USART_BUFFER_SIZE) $\wedge$ UART_Transmit(usartBuffer, min($|data|$, USART_BUFFER_SIZE), USART_TIMEOUT)
- output: None
- exception: exc := (UART_Transmit(...) $\equiv$ HAL_ERROR) $\Rightarrow$ USARTTxFailure

receiveData(length):

- transition: $(length \leq$ USART_BUFFER_SIZE) $\Rightarrow$ UART_Receive(usartBuffer, length, USART_TIMEOUT) $\wedge$ $(length >$ USART_BUFFER_SIZE) $\Rightarrow$ UART_Receive(usartBuffer, USART_BUFFER_SIZE, USART_TIMEOUT)
- output: data
- exception: exc := (UART_Receive(...) $\equiv$ HAL_ERROR) $\Rightarrow$ USARTRxFailure

8.5 Local Functions

- copy_array_data: sequence of $\mathbb{N}\times$ sequence of $\mathbb{N} \times \mathbb{N} \rightarrow$ void
  copy_array_data(src, dest, length) $\equiv$ $(\forall i \in \mathbb{N} | 0 \leq i \leq \text{length} : \text{dest}[i] := \text{src}[i])$

9 MIS of SD Card Interface Module - M3

9.1 Module

This module manages reading from and writing to an SD card connected to the microcontroller via the SPI interface.

9.2 Uses

1. SPI Driver - STM32 HAL Library [3]
2. FATFS Library - Chan's FatFs [4]

9.3 Syntax

9.3.1 Exported Constants

None.

9.3.2 Exported Access Programs

Name	In	Out	Exceptions
initSDCard	-	uint8	SDCardInitFailure
writeLog	data: string	-	SDCardWriteFailure

9.4 Semantics

9.4.1 State Variables

None.

9.4.2 Environment Variables

None.

9.4.3 Assumptions

None.

9.4.4 Access Routine Semantics

initSDCard():

- transition: HAL_SPI_Init(...) $\wedge$ f_mount(...)
- output: (Successful Initialization) $\Rightarrow 0 \wedge \neg$ (Successful Initialization) $\Rightarrow 1$

- exception: exc := (SPI $\vee$ FATFS initialization failure) $\Rightarrow$ SDCardInitFailure.

writeLog(data):

- transition: f_write(data) $\wedge$ f_sync()
- output: None
- exception: exc := ((f_write() $\not\equiv$ FR_OK) $\vee$ (f_sync() failure $\not\equiv$ FR_OK)) $\Rightarrow$ SDCard-WriteFailure

9.5 Local Functions

None.

10 MIS of Microphone Diagnostic Module - M4

10.1 Module

This module performs diagnostics on the microphone array to detect hardware errors.

10.2 Uses

- ADC Driver - STM32 HAL Library [3]

10.3 Syntax

10.3.1 Exported Constants

- MIC_DIAG_ERROR = 1

10.3.2 Exported Access Programs

Name	In	Out	Exceptions
initMicDiagnostics	-	uint8	MicDiagInitFailure
runMicDiagnostics	-	uint8	MicDiagFailure
subscribeDiagnostics	callback: $\mathbb{N} \rightarrow$ void	-	-

10.4 Semantics

10.4.1 State Variables

- diagStatus : $\mathbb{N}$
- callbacks: sequence of ($\mathbb{N} \rightarrow$ void)

10.4.2 Environment Variables

- ADC_Handle: ADC_HandleTypeDef
- adcValue: $\mathbb{R}$

10.4.3 Assumptions

None.

10.4.4 Access Routine Semantics

initMicDiagnostics():

- transition: diagStatus := 0 $\wedge$ ADC_Init(ADC_Handle).
- output: (Successful Initialization) $\Rightarrow$ 0 $\wedge$ $\neg$ (Successful Initialization) $\Rightarrow$ 1
- exception: exc := (ADC_Init(...) $\equiv$ HAL_ERROR) $\Rightarrow$ MicDiagInitFailure.

runMicDiagnostics():

- transition: adcValue = HAL_ADC_GetValue(ADC_Handle) $\wedge$ (map(adcValue, 0, 65535, 0, 3.3) $<$ 3.1) $\Rightarrow$ diagStatus := MIC_DIAG_ERROR $\wedge$ ($\forall$ cb: ($\mathbb{N} \rightarrow$ void) $|$ cb $\in$ callbacks : cb(diagStatus))
- output: diagStatus
- exception: exc := (ADC read failure) $\Rightarrow$ MicDiagFailure

subscribeDiagnostics(callback):

- transition: callbacks := callbacks $\|$ $\langle$ callback $\rangle$
- output: None
- exception: None

10.5 Local Functions

- map: $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$
  map(x, in_min, in_max, out_min, out_max) $\equiv$ (x - in_min) $\times$ (out_max - out_min + 1) / (in_max - in_min + 1) + out_min

11 MIS of Audio Generation Module - M5

11.1 Module

This module generates audio data by simulating room acoustics and generating synthetic microphone array data from a given audio source and position. This includes room response calculations and spatial audio propagation models.

11.2 Uses

1. Python Standard Libraries

11.3 Syntax

11.3.1 Exported Constants

None

11.3.2 Exported Access Programs

Name	In	Out	Exceptions
generateAudio	audioSource: float32[], position: float32[], outputFile: string	microphoneData[][]: [][float32]	AudioGenerationFailure

11.4 Semantics

11.4.1 State Variables

None.

11.4.2 Environment Variables

None.

11.4.3 Assumptions

The pyroomacoustics module is assumed to be working correctly (generating realistic and reliable audio data based on configuration parameters).

11.4.4 Access Routine Semantics

generateAudio():

- transition: None

- output: microphoneData
- exception: AudioGenerationFailure

11.4.5 Local Functions

- simulateRoom(room: pra.ShoeBox) → bool

12 MIS of Direction of Arrival (DOA) Processor Module - M6

12.1 Module

This module processes audio data to estimate the direction of arrival of a sound source. This includes frequency domain analysis, signal processing, and direction estimation algorithms.

12.2 Uses

1. Audio Generation Module
2. Audio Normalizer Module
3. Audio Spectral Leakage Prevention Module
4. Audio Anomaly Detection Module

12.3 Syntax

12.3.1 Exported Constants

None

12.3.2 Exported Access Programs

Name	In	Out	Exceptions
calculateDirection	audioData[][]: [][float32]	direction: float32	AudioProcessingFailure

12.4 Semantics

12.4.1 State Variables

None.

12.4.2 Environment Variables

None.

12.4.3 Assumptions

None.

12.4.4 Access Routine Semantics

calculateDirection():

- transition: None
- output: direction
- exception: AudioProcessingFailure

12.4.5 Local Functions

None.

13 MIS of Audio360 Engine - M7

13.1 Module

Orchestrates the overall audio processing by receiving raw input data and managing module communication.

13.2 Uses

1. Classification Module
2. DOA Processor Module
3. Fault Manager Module
4. Logging Module

13.3 Syntax

13.3.1 Exported Constants

None

13.3.2 Exported Access Programs

Name	In	Out	Exceptions
runProgram	-	-	ProgramStartFailure, ProgramRunTimeFailure

13.4 Semantics

13.4.1 State Variables

- state [Audio360State]: Determines the current state of the the Audio360 engine. Each state will run a specific module in used modules.

13.4.2 Environment Variables

- status [Audio360Status]: Status of the module. This encapsulates initialization, running, ready, or errored.

13.4.3 Assumptions

None

13.4.4 Access Routine Semantics

run():

- transition: The state machine in figure 1 outlines the transition of this module.

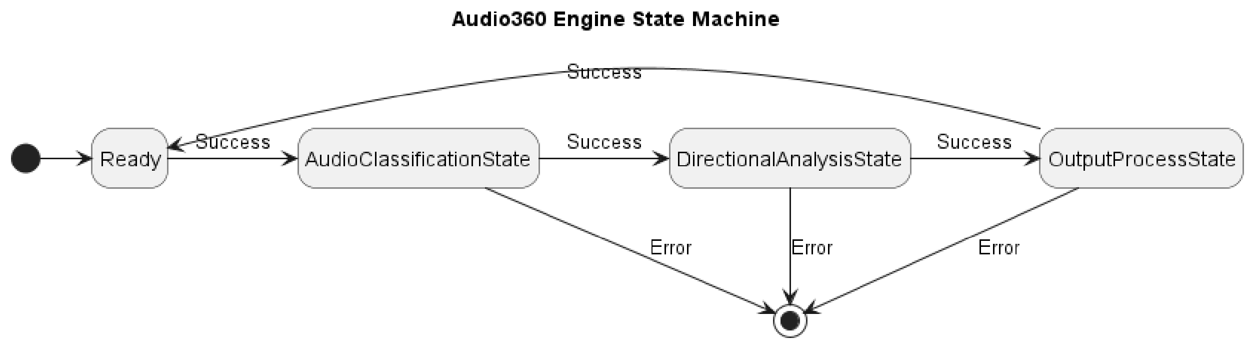

Figure 1: Internal state machine of Audio360 Engine module.

- output: None
- exception: ProgramStartFailure, ProgramRunTimeFailure

13.4.5 Local Functions

None

14 MIS of Audio Sampling Module - M8

14.1 Module

Provides sampling of audio data from a given source while preserving the order of individual samples.

14.2 Uses

- Microphone Input Module
- Microphone Diagnostic Module

14.3 Syntax

14.3.1 Exported Constants

- sampleWindowSize [uint16]: The number of samples in a window. The value is a power of 2.

14.3.2 Exported Access Programs

Name	In	Out	Exceptions
sample	inputSource: string	-	InputSourceError
getSampledData	-	sampleData	-

14.4 Semantics

14.4.1 State Variables

- sampledData [uint32[]]: A cyclic array of size sampleWindowSize that stored the sampled data.
- windowPosition [uint32]: The current reference index position of the array storing sampled data. This denotes the starting point of the cyclic array.

14.4.2 Environment Variables

None

14.4.3 Assumptions

None

14.4.4 Access Routine Semantics

sample():

- transition:
    1. $sampleData[windowPosition] := inputSource.newData$
    2. $((windowPosition + 1) \geq sampleWindowSize) \implies windowPosition := 0 \lor \neg((windowPosition + 1) \geq sampleWindowSize) \implies windowPosition := windowPosition + 1$
- output: None
- exception: InputSourceError

getSampledData():

- transition: None
- output: sampleData
- exception: None

14.4.5 Local Functions

None

15 MIS of Audio Spectral Leakage Prevention Module - M9

15.1 Module

Applying windowing on audio signal to reduce spectral leakage before frequency domain processing.

15.2 Uses

None

15.3 Syntax

15.3.1 Exported Constants

None

15.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
applyFilter	sampleWindow:float32[]	filteredWindow: float32[]	-

15.4 Semantics

15.4.1 State Variables

None

15.4.2 Environment Variables

None

15.4.3 Assumptions

None

15.4.4 Access Routine Semantics

applyFilter():

- transition: None
- output:

1. $(\forall\ i \mid 0 \leq i \leq N-1 :\ filteredWindow[i] := sampleWindow[i] * \sin(\frac{\pi * i}{N-1})^2)$ such that $N =$ size of sampleWindow.

- exception: None

15.4.5 Local Functions

None

16 MIS of Audio Normalizer Module - M10

16.1 Module

Processes audio signals to maintain consistent amplitude across different sources.

16.2 Uses

None

16.3 Syntax

16.3.1 Exported Constants

None

16.3.2 Exported Access Programs

Name	In	Out	Exceptions
normalize	sampleWindow: int32[], source: string	filteredWindow: float32[]	-

16.4 Semantics

16.4.1 State Variables

None

16.4.2 Environment Variables

None

16.4.3 Assumptions

Normalization factor is pre-determined and known for all given sources.

16.4.4 Access Routine Semantics

normalize():

- transition: None
- output:

1. $(\forall\ i \mid 0 \leq i \leq N-1 :\ filteredWindow[i] := \frac{sampleWindow[i]}{F_{source}})$
   such that $N =$ size of sampleWindow and F_{source} is normalization factor of a the sampleWindow source.

- exception: None

16.4.5 Local Functions

None

17 MIS of Audio Anomaly Detection Module - M11

17.1 Module

This module analyzes audio data to detect anomalies based on predefined patterns.

17.2 Uses

- Microphone Input Module

17.3 Syntax

17.3.1 Exported Constants

- AA_SIL_AMP_TIM_THRESH = 10
- AA_SIL_AMP_THRESH = 100
- AA_LOUD_AMP_THRESH = 60000

17.3.2 Exported Access Programs

Name	In	Out	Exceptions
initAnomalyDetector	-	uint8	AnomalyDetectorInitFailure

17.4 Semantics

17.4.1 State Variables

- aStat : AudioAnomalyStatus
- saCounter : $\mathbb{N}$

17.4.2 Environment Variables

- micInputCallback : CALLBACK_MIC_DATA

17.5 Assumptions

None.

17.5.1 Access Routine Semantics

initAnomalyDetector():

- transition: aStat = AA_NONE $\wedge$

  micInputCallback := cb_function(data: [MIC_INPUT_BUFFER_SIZE] of int16) $\rightarrow \{0, 1\}$
  cb_function(data) $\equiv$

  $$\Big(\text{(silent\_function(data)} \equiv 1) \Rightarrow \Big(\text{saCounter} := \text{saCounter} + 1 \wedge ((\text{saCounter} \equiv \text{AA\_SIL\_AMP\_TIM\_THRESH}) \Rightarrow \text{aStat} := \mathit{AA\_SILENT})\Big)\Big) \vee$$

  $$\Big((\neg(\text{silent\_function(data)} \equiv 1)) \Rightarrow (\text{saCounter} := 0)\Big) \vee$$

  $$\Big((\text{loud\_noise\_function(data)} \equiv 1) \Rightarrow (\text{aStat} := \mathit{AA\_LOUD})\Big) \vee$$

  $$\Big((\neg\,\text{silent\_function(data)} \wedge \neg\,\text{loud\_noise\_function(data)}) \Rightarrow (\text{aStat} := 0 \wedge \text{saCounter} := 0)\Big)$$

17.6 Local Functions

- silent_function: [MIC_INPUT_BUFFER_SIZE] of int16 $\rightarrow \{0, 1\}$
  silent_function(data) $\equiv ((+i|0 \leq i <$ MIC_INPUT_BUFFER_SIZE $\wedge$ data[i] $<$ AA_SIL_AMP_THRESH : 1) $\equiv$ MIC_INPUT_BUFFER_SIZE)

- loud_noise_function: [MIC_INPUT_BUFFER_SIZE] of int16 $\rightarrow \{0, 1\}$
  loud_noise_function(data) $\equiv ((+i|0 \leq i <$ MIC_INPUT_BUFFER_SIZE $\wedge$ data[i] $>$ AA_LOUD_AMP_THRESH : 1) $\equiv$ MIC_INPUT _BUFFER_SIZE)

18 MIS of Fast Fourier Transform - M12

18.1 Module

Computes the discrete Fourier transform of the input signal to obtain its frequency domain

18.2 Uses

- Audio Spectral Leakage Prevention Module

18.3 Syntax

18.3.1 Exported Constants

None

18.3.2 Exported Access Programs

Name	In	Out	Exceptions
signalToFrequency	signal: int32[]	frequency float32[]	-

18.4 Semantics

18.4.1 State Variables

None

18.4.2 Environment Variables

- inputsize [uint32]: The size of the signal input, as in number of samples. This is required to interface with hardware acceleration methods on the microcontroller.

18.4.3 Assumptions

None

18.4.4 Access Routine Semantics

signalToFrequency():

- transition: None
- output:
  1. $(\forall\ k \mid 0 \leq k \leq N-1 :\ frequency[k] := \sum_{n=0}^{N-1} x_n * e^{-2i\pi n \frac{k}{N}})$
     such that $N =$ size of input signal.
- exception: None

18.4.5 Local Functions

- createOutput(): returns a FrequencyDomain. This function extract features from the FFT and stores it in the data structure.

19 MIS of Logging Module - M13

19.1 Module

Provides centralized logging of data streams to designated output destinations for debugging and monitoring.

19.2 Uses

- USART Communication Module
- SD Card Interface Module

19.3 Syntax

19.3.1 Exported Constants

None

19.3.2 Exported Access Programs

Name	In	Out	Exceptions
log	text: string	-	-

19.4 Semantics

19.4.1 State Variables

None

19.4.2 Environment Variables

None

19.4.3 Assumptions

None

19.4.4 Access Routine Semantics

log():

- transition: None
- output: input text is logged to output source (SD card or USART port)
- exception: None

19.4.5 Local Functions

None

20 MIS of Fault Manager Module - M14

20.1 Module

Monitors system health and manages critical faults to maintain core functionality.

20.2 Uses

1. Microphone Diagnostic Module
2. Classification Module
3. DOA Processor Module
4. Audio Anomaly Detection Module

20.3 Syntax

20.3.1 Exported Constants

None

20.3.2 Exported Access Programs

Name	In	Out	Exceptions
runFaultAnalysis	-	-	-

20.4 Semantics

20.4.1 State Variables

- faultState [FaultState]: The current fault state of the overall software system.

20.4.2 Environment Variables

- faultState: This variable is shared with the display, to notify user of the internal status of the software system.

20.4.3 Assumptions

Fault Manager module itself does not run into any critical faults.

20.4.4 Access Routine Semantics

faultState():

- transition: update the value of faultState depending on the overall health of the software system.
- output: None
- exception: None

20.4.5 Local Functions

None

21 MIS of Mel Filter Module - M15

21.1 Module

Converts a Spectrogram into the equivalent Mel-Spectrogram by applying a bank of n mel-scaled triangular filters to each frequency frame.

21.2 Uses

None

21.3 Syntax

21.3.1 Exported Constants

None

21.3.2 Exported Access Programs

Name	In	Out	Exceptions
applyMelFilterBank	spectrogram: 2D real array	mel-spectrogram: 2D real array	invalidSpectrogramDimension

21.4 Semantics

21.4.1 State Variables

None

21.4.2 Environment Variables

None

21.4.3 Assumptions

- The input spectrogram is the output of the FFT Module across a constant buffer time. This is arranged as a 2D matrix where the rows represent time frames and columns represent frequency bins. This matrix can be represented as $S(t, k)$ where k is the number of frequency bins, and t is the number of time frames.

- The mel filterbank matrix is precomputed offline and stored as a read only constant within the system. It is computed offline using pre-determined information like Sampling Rate, FFT Size, maximum and minimum frequencies and number of mel filters. This matrix can be represented as $H(m, k)$, where m is the number of mel-filters, k is the number of frequency bins and $H(m, k)$ is a weight that tells you how much frequency bin k contributes to mel band m.

21.4.4 Access Routine Semantics

applyMelFilterBank():

- **transition:** None (function doesn't have states)
- **output:** Returns the equivalent mel-spectrogram, where each time frame is converted from linear-frequency bin representation to mel-scaled frequency bins. This is done by:
  1. Multiplying each spectrogram frame by the mel filterbank matrix.
  2. Summing weighted energy contributions per mel filter.

  $$E(t,m) = \sum_{k} S(t,k) \cdot H(m,k)$$

- **exception:** Invalid spectrogram dimension. If the dimension of the input spectrogram is not $S(t,k)$ and perhaps something like $S(t,p)$ where $p \neq k$, then the matrix multiplication would throw an invalid dimension error, which must be exposed from the module as invalid spectrogram dimension as well. This would only happens if the number of frequency bins in the computed FFT is not equal to the number of frequency bins in the pre-computed mel filterbank matrix.

21.4.5 Local Functions

None

22 MIS of Discrete Cosine Transform Module - M16

22.1 Module

Converts a mel-spectrogram into a 2D array containing the sequence of Mel Frequency Cepstral Coefficients (MFCC) vectors by applying the Discrete Cosine Transform (DCT) to each time frame. This reduces overlap between frequency bands that carry similar information and ensures each coefficient is uncorrelated to the other one. Pre-processing step that makes it easier to perform Principle Component Analysis later on.

22.2 Uses

None

22.3 Syntax

22.3.1 Exported Constants

None

22.3.2 Exported Access Programs

Name	In	Out	Exceptions
applyDCT	mel-spectrogram: 2D real array	mfcc: 2D real array	invalidDimensions

22.4 Semantics

22.4.1 State Variables

None

22.4.2 Environment Variables

None

22.4.3 Assumptions

- The input mel-spectrogram is the output of the Mel Filter Module. This input will be of size $E(t, m)$, where t is the number of time frames, and m is the number of mel-frequency bins.
- The DCT transform matrix $D(c, m)$ will be computed offline, where c is the desired number of MFCC coefficients for each time frame, and m is the number of mel-frequency bands. $D(c, m)$ represents the unique cosine wave that is associated with the

c^{th} coefficient and m^{th} band. A combination of these waves will be used to uniquely represent the spectrogram at time frame t.

$$D(c,m) = \cos\Big(\frac{\pi c}{M}(m-0.5)\Big)$$

- Only 13 MFCC coefficients will be calculated for each time frame t.

22.4.4 Access Routine Semantics

applyDCT():

- **transition:** None (function doesn't have states)
- **output:** Returns a matrix *mfcc(t,c)*, such that for each time frame t and each MFCC coefficient index c, the value of the matrix at $mfcc(t,c)$ can be computed as

$$mfcc(t,c) = \sum_{m=1}^{M} E(t,m) \cdot DCT(c,m)$$

  Where $E(t,m)$ represents the mel-spectrogram at time frame t and mel-frequency m. And $DCT(c,m)$ represents the DCT transform matrix that was computed offline.
- **exception:** Invalid spectrogram dimension. If the dimension of the input spectrogram is not $E(c,m)$ and perhaps something like $E(c,p)$ where $m \neq p$ then the matrix multiplication would throw an invalid dimension error, which must be exposed from the module as invalid spectrogram dimension as well. This would only happens if for some reason the number of mel-frequency bands returned by the Mel Filter Module is not the same as the number of mel-frequency bands in the DCT matrix computation.

22.4.5 Local Functions

None

23 MIS of Principle Component Analysis Module - M17

23.1 Module

Principle Component Analysis (PCA) highlights the features with the greatest variance from an input feature matrix. In this case, uses the MFCC feature vector for each time frame to find the direction of greatest variance. This helps with eliminating noise and unimportant information.

23.2 Uses

Non

23.3 Syntax

23.3.1 Exported Constants

None

23.3.2 Exported Access Programs

Name	In	Out	Exceptions
applyPCA	mfcc: 2D real array	pcaFeatures: 2D real array	invalidDimensions

23.4 Semantics

23.4.1 State Variables

None

23.4.2 Environment Variables

None

23.4.3 Assumptions

- The input MFCC matrix is the output of the Discrete Cosine Transform Module. This input will be of size $mfcc(t, c)$, where t is the number of time frames, and c is the number of coefficients.
- The PCA mean vector and PCA projection matrix will be derived offline during the system training and stored as read-only constants. The PCA mean vector will be of size

$(1, c)$, and will be computed as the average MFCC vectors across all training samples.

$$\bar{x} = \frac{1}{T} \sum_{t=1}^{T} x(t)$$

The PCA projection matrix will be of size (K, c), where K is the number of eigen-vectors with the maximum variance and c is the number of MFCC coefficients. These eigenvectors are extracted from the following covariance matrix.

$$C = \frac{1}{T} \sum_{t=1}^{T} (x(t) - \bar{x}) (x(t) - \bar{x})^T$$

23.4.4 Access Routine Semantics

applyPCA():

- **transition:** None (function doesn't have states)
- **output:** Returns a matrix *pca(t, K)*, where t represents the number of time frames and K represents the number of PCA components retained (proportional to the number of eigen-vectors). Essentially, each input mfcc matrix is projected onto the vectors of maximum variance, therefore extracting only the most important features of the MFCC based on training. This projection can be computed using the following matrix multiplication.

  $$\text{pca} = (mfcc - \mathbf{1}\bar{x}) P^T$$

  Where P represents the projection matrix, $\bar{x}$ represents the PCA mean vector, and $mfcc$ represents the full mfcc matrix.
- **exception:** Invalid mfcc dimension. If the dimension of the input mfcc matrix is not (t, c) and perhaps something like (t, p) where $c \neq p$ then the matrix multiplication would throw an invalid dimension error, which must be exposed from the module as invalid mfcc dimension as well. This would only happens if for some reason the number of coefficients returned by the Discrete Cosine Transform is not the same as the number of coefficients in the projection matrix.

23.4.5 Local Functions

None

24 MIS of Linear Discriminant Analysis Module - M18

24.1 Module

Classifies given feature vector into one of the predefined set of sound classes. This is module is optimized for class separation. The feature vector in this case is already narrowed down to the features with the most variance using the PCA module.

24.2 Uses

None

24.3 Syntax

24.3.1 Exported Constants

None

24.3.2 Exported Access Programs

Name	In	Out	Exceptions
applyLDA	pcaFeatures: 2D real array	classLabels: 1D real array	invalidDimensions

24.4 Semantics

24.4.1 State Variables

None

24.4.2 Environment Variables

None

24.4.3 Assumptions

- The input pcaFeatures is the output of the Principle Component Analysis Module. This input will be of size (t, K), where t is the number of time frames, and K is number of features selected with the highest variances from PCA.
- The LDA projection matrix, and class weight parameters are computed offline using labelled training data. They are stored as read-only constants in the system. The LDA projection matrix is of size $(K, C-1)$, where C is the number of classes, and K is the number of variance eigenvectors. In essence, this matrix maps the input feature space into the discriminant subspace where class separation is maximized.

24.4.4 Access Routine Semantics

applyLDA():

- **transition:** None (function doesn't have states)
- **output:**

  Returns a vector of class labels, one per time frame.

  Each feature vector from PCA is projected onto the discriminant space using the LDA projection matrix using the following multiplication.

  $$z(t,:) = pca(t,:) \cdot W_{\text{LDA}}$$

  Where $pca(t,:)$ represents the PCA eigenvectors at time t, and W_{LDA} represents the LDA projection matrix.

  Classification is then performed using linear discriminant functions for each class j, where w_j and b_j is the weight and bias matrices that were pre-computed offline during training for class j.

  $$g_j(t) = w_j^T z(t,:) + b_j$$

  The predicted class for that time frame is then assigned as the class with the maximum value.

  $$\text{classLabel}(t) = \arg\max_j g_j(t)$$

  The output of the module is then a vector of length t, representing the best classification for all time frames.

- **exception:** Invalid feature space dimension. If the dimension of the input feature matrix is not (t, K) and perhaps something like (t, p) where $K \neq p$ then the matrix multiplication would throw an invalid dimension error, which must be exposed from the module as invalid feature dimension as well. This would only happens if for some reason the number of eigenvectors returned by the PCA module is not the same as the number of eigen-vectors in the precomputed LDA matrix.

24.4.5 Local Functions

None

25 MIS of Classification Module - M19

25.1 Module

Orchestrates the overall audio classification flow by receiving signals in the frequency domain and encapsulating the complexity of the various modules involved.

25.2 Uses

1. Audio Sampling Module
2. Audio Normalizer Module
3. Fast Fourier Transform Module
4. Mel Filter Module
5. Discrete Cosine Transform Module
6. Principle Component Analysis Module
7. Linear Discrete Analysis Module

25.3 Syntax

25.3.1 Exported Constants

None

25.3.2 Exported Access Programs

Name	**In**	**Out**	**Exceptions**
classify	spectrogram: 2D real array	-	ClassificationStartFailure, ClassificationRunTimeFailure

25.4 Semantics

25.4.1 State Variables

- state[classificationState]: Stores the current classification. Will continuously be updated at runtime after receiving continuous audio signals.
- state[classificationStatus]: Stores the confidence of the classification that was done. Values for status include high, medium or low confidence.

25.4.2 Environment Variables

- state[status]: Encapsulates initialization, running, ready or errored status of the module.

25.4.3 Assumptions

None.

25.4.4 Access Routine Semantics

applyLDA():

- **transition:** State machine below outlines the transition of this module.

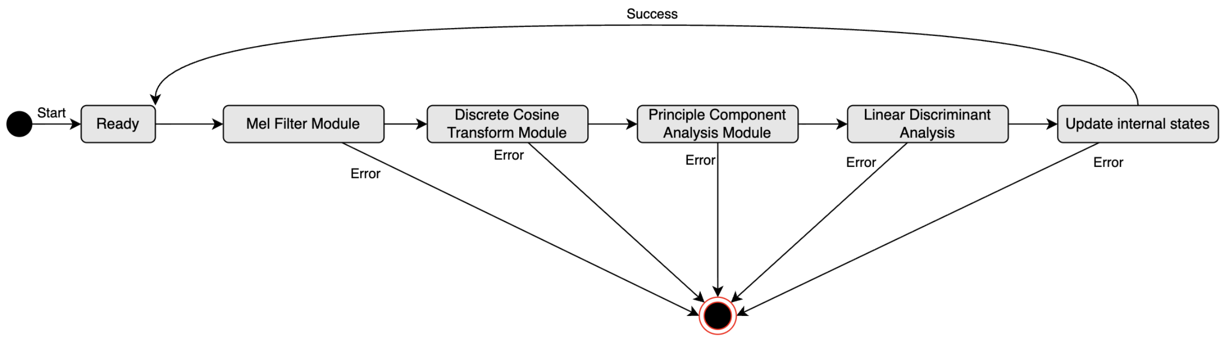
Success
Start
Ready
Mel Filter Module
Discrete Cosine Transform Module
Principle Component Analysis Module
Linear Discriminant Analysis
Update internal states
Error
Error
Error
Error
Error

Figure 2: Internal state machine of Classification module.

- **output:** None
- **exception:** ClassificationStartFailure, ClassificationRunTimeFailure

25.4.5 Local Functions

None

26 MIS of Visualization Module - M20

26.1 Module

This module formats and renders visual information on the smart glasses display. It receives classification labels and direction angles from the Audio360 Engine, formats this information according to display constraints and prioritization rules, and renders visual indicators on the smart glasses display. The module ensures non-intrusive presentation while maintaining a minimum update rate of 30 frames per second. It also handles display of error states and safety feature failure alerts.

26.2 Uses

1. Audio360 Engine Module
2. Fault Manager Module

26.3 Syntax

26.3.1 Exported Constants

- minUpdateRate [uint16]: The minimum display update rate in frames per second. The value is 30 FPS as specified by NFR8.1.

26.3.2 Exported Access Programs

Name	In	Out	Exceptions
updateDisplay	classificationLabel: string, directionAngle: float32, priority: uint8	-	DisplayUpdateFailure, InvalidInputError
displayError	errorType: string, errorMessage: string	-	DisplayUpdateFailure

26.4 Semantics

26.4.1 State Variables

- currentDisplayState [dict]: Stores the current state of the display, including active visual indicators, their positions, and priorities. This state is continuously updated at runtime as new classification and direction data is received.

- displayQueue [list]: A priority queue that stores pending display updates. Items are ordered by priority to ensure safety-critical information is displayed first, as required by NFR6.1.

26.4.2 Environment Variables

- display [Display Hardware]: The smart glasses display hardware interface. This encapsulates the physical display device and its driver interface, allowing the visualization format to be modified independently of the underlying display technology.

26.4.3 Assumptions

- The display hardware is properly initialized and functional.
- The Audio360 Engine provides valid classification labels and direction angles in radians within the range $[0, 2\pi)$.
- Priority values are non-negative integers, with lower values indicating higher priority (safety-critical information has priority 0).

26.4.4 Access Routine Semantics

updateDisplay():

- **transition:** The module receives classification and direction data, applies prioritization logic to determine if the information should be displayed (based on NFR6.1), formats the information according to display constraints and non-intrusive presentation requirements (NFR6.2), and updates the display state. The visual indicator is rendered on the smart glasses display at the appropriate position corresponding to the direction angle. If multiple sources are present, only the highest priority source is displayed.
- **output:** None
- **exception:** DisplayUpdateFailure if the display hardware fails to update, or InvalidInputError if the direction angle is outside the valid range $[0, 2\pi)$ or if the classification label is empty.

displayError():

- **transition:** The module formats and displays an error message on the smart glasses display to alert users when core safety features such as direction determination or classification fail, as required by FR6.2. Error messages are displayed with high priority to ensure visibility.
- **output:** None
- **exception:** DisplayUpdateFailure if the display hardware fails to update the error message.

26.4.5 Local Functions

- formatDisplayData(classificationLabel: string, directionAngle: float32) → dict: Formats the classification label and direction angle into a display-ready data structure, including position calculations for directional indicators and text labels.
- applyPrioritization(displayQueue: list) → dict: Selects the highest priority item from the display queue to be rendered, ensuring safety-critical information is displayed first as required by NFR6.1.
- renderNonIntrusive(displayData: dict) → bool: Renders visual indicators in a non-intrusive manner, minimizing visual obstruction so users can safely perform external activities, as required by NFR6.2.

27 MIS of Independent Component Analysis (ICA) Module - M21

27.1 Module

This module performs blind source separation on multi-channel audio signals to separate mixed audio signals into individual independent source components. The module accepts multi-channel audio signals from the microphone array and outputs separated audio streams, each representing an individual sound source. This enables improved classification and direction estimation accuracy when multiple sound sources are present simultaneously. The module is conditionally invoked by the Audio360 Engine when multiple sources are detected, allowing the system to operate with or without this capability.

27.2 Uses

1. Audio360 Engine Module
2. Audio Sampling Module

27.3 Syntax

27.3.1 Exported Constants

None

27.3.2 Exported Access Programs

Name	In	Out	Exceptions
separateSources	mixedAudio: [][float32]	separatedSources: [][float32]	ICASeparationFailure, InvalidInputError

27.4 Semantics

27.4.1 State Variables

- separationModel [dict]: Stores the ICA separation model parameters and learned mixing matrix. This state may be updated during the separation process to adapt to the input signal characteristics.

27.4.2 Environment Variables

None

27.4.3 Assumptions

- The input audio signals contain multiple independent sound sources that can be separated using ICA.
- The number of microphones is at least equal to the number of sound sources to be separated.
- The sound sources are statistically independent and non-Gaussian.
- The mixing process is linear and instantaneous (no significant delays between sources reaching different microphones).

27.4.4 Access Routine Semantics

separateSources():

- **transition:** The module applies Independent Component Analysis (e.g., FastICA algorithm) to the input mixed audio signals from multiple microphones. The algorithm estimates the mixing matrix and separates the mixed signals into independent source components. The separation model state may be updated during this process. Each output stream represents a separated audio source that can be further processed by classification and direction estimation modules.
- **output:** An array of separated audio source streams, where each stream contains the audio data for one independent sound source. The number of output streams corresponds to the number of detected independent sources.
- **exception:** ICASeparationFailure if the ICA algorithm fails to converge or cannot successfully separate the sources. InvalidInputError if the input audio data is invalid, has incorrect dimensions, or does not meet the assumptions required for ICA separation.

27.4.5 Local Functions

- estimateMixingMatrix(mixedAudio: [][float32]) → dict: Estimates the mixing matrix that describes how the independent sources are combined in the mixed signals. This is a core step in the ICA algorithm.
- applySeparation(mixedAudio: [][float32], mixingMatrix: dict) → [][float32]: Applies the inverse of the estimated mixing matrix to separate the mixed signals into independent source components.
- checkConvergence(previousModel: dict, currentModel: dict) → bool: Checks if the ICA algorithm has converged by comparing the current model parameters with the previous iteration's parameters.

References

[1] D. M. Hoffman and P. A. Strooper, *Software Design, Automated Testing, and Maintenance: A Practical Approach*. New York, NY, USA: International Thomson Computer Press, 1995. [Online]. Available: http://citeseer.ist.psu.edu/428727.html

[2] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

[3] STMicroelectronics, “User manual,” 2023.

[4] Chan, “Fatfs - generic fat filesystem module,” 2023.

Appendix — Reflection

1. What went well while writing this deliverable?

   Sathurshan: The system was decomposed into modules that was small enough. This allowed the team to easily split up the work without having much dependencies on each other. Furthermore, writing the MIS has helped the team further align with design decisions and formally document.

   Kalp: I think that developing the PoC (Proof of Concept) was a great way to get a better understanding of the system and the modules that are involved. Though maybe not ideal in other scenarios, getting a small headstart on the design implementation helped us really easily scope out what the modules on the system should be.

   Nirmal: Since the classification POC was already created, I feel like it was a lot easier to decompose the classification feature into distinct features. Furthermore, since the POC was created earlier, a lot of the research was already completed, which means I knew what equations I needed for each module.

   Jay: I think the part that went well was how aligned the team already was before writing the document. Because we had been discussing the system and reviewing each module ahead of time, putting everything together felt more organized. The MIS ended up reinforcing our understanding and helped us clearly lay out how each part of the system will work.

   Omar: Our team knocked it out of the park by using formal math to define each module's functionality whenever possible. This was especially helpful for reducing workload looking ahead.

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

   Omar: The major pain point was the timing of this deliverable. Although it is done well, it required too much time and effort while balancing other deliverables and courses. We tried to resolve this by delegating sections to each team member based on their expertise.

   Sathurshan: The main pain point was the deliverable deadline as the team was asked to finish the first revision in a week while preparing for the proof of concept of demo. The team expressed the concerns with the professor, and fortunately received an extension allowing us to put more effort into this document.

   Kalp: I think that the main pain point was just timing. With the VnV plan due shortly prior to this deliverable, and the team PoC implementation being due a week after this deliverable, there was really not much time to write out this document, especially given the length and detail that it goes into. This was slightly mitigated by the provided extension and the PoC implementation, but it was still a challenge to get everything done in time.

Nirmal: Although I mentioned I had a good idea of which modules are needed for classification, modularizing which equation goes in what module and in what order was a difficult and time consuming task. For example the POC was done as a python implementation, and as such the logic with coming up with the LDA projection matrix was encapsulated into python libraries. For this document, I had to reverse engineer how exactly this is done using equations.

Jay: The biggest pain point for me was keeping everything consistent across the sections, especially since multiple people were writing at the same time while also balancing work on the PoC. It was a bit overwhelming to juggle both. We resolved this by coordinating early and reviewing each other's sections so the document stayed cohesive. The extension also gave us some breathing room to fix issues we noticed.

3. Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. your peers, stakeholders, potential users)? For those that were not, why, and where did they come from?

   Omar: Many design decisions stemmed from past experience working on embedded projects and large system designs. The approach to effectively break down larger tasks such as audio classification into smaller modules (e.g., FFT, Mel Filter, DCT, PCA, LDA) was inspired by prior discussion with our capstone supervisor.

   Sathurshan: The supervisor mentioned the difficulty of getting access to hardware that will meet our software specification. As a result, we designed the system such that the software is portable and can take input from different sources. This ensures that the capstone project can still be successful in the case we are not able to find the right hardware within our budget.

   Kalp: A lot of the hardware and software design decisions that we made was based from speaking to the supervisor and the team. Our supervisor would be able to give better insight on what would and wouldn't work for the scope of what we're trying to accomplish. We selected the pipeline design of audio to processing to output with the specific hardware, but the specific algorithm (ex. LDA for classification) or the specific hardware came through our supervisors insights.

   Nirmal: The decision of using PCA and LDA for classification stems from asking our supervisor (MVM) for input on an approach that has low computation. Originally, I was using a SVM for classifying audio sources, but our supervisor mentioned this isn't really necessary, and might be overkill.

   Jay: Most of the high-level decisions came from discussions with the supervisor, especially around what is realistic for the hardware we have access to. Hearing their perspective helped steer us toward a design that is flexible and doesn't rely too heavily on one specific setup. Some of the lower-level decisions, like how we break up certain processes, came from our own team discussions and figuring out what would make implementation more manageable.

4. While creating the design doc, what parts of your other documents (e.g. requirements, hazard analysis, etc), it any, needed to be changed, and why?

   Omar: While writing the design document, some parts of the SRS needed to be updated to add labels and to link module decomposition to the SRS requirements.

   Sathurshan: SRS needed to be updated. It was updated because part of writing this document required reviewing the SRS. From this, I have found parts of the requirements that can be improved based on recent better understanding of the system.

   Kalp: The SRS document was updated to reflect the changes that we made to the system. A lot of the requirements changes revolved around the hardware and software design decisions that we made (ex. choosing to separate a process into multiple modules leads to more specific requirements).

   Nirmal: For the classification specific modules, no parts of any other documents needed to be changed, since the other documents never mentioned any exact implementation details. Which is what this document is focused on.

   Jay: Similar to the others, I found that parts of the SRS needed updates. While detailing the modules in this document, it became clear that a few requirements could be phrased more precisely or split into smaller pieces based on the updated understanding of the system. So the changes mostly came from wanting the requirements to better match the actual design.

5. What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)

   Omar: The main limitation in our case is the hardware constraints. Given unlimited resources, we would be able to design custom audio hardware that meets all of our software requirements without any guess work. We would also be able to design PCBs that are optimized for reducing trace lengths between the microphones and the microcontroller, improving signal integrity.

   Sathurshan: The limitation of the solution is compute and memory power. Given further time, the team would be able to design a system that is more low level and is optimized in accomplishing the tasks that are required.

   Kalp: I think the biggest limitation that we can see right now is the hardware limitations. The software that we're using for this project is good enough to handle the tasks that we're trying to accomplish within the software requirements, but the hardware limitations are what are holding us back slightly. With just better hardware (at the expense of higher cost), a lot of the issues we're dealing with could be mitigated.

   Nirmal: Given unlimited resources, we could have a more robust classification approach. The current approach of using PCA and LDA comes from not having computation power to use resources like SVM or neural networks taking in the full audio input.

Jay: The biggest limitation I see is tied to performance and the hardware we're working with. If we had more resources, we could definitely explore more powerful hardware and push the system to handle more complex processing or higher-resolution data. With more time and computing power, we could also refine the software to be more efficient and include features that aren't feasible right now.

6. Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)

   Omar: During the research phase, we considered using many different hardware platforms including Raspberry Pi, Texas Instruments and Arduino. We settled on the STM32 platform due to its inclusion of DSP libraries and audio peripherals. While the Texas Instruments controllers have great documentation and software support, they are specialized for motor control applications which means they lack the real-time audio peripherals required for the project. On the software side, we considered using more complex machine learning algorithms for audio classification, but ultimately chose not to due to simplicity and compute resource constraints

   Sathurshan: Personally, there was not a lot of time to think about other designs. The team has been implementing the software before this document was created since the proof of concept is one week after this document is due. Thus, the team already considered and analyzed high level designs months ago and is too far back to document them.

   Kalp: I know for the software side, specifically the audio classification, we considered using a neural network for the classification. This was a potential solution, but we ultimately decided against it because of the hardware limitations. The neural network would require a lot of memory and processing power, which is not available on the hardware that we're using. For this reason, we decided to use the LDA for classification because it is a simple algorithm that is easy to implement and understand, and a lot less computationally expensive than a neural network.

   Nirmal: Another design solution we considered for classification was using SVM. Although that seemed to work really well, compared to the PCA and LDA approach, this approach would firstly take a lot of storage on the microcontroller which we didn't have. And also involved running full blackbox models. The approach now just involves doing matrix multiplications which is what the microcontroller is optimized for.

   Jay: One alternative we discussed early on was using more advanced machine learning methods for classification, like a small neural network or more complex models. They might have improved accuracy, but they also would have required more memory and processing power than we realistically have. The chosen design gives us a simpler, reliable system that fits within our constraints, which made it the more practical option for the project.