

An Investigation into the Application of Distributed Endpoint Processing to 3D Immersive Audio Rendering

A thesis submitted in fulfilment of the
requirements for the degree of

MASTERS IN SCIENCE

of

RHODES UNIVERSITY

by

ROBIN SEAN DEVONPORT

February 2019

Abstract

3D immersive audio rendering systems use algorithms that accurately render the perceived location of an audio source within a space using multiple sets of loudspeakers. The algorithms used by these renderers can become very processor intensive as the number of tracks and loudspeakers being used increases. The ImmerGo spatial audio system provides a solution to this by using distributed endpoint processing. ImmerGo uses a client-server architecture that controls multiple smaller endpoints distributed on an Ethernet Audio Video Bridging (AVB) network. Each endpoint contains an audio mixer which is dedicated to mixing audio to a smaller subset of speakers according to commands from ImmerGo.

The ImmerGo system performs its rendering calculations within the host computer and then mixes the audio at each endpoint. The work in this thesis identified that the processing load put on the host computer due to these calculations may be mitigated by offloading the rendering algorithm calculations to the endpoints as well. In order to prove this for a number of different algorithms, the ImmerGo system was modified to include three new algorithms as well as the one it currently uses. Each algorithm then had its processing offloaded to the endpoints at three successive steps of distribution. Tests were performed to determine the limitations of each step, and how the host processor and endpoint processors were affected by distributing the calculations. Listening tests were also performed to ensure that each newly introduced algorithm was implemented correctly.

Acknowledgements

I would like to extend a warm thanks to my supervisor, Richard Foss, for all his attentive guidance throughout this research. His ability to provide the necessary tools, funding, feedback and constant enthusiasm has been invaluable to me. As well as this, a huge thank you to miniDSP in Hong Kong for providing the funding to cover my living expenses throughout this research, as well as providing the gear used for the development without which the project would not have been possible. In particular, Tony Rouget and Dennis Wong from miniDSP for all their hard work and constant feedback that really pushed this project forward. Thanks to the Rhodes University Computer Science (CS) Department that provided the space for me to conduct this research comfortably and the Telkom Center of Excellence (COE) who provide so much for CS post-graduates and Alfredo Terzoli for arranging funding to visit some amazing conferences.

Some more personal thanks to the people close to me. Brent Shaw and Sean Pennefather from the CS Department for their willingness to share their extensive knowledge that seemed to know no bounds. A massive thank you to my mother and brother, Leslie and Nico Devonport. Their constant loving support has provided me with a solid foundation from which I can grow. Massimo Meneghetti for always being a friend I can talk to, though we're far away, our consistent chats over WhatsApp have helped keep me moving, questioning and inspired. Another massive thank you to my special friend, Shannon Hansen, who has done so much to keep me healthy, sustained and happy – you are one of kind and I'm blessed to have you care about me the way you do.

There are so many more people I'd like to thank that if I were to write them all down, it would take more than a good portion of time for them to read through this to get to their name, so I hope to see them soon and be able to thank them personally.

Table of Contents

Abstract	i
Acknowledgements.....	ii
Table of Contents	iii
List of Figures	ix
List of Tables.....	xii
List of Code Listings.....	xiii
Abbreviations.....	xvi
Chapter 1: Introduction.....	1
1.1 3D Immersive Audio.....	1
1.1.1 Representations used by 3D Spatial Audio	2
1.1.1.1 Object-based Audio (OBA).....	2
1.1.1.2 Scene-based Audio (SBA).....	3
1.2 Distributed Digital Signal Processing (DSP) in an Immersive Context.....	4
1.3 The “Big Box” Approach.....	4
1.4 The Need for Distributed Endpoint Approach	5
1.5 ImmerGo with Ethernet Audio Video Bridging	6
1.6 Hypothesis.....	7
Chapter 2: Ethernet AVB and ImmerGo – A platform for distributed immersive audio.....	9
2.1 Ethernet AVB.....	9
2.1.1 Overview of Ethernet AVB.....	9
2.1.2 Ethernet AVB Devices	10
2.1.3 Device Clock Synchronization, Traffic Shaping and Stream Reservation	12
2.1.4 The IEEE 1722 Protocol	13
2.1.5 The IEEE 1722.1 Protocol	14
2.2 ImmerGo	16
2.2.1 ImmerGo Overview.....	16
2.2.2 The NW.js Environment.....	18
2.2.3 The ImmerGo Client.....	18
2.2.4 ImmerGo Renderer Definition Models.....	20
2.2.4.1 Room Configuration Model.....	20

2.2.4.2 ImmerGo Track Object Model.....	21
2.2.5 ImmerGo's Rendering Algorithm – Distance Based Amplitude Panning.....	22
2.2.5.1 DBAP Theory.....	22
2.2.5.2 Spatial Blur	24
2.2.5.3 Weighting sources to chosen loudspeakers.....	25
2.2.5.4 DBAP Algorithm.....	25
2.2.6 Panning Sources in ImmerGo.....	26
2.2.6.1 Receiving Track Updates from the Client	27
2.2.6.2 MIDI Synchronization from the DAW	28
2.2.6.3 Recording Track Object Updates as Automation.....	29
2.2.6.4 Automation Playback.....	30
2.2.7 ImmerGo Speaker Endpoint Processors.....	31
2.2.8 Controlling the Speaker Endpoint with ImmerGo.....	32
2.3 Summary.....	33
Chapter 3: Speaker Endpoint Processors	34
3.1 XMOS Processor Architecture	35
3.1.1 XMOS tasks.....	36
3.1.2 XMOS Pin Allocation	37
3.1.3 XMOS Inter-task Communication.....	38
3.1.4 XMOS Analysis Tools.....	39
3.2 ImmerGo's XMOS Processor.....	39
3.2.1 The Audio Path.....	40
3.2.2 The ImmerGo Mixer Matrix.....	42
3.2.3 Controlling the XMOS Mixer Matrix.....	44
3.3 Summary.....	45
Chapter 4: Modifications to ImmerGo	47
4.1 Modifying the ImmerGo Host to Send Audio Definition Models	47
4.1.1 The Audio Definition Model Transmission Template.....	48
4.1.2 Sending the ImmerGo Room Definition Model to Endpoints.....	48
4.1.3 Sending the ImmerGo Track Model to Endpoints.....	50
4.2 Introduction to the Spatial Audio Renderer task.....	52
4.2.1 The Spatial Audio Renderer (SAR).....	52

4.2.1.1 Functions within the Spatial Audio Renderer.....	53
4.2.1.2 The Spatial Audio Renderer Structure.....	53
4.2.1.3 Handling ImmerGo Commands.....	55
4.2.1.3.1 Controlling the XMOS mixer matrix.....	55
4.2.1.3.2 Unpacking the Speaker Configuration.....	56
4.2.1.3.3 Unpacking Track Sources.....	57
4.2.2 DBAP within the SAR.....	57
4.3 DBAP with the SHARC DSP Mixer Matrix	59
4.3.1 SHARC DSP Control.....	60
4.3.1.1 The SHARC Mixer matrix Parameters.....	60
4.3.1.2 Setting SHARC DSP Mixer matrix Crosspoints.....	61
4.4 Endpoint Clock Synchronization	63
4.5 Summary.....	64
Chapter 5: Vector Base Amplitude Panning	65
5.1 The Theory Behind VBAP	65
5.1.1 Stereophonic Panning.....	65
5.1.2 Two-Dimensional VBAP.....	66
5.1.3 Three-dimensional VBAP	68
5.1.4 VBAP and the Convex Hull.....	69
5.1.5 VBAP Algorithm	70
5.1.6 Multiple Direction Amplitude Panning (MDAP).....	70
5.2 VBAP Implementation	71
5.2.1 VBAP in NWJS.....	71
5.2.1.1 Calculating VBAP configuration	74
5.2.1.2 Performing VBAP within NWJS.....	76
5.2.1.3 Performing MDAP in NWJS	77
5.2.2 VBAP on the XMOS processor	79
5.2.2.1 Sending the VBAP configuration to the XMOS Processor.....	80
5.2.2.2 Performing the VBAP calculation within the XMOS.....	82
5.3 Summary.....	83
Chapter 6: Wave Field Synthesis.....	85
6.1 Theory of Wave Field Synthesis	85

6.1.1 Derivation of Loudspeaker Driving Signals.....	87
6.1.2 Virtual Source Classification.....	89
6.1.2.1 Plane Wave Source.....	90
6.1.2.2 Non-focused Source.....	90
6.1.2.3 Focused Source.....	91
6.1.2.4 Generalized WFS Focusing Operator	92
6.2 Known Problems with Wave Field Synthesis	92
6.2.1.1 Spatial Aliasing.....	92
6.2.1.2 Truncation Effect.....	94
6.3 WFS within ImmerGo.....	95
6.3.1 Modification to the XMOS Mixer matrix	96
6.3.2 Modifications to the SAR task.....	99
6.3.3 The WFS library and ImmerGo Modifications.....	100
6.3.4 WFS Panning in ImmerGo	102
6.3.4.1 Calculating WFS for a Source.....	103
6.3.4.2 Calculating the Tapering Window.....	104
6.3.4.3 WFS within the SAR	105
6.3.4.4 Mixing on the SHARC.....	107
6.3.4.4.1 Setting SHARC DSP Filters.....	108
6.4 Summary.....	109
Chapter 7: Ambisonics.....	110
7.1 Ambisonics Theory	110
7.1.1 Ambisonics Overview	111
7.1.1.1 The Spherical Harmonic Functions	111
7.1.1.2 Ambisonics Playback.....	112
7.1.2 Encoding an Ambisonic Soundfield.....	113
7.1.2.1 First Order	114
7.1.2.2 Higher Orders.....	115
7.1.3 Decoding an Ambisonic Soundfield.....	115
7.1.4 Ambisonic Formats.....	117
7.1.4.1 FuMa format	118
7.1.4.2 AmbiX format.....	118

7.1.4.3 Two Big Ears format	119
7.2 Ambisonics in ImmerGo.....	119
7.2.1 Ambisonics Library and ImmerGo Modifications.....	120
7.2.2 Ambisonics Decoding within the SAR	122
7.2.2.1 Setting the Mixer Matrix.....	122
7.2.2.1.1 Setting the SHARC Mixer Matrix.....	124
7.3 Summary.....	124
Chapter 8: Testing and Results	125
8.1 Listening Tests.....	125
8.1.1 DBAP and VBAP Listening Tests	126
8.1.1.1 DBAP Listening Test.....	128
8.1.1.2 VBAP Listening Test.....	129
8.1.2 WFS Listening Test	130
8.1.3 Ambisonics Listening Tests.....	132
8.2 Algorithm Time Tests.....	133
8.2.1 NWJS Time Profiling.....	133
8.2.1.1 Time Taken to Perform DBAP within the NWJS Application	133
8.2.1.2 Time Taken to Perform VBAP within the NWJS Application.....	134
8.2.1.3 Time Taken to Perform WFS within the NWJS Application	135
8.2.2 XMOS Time Profiling.....	136
8.2.2.1 XTA Results for Step 1 of Distribution	136
8.2.2.2 XTA Results for Step 2 and 3 of Distribution.....	137
8.2.2.2.1 XTA Timings for Endpoint Distribution of DBAP	137
8.2.2.2.2 XTA Timings for Endpoint Distribution of VBAP.....	139
8.2.2.2.3 XTA Timings for Endpoint Distribution of WFS.....	140
8.3 Host Processor Load Tests.....	140
8.3.1 Host CPU Load Results for DBAP.....	141
8.3.2 Host CPU Load Results for VBAP.....	142
8.3.3 Host CPU Loads for WFS	144
8.3.4 Host CPU Loads for Ambisonic Decoding.....	145
8.4 Discussion of Results	146
Chapter 9: Conclusion	148

9.1 ImmerGo as a Framework for Distributed Endpoint Processing.....	148
9.2 Project Work.....	149
9.2.1 Implementing the Spatial Audio Algorithms.....	149
9.2.2 Results of the Implementation.....	151
9.3 The Novelty of Extending the Distributed Endpoint Approach.....	153
9.4 Future Work.....	153
Appendix A	160
A1 Solving the Covariance Factor	160
A2 Sending Mixer Updates to Endpoints.....	160
A3 Sending ImmerGo Room Definition Model.....	162
A4 Sending ImmerGo Track Object Model.....	165
A5 Unpacking the Speaker Configuration	167
A6 Unpacking the Track Object Model	168
A7 Updating the SHARC Mixer Matrix Crosspoints.....	168
Appendix B.....	171
B1 Solving for the VBAP configuration	171
B2 Squeezing Vectors for MDAP.....	173
B3 Sending the VBAP configuration	176
B4 Receiving the VBAP Configuration within the SAR.....	178
B5 Performing VBAP within the SAR.....	179
Appendix C	181
C1 Sending the Gains and Delays Values to the Endpoint.....	181
C2 Calculating Delay and Attenuation Values with the WFS.js Library	182
C3 Calculating and Sending the Tapering Window	185
Appendix D	186
D1 Calculating the Ambisonic Decoder.....	186

List of Figures

Fig. 1: The distributed “big box” approach.....	5
Fig. 2: The distributed endpoint approach.....	6
Fig. 3: A simple AVB network.....	11
Fig. 4: The Apple Macintosh virtual AVB entity setup.....	11
Fig. 5: The Apple Macintosh native AVDECC controller connection matrix.....	12
Fig. 6: A typical ImmerGo setup. The red dotted lines show control messages being sent to the mixer matrix. The green dotted lines show audio being sent from the DAW.....	17
Fig. 7: The Immergo UI.....	19
Fig. 8: Geometry for DBAP.....	22
Fig. 9: Diagram showing DBAP setup and panning mechanism.....	27
Fig. 10: Receiving an MTC frame.....	29
Fig. 11: Recording automation.....	30
Fig. 12: Playing automation.....	31
Fig. 13: The SPK-4P active speaker.....	32
Fig. 14: Sending endpoint mixer updates from ImmerGo.....	33
Fig. 15: SPK-4P internal configuration.....	34
Fig. 16: XMOS XCORE-200 processor architecture.....	35
Fig. 17: Tiles and tasks.....	37
Fig. 18: ImmerGo XMOS AVB firmware with Mixer Matrix.....	40
Fig. 19: Path of audio through the Audio Buffer Manager task.....	41
Fig. 20: Pulling samples from the FIFO and passing to the I ² S master.....	41
Fig. 21: Attenuate mixer matrix.....	42
Fig. 22: Controlling the XMOS mixer matrix.....	45
Fig. 23: Speaker Configuration with their associated MAC address and unique Speaker IDs.....	48
Fig. 24: Sending the Room Definition Model to the Endpoint processor.....	50
Fig. 25: Track queue updates for every interval.....	51
Fig. 26: Sending track object updates to the processor from the ImmerGo server.....	51
Fig. 27: SPK-4P XMOS AVB firmware layout with modifications in highlighted in red.....	52
Fig. 28: Interfaces between the SAR and other tasks.....	54
Fig. 29: Routing the descriptor indices to functions within SAR using the <i>i_1722_1_control</i> interface.....	55
Fig. 30: Unpacking the speaker configuration into the SAR.....	56
Fig. 31: Unpacking the track source for DBAP calculation within the SAR.....	57
Fig. 32: DBAP algorithm within the XMOS processor.....	58
Fig. 33: Audio path and control path from the host computer to SHARC DSP.....	59
Fig. 34: The MIMO matrix found in the SHARC processor.....	60
Fig. 35: Address schema for the processing blocks within the SPK-4P SHARC mixer matrix.....	61
Fig. 36: Q1.15 representation.....	61
Fig. 37: Mixer matrix crosspoint parameter.....	62
Fig. 38: Setting crosspoints 2-5.....	63

Fig. 39: Setting crosspoints 2-3.....	63
Fig. 40: Setting the SHARC mixer with mixer values from DBAP.	64
Fig. 41: Typical stereophonic setup.	66
Fig. 42: VBAP formulation for two dimensions.	67
Fig. 43: VBAP formulation for three dimensions.....	68
Fig. 44: Convex-hull of a 15.0 speaker array.	69
Fig. 45: MDAP spread sources. Spread angle is 60° with 3 rings containing 20 sources.....	71
Fig. 46: Diagram showing functions exposed from to the ImmerGo server from the VBAP library.....	72
Fig. 47: ImmerGo VBAP client.....	73
Fig. 48: Checking Speaker radii is in bounds.	74
Fig. 49: Solving the VBAP configuration.....	75
Fig. 50: Discarding triangles.	76
Fig. 51: Panning with VBAP in ImmerGo.....	77
Fig. 52: Panning with MDAP in ImmerGo.....	78
Fig. 53: Creating secondary sources	79
Fig. 54: Squeezing secondary sources	79
Fig. 55: Sending the VBAP configuration from the ImmerGo server.	80
Fig. 56: Receiving the VBAP configuration within the SAR.....	81
Fig. 57: Loudspeaker selection.....	82
Fig. 58: Conditions of the Kirchhoff-Helmholtz integral.	87
Fig. 59: Geometry for a point source behind the array.....	89
Fig. 60: Spherical non-focused source, plane wave source and focused source.....	90
Fig. 61: Simulation of a non-focused source. Generated by wfs-visualizer.	91
Fig. 62: Simulation of a focused source. Generated by wfs-visualizer.....	92
Fig. 63: Aliasing frequency.....	93
Fig. 64: Spatial aliasing that reduces as loudspeaker spacing gets narrower. Generated by wfs-visualizer....	94
Fig. 65: Hann tapering window. Generated by wfs-visualizer.....	95
Fig. 66: ImmerGo WFS Client.	96
Fig. 67: Delay and attenuate mixer matrix.....	97
Fig. 68: Delay buffer for a non-focused source.....	98
Fig. 69: Delay buffer for a focused source.....	98
Fig. 70: The WFS.js module.....	101
Fig. 71: Prepanning setup with WFS.js.	102
Fig. 72: Panning with the WFS library.....	102
Fig. 73: Solving WFS gains and delays.	104
Fig. 74: Hann window with a width of 0.38.	105
Fig. 75: Hann window with a width of 0.61.	105
Fig. 76: Sending tapering window updates to endpoints.....	105
Fig. 77: Setting the listener position within the SAR.....	106
Fig. 78: Solving WFS within the SAR.....	107
Fig. 79: Setting the SHARC mixer with WFS values.....	108
Fig. 80: Mid/Side recording technique.....	111

Fig. 81: Hierarchy of Spherical Harmonics and associated B-format	112
Fig. 82: Assuming the loudspeakers are at each vertex, any of these layouts could be decoded to.	113
Fig. 83: Encoding process of a mono signal to first order Ambisonics.	114
Fig. 84: Decoding a B-format signal to multiple speakers.....	116
Fig. 85: Decoder functions within each speaker's Ambisonic decoder.....	116
Fig. 86: FuMa channel ordering	118
Fig. 87: ACN channel ordering	119
Fig. 88: Solving the Ambisonics decoder for each speaker.....	121
Fig. 89: Sending Ambisonic decoders to endpoints. From the ImmerGo server.....	121
Fig. 90: Parsing the decoder values into the SAR.....	123
Fig. 91: Setting the endpoint mixer matrix according to decoder mixer values.....	123
Fig. 92: Speaker Layout for DBAP and VBAP.....	126
Fig. 93: Camera above the hemispherical array.....	126
Fig. 94: Duvets covering the hemispherical array.....	127
Fig. 95: The RT60 analysis given by REW.....	127
Fig. 96: DBAP listening test source locations.....	128
Fig. 97: Distribution of incorrect and correct responses for DBAP listening tests.....	129
Fig. 98: VBAP listening test source locations.....	129
Fig. 99: Distribution of incorrect and correct responses for VBAP listening tests.	130
Fig. 100: WFS speaker array.....	130
Fig. 101: WFS listening test source locations.....	131
Fig. 102: Distribution of incorrect and correct responses for WFS listening tests.....	131
Fig. 103: Ambisonics listening test source locations.	132
Fig. 104: Distribution of incorrect and correct responses for Ambisonics listening tests.....	132
Fig. 105: Time taken to perform DBAP within the NWJS application.	134
Fig. 106: Time taken to perform VBAP within the NWJS application.	135
Fig. 107: Time taken to perform WFS within the NWJS application.	135
Fig. 108: XTA showing time taken to perform attenuation based mixing within XMOS according to mixer parameters received from the host.....	136
Fig. 109: XTA showing time taken to perform attenuation-delay based mixing within XMOS according to mixer parameters received from the host.....	137
Fig. 110: XTA time analysis of the DBAP algorithm with the second step of distribution.	138
Fig. 111: DBAP times for step 3 of distribution.	138
Fig. 112: XTA time analysis of the VBAP algorithm with the second step of distribution.	139
Fig. 113: XMOS timing for VBAP at varying sources and speaker sizes.....	139
Fig. 114: Host CPU loads for DBAP.....	142
Fig. 115: Host CPU loads for DBAP with Reaper.....	142
Fig. 116: Host CPU loads for VBAP.....	143
Fig. 117: Host CPU loads for VBAP with Reaper.....	143
Fig. 118: Host CPU loads for ImmerGo WFS.....	144
Fig. 119: Host CPU loads for ImmerGo WFS with Reaper.....	144
Fig. 120: CPU load exerted by Reaper with higher orders of Ambisonic decoding.....	145

List of Tables

Table 1: Iterations of distribution.....	8
Table 2: The common AVTPDU layout.....	13
Table 3: The common AECPDU layout.....	15
Table 4: The AECPDU command layout.....	15
Table 5: Room Configuration Model.....	20
Table 6: Loudspeaker model.....	20
Table 7: ImmerGo Audio Object Model.....	21
Table 8: ImmerGo's Automation Track Node.....	29
Table 9: AECP packet structure for the spatial audio renderer.....	48
Table 10: AECP SET_MIXER packet holding the Speaker Configuration.....	49
Table 11: AECP SET_MIXER descriptor indices for DBAP operations within the SAR.....	53
Table 12: AECP SET_MIXER Descriptor Indices for the VBAP algorithm in the SAR.....	79
Table 13: AECP SET_MIXER Descriptor Indices for the operations within the SAR.....	100
Table 14: Loudspeaker model with modifications for WFS.....	101
Table 15: SN3D Ambisonic encoding functions up to third order with FuMa weightings.....	115
Table 16: Ambisonic decoder in the Loudspeaker Model.....	120
Table 17: AECP SET_MIXER Descriptor Indices for Ambisonics decoding within the SAR.....	122
Table 18: AECP Set Ambisonic Decoder packet template.....	122
Table 19: WFS times for XMOS calculations and XMOS mixing.....	140
Table 20: WFS times for XMOS calculations and mixing within the SHARC.....	140

List of Code Listings

Listing 1: Module linking in Javascript.....	18
Listing 2: The room configuration XML.	21
Listing 3: Receiving track object updates within the server.....	27
Listing 4: The DBAP algorithm and updating the speaker mix levels within <i>updateSpeakerMixLevels</i>	28
Listing 5: Device architecture structure contained within the D4U-miniDSP2x4.xn file.....	36
Listing 6: Port declaration within xC.	37
Listing 7: Channels between two tasks within xC.	38
Listing 8: Interface between two tasks within xC.	39
Listing 9: Function header for the Audio Buffer Manager task.....	43
Listing 10: I ² S line settings.....	43
Listing 11: XMOS mixer matrix variables within the Audio Buffer Manager task.....	43
Listing 12: XMOS attenuation-only mixer matrix in the Audio Buffer Manager task.....	44
Listing 13: Internal interface used to update the XMOS mixer matrix.....	44
Listing 14: Passing the mixer matrix values from the Application task to the Audio Buffer Manager.....	45
Listing 15: Receiving the mixer matrix updates.....	45
Listing 16: Spatial Audio Renderer declaration.	53
Listing 17: The control interface between the application task and SAR.....	55
Listing 18: Updating the XMOS mixer matrix gain values.....	56
Listing 19: XMOS project with the SHARC mixer matrix.....	59
Listing 20: Code that converts floating point into Q1.15 representation.....	62
Listing 21: Module exports within the VBAP.js library.....	73
Listing 22: VBAP library global variables.....	73
Listing 23: Determining which convex hull triplets to discard.....	75
Listing 24: The VBAP calculation.....	77
Listing 25: Code to squeeze vectors.....	79
Listing 26: VBAP configuration variables introduced to the SAR.	79
Listing 27: New functions added to the <i>avb_1722_1_control</i> internal interface.....	80
Listing 28: XMOS delay summation mixer matrix in the Audio Buffer Manager task.....	97
Listing 29: Receiving XMOS mixer gain and delay updates from SAR in the Audio Buffer Manager task..	99
Listing 30: Variables added to the SAR task.....	99
Listing 31: Modifications to the <i>avb_1722_1_control</i> internal interface.....	100
Listing 32: Function to load filter coefficients	109
Listing A1: Solving for the covariance factor.....	160
Listing A2: <i>SendMixLevels</i> function which updates all the endpoint mixer parameters.....	160
Listing A3: The <i>SET_MIXER</i> packet structure within the ImmerGo source code.....	161
Listing A4: Sending AVDECC packets to an endpoint processor.	161
Listing A5: <i>sendSpkConftoXMOS</i> called after all the endpoint devices in XML have been discovered and connected.....	162
Listing A6: <i>sendSpkConftoXMOS</i> function.....	162

Listing A7: Packetizing the flattened speaker configuration.....	163
Listing A8: Speaker Configuration packet template.....	164
Listing A9: <i>sendSpkConftoXMOS</i> called after all the endpoint devices in XML have been discovered and connected to.....	165
Listing A10: The tracks for update array.....	165
Listing A11: Adding track IDs for updating	165
Listing A12: Sending source updates to the speaker endpoints.....	165
Listing A13: Creating source packet data.....	166
Listing A14: The source packet template.....	167
Listing A15: Sending the source packet.....	167
Listing A16: The server interface that receives the speaker configuration into the endpoints.....	168
Listing A17: Unpacking a track object before rendering.....	168
Listing A18: Setting a single matrix crosspoint.....	169
Listing A19: Function to set matrix crosspoints for a single source.	169
Listing A20: Function to load filter coefficients	170
Listing A21: The <i>DSP_AddrParaLoad</i> function.....	170
Listing B1: Determining the radius of each speaker in the ImmerGo server.	171
Listing B2: Calling <i>SetMesh</i> from the VBAP library.....	171
Listing B3: Checking each radius is within 5cm within the <i>checkAllValuesWithinBound</i> function.....	171
Listing B4: The <i>SetMesh</i> function.....	172
Listing B5: The <i>FindLsTriplets</i> function.....	172
Listing B6: Determining which convex hull triplets to discard.	173
Listing B7: Determining convex hull inverted loudspeaker matrices.	173
Listing B8: <i>getSpreadSrcDirs</i> function.....	175
Listing B9: Function called to send the VBAP configuration to the endpoint processors.	176
Listing B10: Sending the VBAP configuration to each amp.....	176
Listing B11: The VBAP configuration template packet.....	176
Listing B12: Sending the VBAP configuration template packet.....	178
Listing B13: <i>sendNoFacestoXMOS</i> function.....	178
Listing B14: Routing the VBAP configuration to the SAR within the XMOS.....	178
Listing B15: Parsing the VBAP configuration into the XMOS	179
Listing B16: Perform VBAP calculation in SAR.....	180
Listing C1: Calculating the WFS panning values for all speakers within the WFS.js library.....	181
Listing C2: Sending the gains and delay values to the endpoints.....	182
Listing C3: Calculating the WFS panning values for all speakers within the WFS.js library.....	182
Listing C4: Calculating a speaker gain value according to a source.....	183
Listing C5: Calculating the delay value according to a source.....	184
Listing C6: Calculating the WFS panning values for all speakers within the WFS.js library.....	184
Listing C7: Calculating new tapering windows.....	185
Listing C8: Solving for tapering weights.....	185
Listing C9: Tapering according to speaker index.....	185
Listing D1: Calling the <i>getAmbisonicsDecoder</i> function.....	186

Listing D2: The <i>getAmbisonicsDecoder</i> function.	186
Listing D3: Converting cartesian coordinates into spherical.	187
Listing D4: The <i>getAmbisonicsDecoder</i> function.	188

Abbreviations

CBA – Channel-based Audio

OBA – Object-based Audio

SBA – Scene-based Audio

DBAP – Distance Based Amplitude Panning

VBAP – Vector Base Amplitude Panning

WFS – Wave Field Synthesis

DAW – Digital Audio Workstation

MTC – MIDI Time Code

Ethernet AVB – Ethernet Audio Video Bridging

AVTP – Audio Video Transport Protocol

AVDECC – Audio Video Discovery, Enumeration, Connection Management and Control protocol

API – Application Program Interface

Chapter 1

Introduction

Since the advent of modern spatial recording, sound designers and composers have been using directional sound to create sonic spaces. Be it recreating scenes from history in the cinema house, generating new and unimaginable environments within virtual reality headsets, or even just listening to a concerto recording with headphones; the ability to have a listener completely immersed within the environment and its audio content is of utmost importance to the audio designer.

There are many different reasons for creating 3D spatial audio content and as such, many different configurations for realizing spatial audio. For instance, a virtual reality context may be better suited to headphone-based reproduction. A sound art installation may have a very irregular layout of speakers to reproduce the audio, while a movie house has very specific positions for the speakers. Enabling spatial audio in these different environments becomes quite a challenge, but with available techniques and methods, there is ample opportunity for this to be achieved.

This document aims to give insight into some of the more popular spatial audio rendering techniques, as well as describe a novel implementation of them. This implementation uses distributed processing and audio networks in an attempt to overcome some of the challenges that arise when creating larger immersive loudspeaker setups. The main challenges are,

- Complicated system setup with many devices, cables and audio interfaces.
- Large processing overheads.
- Difficulty when interfacing between the various audio software systems, devices and algorithms that make up an immersive audio system.

The rest of this chapter will introduce the reader to the concept of 3D immersive audio and discuss its different representations. Then, a brief outline of distributed processing within the context of 3D Immersive Audio will be given, with reference to some currently available systems. This will provide the necessary context to introduce the problem statement and motivation for the implementation described in this thesis. There is a brief introduction to the application framework that is used for the project is provided followed by how this framework was enhanced so to implement a novel distributed solution for 3D immersive audio.

1.1 3D Immersive Audio

Immersive audio (also referred to as 3D spatial audio or immersive sound) is different from other monophonic, stereophonic and surround playback techniques. As the name suggests, its intention is to immerse, or envelop, a listener using a multitude of loudspeakers, or in some cases, headphones. In this thesis, 3D immersive audio will refer to using multiple loudspeakers within a 3D environment. This differs from surround sound even though surround sound techniques use multiple speakers. In the case of

immersive sound, an algorithmic method is used to localize multiple sound sources to speakers within an environment. Immersive sound algorithms make it easier to both create and correctly playback the audio scene to an unspecified layout of loudspeakers. In contrast, a mix engineer creates surround sound mixes consisting of finite audio channels driving speakers within specific configurations. A surround sound mix requires that the audio is distributed in a multichannel format that can only be listened to correctly on standardized loudspeaker configurations. Whilst there are ways to encode to different loudspeaker configurations at the playback stage, it still requires the engineer to create the audio channels that feed the loudspeakers directly. This approach has been aptly named channel-based audio (CBA).

In order to create flexible listening systems that can be personalized to a user's listening environment, the environment must be modelled according to a more abstract notion of objects. Objects being associated with the physical location of track sources, loudspeakers, loudspeaker materials and room materials. This means that the 3D audio mixing process may be split into two parts,

1. Creating spatial metadata associated with audio sources within a spatial audio scene – the authoring stage.
2. Using a renderer to generate the appropriate loudspeaker signals according to source and room metadata – the playback stage.

In order to accomplish this, object-based audio (OBA) and scene-based audio (SBA) was developed as a means for a user to create a 3D 'audio scene' that both describes and produces the spatial information and directivity of audio tracks, whilst abstracting away from the act of manually generating the loudspeaker signals. The representation of this audio scene is no different from that of game objects in a video game. It gives a mix engineer the freedom to author the audio scene using a more intuitive representation of 3D space and passes the responsibility of generating appropriate loudspeaker signals for a particular configuration to an automated mixing renderer at the playback stage.

3D immersive audio is now being adopted in many different systems around the world. A large amount of public research and open source projects has pushed not only large commercial entities to create their own systems, but also many other smaller developers, to create their own do it yourself systems. There are various algorithms that are used to render 3D immersive content and, due to the nature of the algorithms, some may suit some applications more than the other. As alluded to at the beginning of this introduction, these applications can range from virtual reality, cinema, art and museum installations and music production and performance.

1.1.1 Representations used by 3D Spatial Audio

1.1.1.1 Object-based Audio (OBA)

Object-based mixing has been designed specifically for spatial audio and intends to simplify the mixing, distribution and rendering of spatial audio content (Bleidt, Borsum, Fuchs, & Weiss, 2014). Borrowing from game development concepts, each audio track is seen as an object with positional metadata that can be used by a spatial audio renderer to mix the audio appropriately to speaker outputs. Various parameters may be contained in the metadata that are used to describe the audio object's position, volume, distance

and spread within the mix. This metadata may also be associated with timestamps that allow these parameters to change over time in order to compose a spatial audio mix. A user would typically use a graphical interface to place audio objects on a 3D canvas that then have their metadata recorded.

As mentioned at the beginning of this chapter, object-based audio mixing is distinguished from typical channel-based audio mixing in that it abstracts the spatial information of audio tracks from the physical mixing process. This provides a handle for a user to think about the object's location rather than the mix levels required to place the object in the space (Barco, 2015). In contrast, if channel-based mixes are to be perceived correctly they need to be played back on an audio system very similar to the one that was used to mix it. Thus, these mixes are restricted to specific channel counts and need to be remixed and remastered for different speaker configurations (such as 5.1, 7.1, 11.1 etc). Object-based audio differs from the channel-based approach as it relies on the renderer to use track metadata to calculate the speaker signal levels at playback time. This offers more flexibility when deploying spatial audio content as only the positional information of track objects need to be recorded. The renderer can then render the audio according to the loudspeaker configuration. The signals feeding each loudspeaker channel may also be recorded should the user want a channel-based mixdown for the loudspeaker array. Also, important to note is these two approaches are often combined to provide accurate object-based localization for specific objects and using channel-based ambience. This is done to help minimize loads put on the object-based renderer.

A standardised object-based audio specification was first introduced by the European Broadcasting Union (EBU) within their "TECH 3364" as an "Audio Definition Model (ADM) - Metadata Specification" (European Broadcasting Union, 2014), which has recently been accompanied by the EBU ADM Renderer (EAR) open specification that describes a method of rendering the ADM (European Broadcast Union, 2018). This has been followed by the MPEG-H 3D audio standard which is directed at broadcasting and distribution of immersive audio content (Herre, Hilpert, Kuntz, & Plogsties, 2015) (MPEG-H, 2015) as well as a loudness measurement specification for multichannel audio (International Telecommunications Union, 2017). There is also the Society of Motion Picture and Television Engineers (SMPTE) working group which have developed the Multi-Dimensional Audio (MDA) object-based audio specification for similar reasoning (SMPTE, 2018) (SMPTE, 2018). Along with these open specifications, there are also many other proprietary versions of OBA that have been developed and used in software developed by larger companies' such as Dolby Atmos ¹, DTS:X ² and Auro3D AuroMax ³. Dolby has also implemented an audio format known as AC-4 which allows for OBA to be encoded and distributed alongside CBA (Dolby Laboratories, 2015). AC-4 is recognized as the official standard within the European Telecommunications Standards Institute (ETSI) and has been adopted by Digital Video Broadcasting (DVB) (ETSI, 2015) (DVB, 2018).

1.1.1.2 Scene-based Audio (SBA)

SBA is a hybridization of the object and channel-based approaches. The main concept of SBA is to encode all the audio objects into a finite number of channels that may then be distributed and decoded to a non-

¹ Doby Atmos - <https://www.dolby.com/us/en/technologies/home/dolby-atmos.html>

² DTS:X Creator Suite - <https://dts.com/dtsx>

³ Auro3D - <https://www.auro-3d.com/system/>

defined speaker configuration at the playback stage. The reduction of an entire audio scene into a known number of components significantly reduces the bandwidth and processing that would otherwise be required when distributing and rendering OBA on a large scale. This makes SBA a likely candidate for future developments with household 3D immersive audio and browser-based media.

Currently, the only rendering algorithm that has this feature is Ambisonics, which is also known as Higher Order Ambisonics (HOA) (Shivappa, Morrell, Sen, Peters, & Salehin, 2016). SBA implemented as HOA is included alongside CBA and OBA representations within the EBU ADM, the MPEG-H 3D Audio standard and the SMPTE audio bitstream (European Broadcasting Union, 2014) (MPEG-H, 2015) (SMPTE, 2018).

1.2 Distributed Digital Signal Processing (DSP) in an Immersive Context

The concept of distributed audio processing discussed in this thesis is not new and a number of distributed DSP products such as the UAD satellite ⁴ are available. These products have a dedicated external processor, which can process the audio externally by being inserted into a DSP signal chain similar to a plugin. Once processed, the audio is then routed back into the signal chain; not unlike an insert on a traditional analogue audio mixer.

Adapting this to immersive audio, systems that render 3D immersive audio often need to run in tandem with other audio applications running on the CPU such as a Digital Audio Workstation (DAW). The algorithms used for the rendering are required to handle mathematical calculations prior to actually processing the audio – with all this needing to happen in real-time. The algorithms are processor intensive, and even more so as these systems become larger. This makes 3D immersive audio a good candidate for distributing processing and there are a number of distributed spatial audio systems. The next chapter describes the typical approach that has been used by a number of available spatial audio systems.

1.3 The “Big Box” Approach

There are many spatial audio systems which have adopted what can be called “the big box” approach. The big box approach is a type of distributed processing, except it requires computer workstations with operating systems attached to external audio interfaces that provide the spatial audio rendering capability. A typical distributed setup may be similar to what is shown in **Fig. 1**. The renderer is generally controlled by a single controller that sends both clock synchronization messages and renderer parameters over a network to each big box. The multichannel playback requires that a multichannel audio interface is connected to these large processors, whereby each channel is fed from the processor through the interface to its respective loudspeaker in the 3D speaker array.

⁴ <https://www.uaudio.com/uad-accelerators/uad2-satellite-thunderbolt.html>

A variety of open source systems that use this approach ^{5 6}. There are a few problems that can be highlighted when using this method,

1. It does not use a standard audio transport protocol.
2. Control methods are not interoperable between devices and operating systems.
3. Audio Clock Synchronization methods can be unreliable.
4. Processing power in each big box may not be fully utilized.
5. Scaling to large speaker arrays becomes more expensive.

For instance, systems like the HKU Game Of Life WFSCollider evade the problem of streaming multichannel audio by copying the audio stems to each big box ⁷. This results in a complicated, expensive setup prior to localizing the audio and is not typically wanted by performing artists looking to adopt spatial audio. The key problems for them being the lack of mobility and setup complexity.

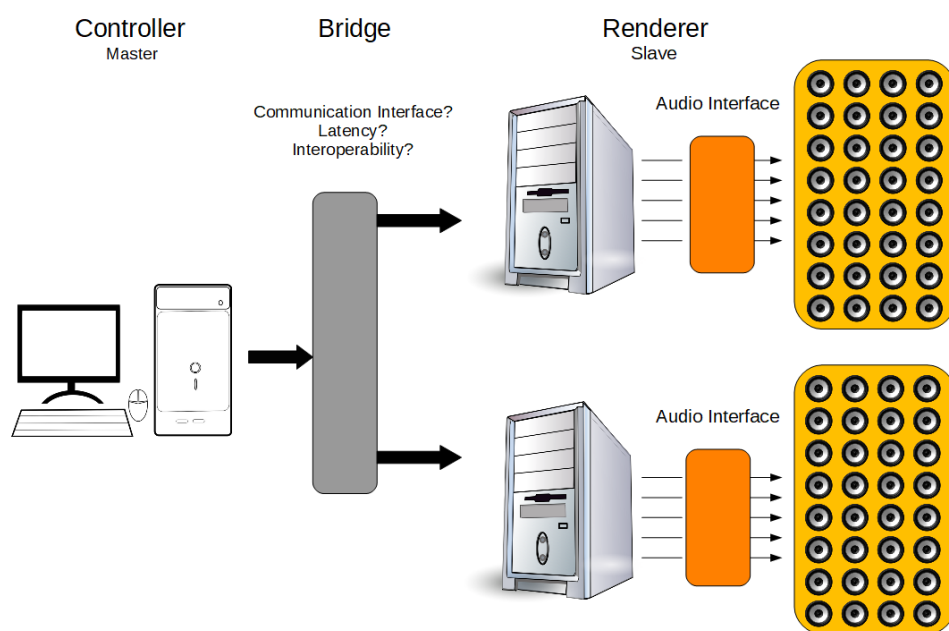


Fig. 1: The distributed “big box” approach.

1.4 The Need for Distributed Endpoint Approach

The problems that are presented with the above approach may be overcome by splitting the processing power into small dedicated processing nodes, with each node performing the arithmetic and processing for a small number of speakers. This thesis introduces this concept as the “distributed endpoint approach”.

⁵ <http://forumnet.ircam.fr/product/panoramix-en/>

⁶ <http://www.jamoma.org/>

⁷ <https://github.com/GameOfLife/WFSCollider>

This approach makes use of an open and standardized audio network technology, and dedicated microcontrollers attached to speakers that can communicate on the network. The use of this technology addresses the problems of synchronization, audio transport, control and inefficient processor usage that has been highlighted with the big box approach. Also, since each endpoint processor is dedicated to powering a small subset of speakers, the spatial audio system can be scaled up incrementally whilst still providing the user with the benefit of a plug and play environment. For example, there is no need for additional audio interface installation.

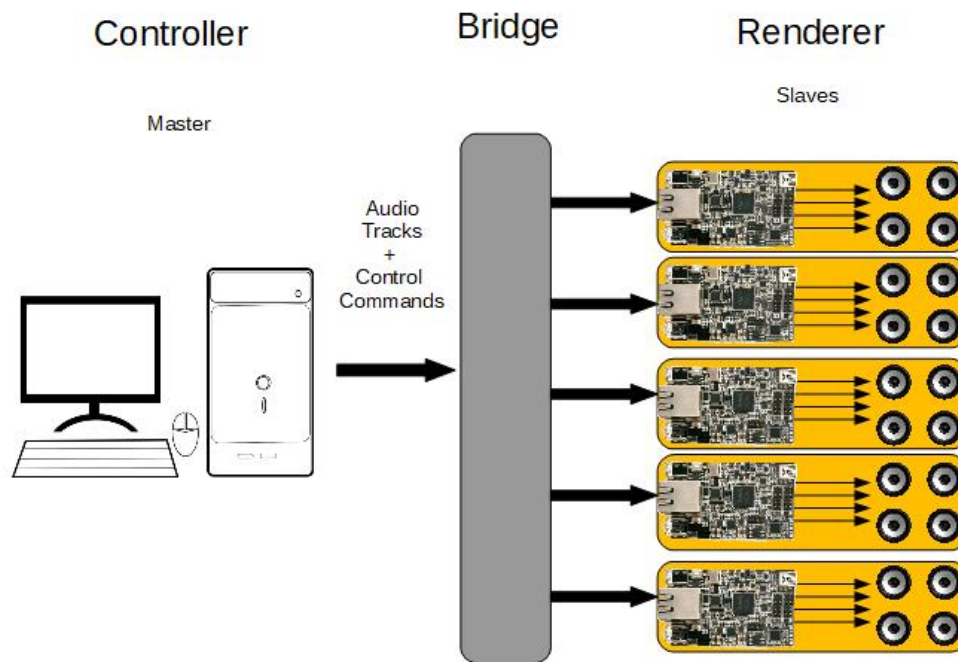


Fig. 2: The distributed endpoint approach.

Fig. 2 shows a basic distributed DSP approach. Distributed DSP endpoint processors situated on a network are controlled using commands sent from the host computer whilst also receiving audio from the host. There are three main issues that need to be addressed when distributing audio processes:

1. Real-time audio transport from host to processors.
2. Synchronization of processor clocks.
3. Remote processor control.

Provided a standard Ethernet audio protocol is used, these problems can be avoided whilst also providing the communication protocols to have each endpoint identified and connected when they are plugged into the network.

1.5 ImmerGo with Ethernet Audio Video Bridging – The potential for distribution to speaker endpoints

A type of distributed endpoint spatial audio system is the ImmerGo spatial audio system that has been developed at Rhodes University (Foss & Rouget, 2015). It is a client-server application that gives users the ability to render OBA content in real-time using a mobile device graphical user interface (GUI). It runs as a lightweight standalone desktop application that controls multiple network endpoint audio processors attached to loudspeakers. Each endpoint contains both an XMOS and SHARC processor that perform controllable audio mixing operations on audio signals received from the network. ImmerGo controls the endpoint mixing according to values output by its spatial audio rendering algorithm within the host machine. This mechanism allows for audio tracks to be localized with the desired location to the speaker configuration whilst pushing the audio mixing processing to the endpoint processors.

ImmerGo is built on top of the Ethernet AVB standard, which provides the control protocol, clock synchronization and audio transport for the endpoint processors. The standard is built on layer 2 of the network stack and ensures a Quality of Service (QoS) that guarantees the following:

1. Zero packet loss.
2. Jitter-free clock synchronization.
3. Low-latency delivery of audio samples and control packets.

Ethernet AVB also hides the complexity of important components such as device discovery, audio transmission and synchronization which allows for simple plug and play operation by a user.

Because of this, the distributed endpoint approach used within ImmerGo already provides a solution to some of the problems described with the big box approach. In particular,

1. For every speaker, there is a small processor that handles the calculations and audio processing for that speaker. This provides the ability to scale the speaker configuration in small incremental steps.
2. The Ethernet AVB protocol provides a standard audio transport and connection management protocol discovers devices on the network, and can stream audio synchronously between audio sources and endpoints.
3. Ethernet AVB's clocking algorithm ensures that the endpoint devices have their word clocks synchronized.
4. The processors may be controlled remotely by a server that responds to clients served in a web browser which enables interoperability between devices and operating systems.

ImmerGo's use of OBA and Ethernet AVB provide the framework for implementing and testing a distributed endpoint processor 3D spatial audio system. In particular, the OBA representation enables the use of other spatial audio algorithms to calculate mixing parameters based on locational information of tracks and speakers.

1.6 Hypothesis

The ImmerGo system provides a level of distributed processing by distributing the task of mixing multiple audio channels to the speaker endpoints. This mixing process is the final stage of the Distance Based

Amplitude Panning algorithm (DBAP) currently utilized by ImmerGo. DBAP is in itself quite processor intensive, and all the calculations for it, apart from the final mixing stage, are performed on the server.

The project work of this thesis had two goals:

1. To expand on the range of algorithms that the ImmerGo system could utilize.
2. To distribute further algorithm processing beyond just the final mixing stage.

The hypothesis that this project work intended to prove was the following:

“The algorithmic processing required to deliver multichannel sound source localization can be successfully offloaded to processors that reside within speaker endpoints.”

To prove this hypothesis for a variety of spatial audio algorithms, ImmerGo was modified to contain further spatial audio algorithms implemented at 3 successive steps of distribution as described in **Table 1**.

Step	Host Computer	XMOS Microcontroller	SHARC DSP Chip
# 1	Spatial Audio Rendering Calculations	Audio Mixing Only	Nothing
# 2	Nothing	Spatial Audio Rendering Calculations + Audio Mixing	Nothing
# 3	Nothing	Spatial Audio Rendering Calculations	Audio Mixing

Table 1: Iterations of distribution.

The first step makes use of the distributed audio mixing mechanisms that are currently in place within the ImmerGo system, but with additional rendering algorithms. The second step increases the level of distribution by performing both the spatial audio rendering calculations and mixing of these algorithms within the endpoints. The third and final stage increases this level of distribution by performing the calculations on the XMOS processor and the audio mixing on the SHARC DSP chip.

Tests have been constructed to determine the viability of distribution for each of the steps and applied to a range of algorithms. These tests are described, and conclusions are drawn relating to the hypothesis.

Throughout this thesis, code examples are used to show exactly how certain mechanisms were built in either the ImmerGo software application or the XMOS processor firmware. Wherever shown, the shaded code examples show the code that is within the software application, and the code that is not shaded refers to code within the XMOS processor firmware.

Chapter 2

Ethernet AVB and ImmerGo – A platform for distributed immersive audio

The ImmerGo spatial audio system provides a framework for the implementation and testing of a distributed endpoint spatial audio rendering system at the three levels of distribution given in the previous chapter. As discussed in the introduction, ImmerGo is built upon the Ethernet AVB standard and uses Ethernet AVB compliant endpoint devices that handle the spatial audio mixing. As well as this, ImmerGo uses object-based audio representations to perform its spatial audio rendering. These two features are able to be utilized to further extend the endpoint distribution that is currently used by ImmerGo. The extension is a remote-control processing system where track object coordinates are sent to the distributed endpoint processors which perform spatial audio rendering calculations as well as audio mixing. This chapter will discuss the important aspects of Ethernet AVB and ImmerGo to provide a context for the further steps of distributed processing implemented within this project.

2.1 Ethernet AVB

2.1.1 Overview of Ethernet AVB

The Ethernet AVB standard comprises a group of open standards developed by the Institute of Electrical and Electronics Engineers (IEEE) designed specifically as a solution for time-sensitive audio networking. It operates at the link layer of an Ethernet network and hides the complexity of real-time audio transmission from the user ensured the appropriate *Quality of Service* (QoS). The device control and audio transport features make it a worthy candidate for further levels of the distributed endpoint processing approach.

The Ethernet AVB Standard adds two protocols:

1. IEEE 1722 – Audio Video Transport Protocol (AVTP) (IEEE, 2016).
2. IEEE 1722.1 – Audio Video Discovery, Enumeration, Connection management and Control protocol (AVDECC) (IEEE, 2018).

On top of three older IEEE 802.1 protocols:

1. IEEE 802.1BA – for profiles and overview (IEEE , 2011).
2. IEEE 802.1Q – for traffic shaping (FQTSS) and stream reservation protocol (SRP) (IEEE , 2014).
3. IEEE 802.1AS – for time synchronization (gPTP – Generalized Precision Time Protocol) (IEEE , 2011).

The 1722 protocol provides the ability for real-time multichannel audio transmission between devices. The 1722.1 protocol provides control and connection management of devices on the network. The older 802.1 protocols ensure a *Quality of Service* (QoS) delivery of media and synchronization of device clocks.

2.1.2 Ethernet AVB Devices

Ethernet AVB compliant devices may be assigned a combination of roles specified by the 1722.1 standard.

1. **A Controller**, which performs control operations by sending commands to devices on the network.
2. **A Talker**, which transmits one or more audio streams onto an Ethernet AVB network.
3. **A Listener**, which receives one or more audio streams from an Ethernet AVB network.
4. **An Interface**, used by all of the above to communicate with the Ethernet AVB network.

Each of these roles has associated with them representations, commands and state mechanisms that the device must implement.

A talker device can stream audio to other devices on the network. A listener device is one that can receive audio streams from a talker. The controller is able to make connections between talkers and listeners, and control any functions within the device. A device can be any combination of these roles. For instance, a device is able to be both a talker and listener, which means it has the ability to both stream and receive audio streams to and from the network. A device that is capable of streaming audio may also be a controller capable of controlling other AVB devices on the network. These all use an AVB interface to communicate with the AVB network.

Fig. 3 shows a simple Ethernet AVB network. Here, audio processing is performed on a live feed, and the processed audio is routed to the speaker outputs. The blue and purple boxes are the Ethernet AVB interfaces which are used by either a talker sending an audio source, or a listener receiving a source. The audio source received into the processor from the live feed is shown as a green arrow and the audio sent from the processor to the speakers is shown as a purple arrow. The red path shows the controller commands sent from the AVB controller to the endpoints which could control audio processes applied to the signal such as volume control, delays and equalizers, as well as device connections between all the devices on the network.

The Apple Macintosh has a native Ethernet AVB implementation that is both a controller and Ethernet AVB talker and listener entity. It is exposed to the user as a ‘virtual AVB entity’ audio device. The configuration of this virtual entity may be seen in **Fig. 4**. It can be customized to handle different stream sizes using a graphical interface. This is important when interfacing with different AVB devices on the network as different devices are able to have their own stream configurations.

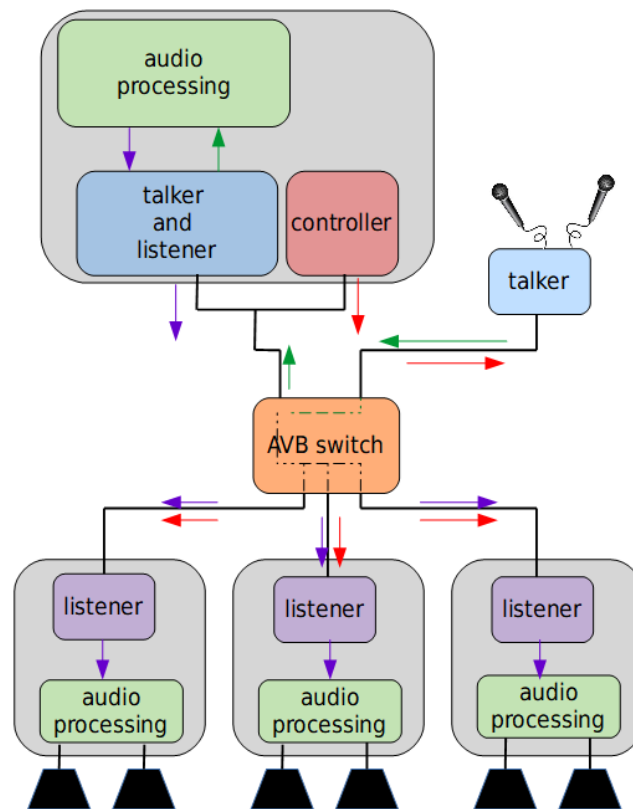


Fig. 3: A simple AVB network.

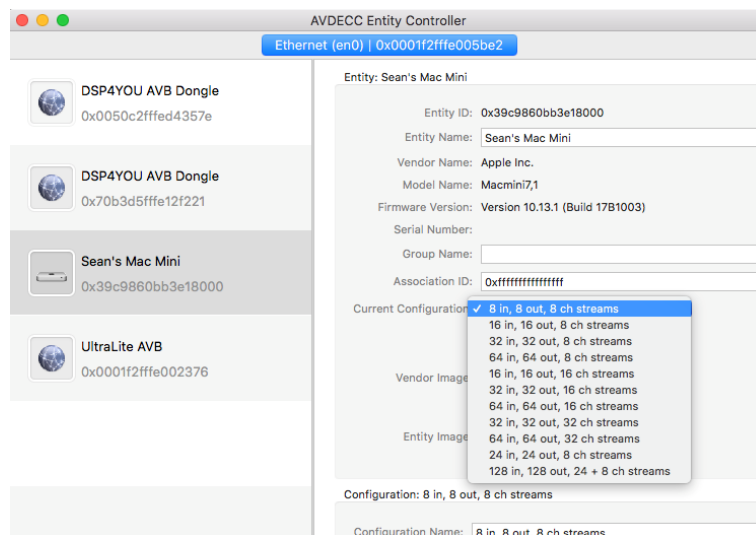


Fig. 4: The Apple Macintosh virtual AVB entity setup.

A controller can manage stream connections between talker and listener devices, as well as send control command requests to devices. The connection management process can be visualized using a connection routing matrix. Fig. 5 shows the Apple Macintosh native AVDECC controller's connection matrix. In this diagram, the talker output streams from the Apple's virtual AVB entity to multiple DSP4YOU dongle devices. The talker of the Ultralite device is also connected to the listener of the Apple virtual AVB entity.

Once a connection to a listener has been made, it needs to be disconnected before connecting to a different talker output.

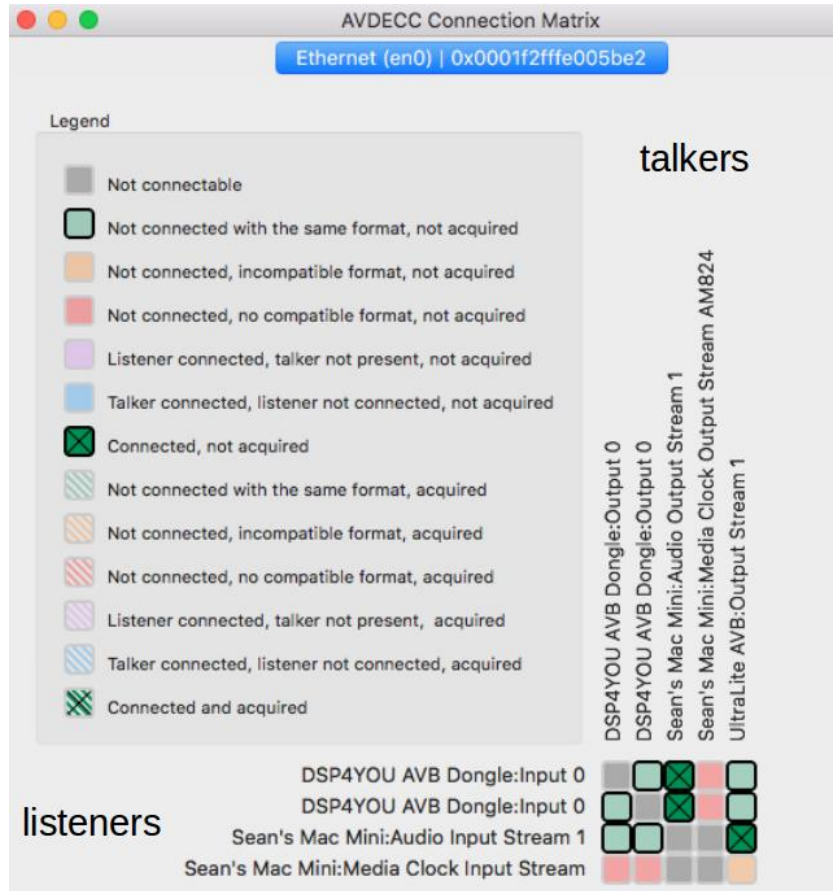


Fig. 5: The Apple Macintosh native AVDECC controller connection matrix

2.1.3 Device Clock Synchronization, Traffic Shaping and Stream Reservation

The project in this thesis relies on sample accurate, real-time delivery of audio streams to multiple devices. This is handled within the scope of Ethernet AVB and a brief overview of the components that allow this will be discussed.

The older IEEE 802.1 real-time network protocols that Ethernet AVB is built on, provide sample accurate transmission, presentation and jitter-free media clock synchronization for streams transmitted between AVB devices. The clock synchronization is handled by the IEEE 802.1AS generalized precision time protocol (gPTP) (IEEE , 2011). The gPTP provides a common time base for devices that is determined according to the gPTP grandmaster clock. The grandmaster clock for an AVB network may be set manually, or determined automatically using the gPTP Best Master Clock algorithm. Packets containing embedded timing information from this algorithm are exchanged between devices to keep device clocks precisely synchronized within a range of roughly 500ns.

The traffic shaping process and stream reservation protocol are handled by the IEEE 802.1Q protocol. It provides the mechanisms⁸ that ensure even packet transmission rates, network resource reservation and the dynamic maintenance of end to end connection streams.

2.1.4 The IEEE 1722 Protocol

The IEEE 1722 protocol is also known as the Audio Video Transport Protocol (AVTP). It allows for the streaming of uncompressed, synchronized audio to devices on the network with low latency, and provides the *data unit* encapsulation of both control messages and audio streams transmitted between devices. The packet layout for the AVTP is known as the AVTP *data unit* (AVTPDU). The AVTPDU packet layout is given in **Table 2** below.

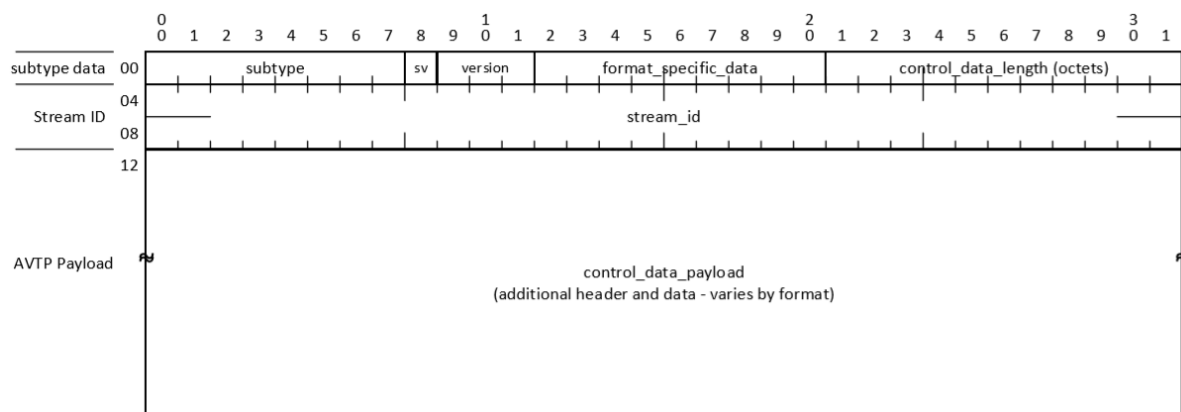


Table 2: The common AVTPDU layout

The AVTPDU allows a variety of audio formats to be sent over the AVB network⁹. The common format for audio is known as the AVTP Audio Format (AAF) which is given its own subtype. This format allows streams to contain single or multiple audio channels with a variety of sample rates, bit depths, and formats. An AAF stream consists of an audio frame that has a group of samples which are encoded using pulse code modulation (PCM). The audio sample buffer sizes for each audio sample frame are determined according to a ratio of the audio playback sample rate to the packet transmission rate. Generally, the transmission rate is set to 125μs. This means that for 32 channels with a playback sample rate of 48kHz, there will be 6 audio sample frames in a packet, with each frame holding 32 audio samples.

The IEEE working group has also adapted¹⁰ other audio formats such as the 61883-6 format first implemented within firewire (Technical Committee 100 - Digital system interfaces and protocols, 2005) to be transferrable over the Ethernet network. This is the audio format currently used by the XMOS endpoints within the ImmerGo system.

⁸ - IEEE Std 802.1Q-2011 Clause 34, "Forwarding and Queuing for time-sensitive streams"

- IEEE Std 802.1Q-2011 Clause 35, "Stream Reservation Protocol"

⁹ Pg22 of (IEEE, 2016)

¹⁰ Pg41 of (IEEE, 2016)

2.1.5 The IEEE 1722.1 Protocol

The IEEE 1722.1 protocol is also known as the AVDECC (Audio/Video Discovery Enumeration Connection management and Control) protocol. The protocol describes a device description model, known as the AVDECC Entity Model (AEM), that allows an AVB device to advertise its controls and functionality to other devices on the network. It also provides the necessary packet layout for transmitting control and connection management messages to these devices.

The protocol is split into four core components which are as follows,

1. **Discovery** is the process of AVDECC entities advertising themselves to other AVDECC devices on the network. Devices also advertise their network connection status.
2. **Enumeration** is the process of advertising the device AEM so it's capabilities may be used by an AVDECC controller. This information is also used in connection management to provide information on device capabilities to ensure device configurations are compatible prior to connecting.
3. **Connection Management** is the process of connecting audio streams from AVDECC devices to other AVDECC devices.
4. **Control** is the process that allows for device functions to be controlled by an AVB controller on the network.

Each of these components communicate according to their own protocol as shown below,

1. AVDECC Discovery Protocol (**ADP**)
2. AVDECC Connection Management Protocol (**ACMP**)
3. AVDECC Enumeration and Control Protocol (**AEC**P)

These protocols each have a unique packet structure, or data unit, associated with them. When AVDECC protocol packets are sent, the AVDECC data unit is encapsulated within an AVTPDU packet with the subtype field specific to the type of AVDECC message and the stream valid (sv) field set to 0. The research presented in this thesis uses AEC, and its associated AEC data unit (AECPU), to allow for various parameters needed for audio spatialization to be addressed within endpoint devices.

The common AECPU layout is given in **Table 3**.

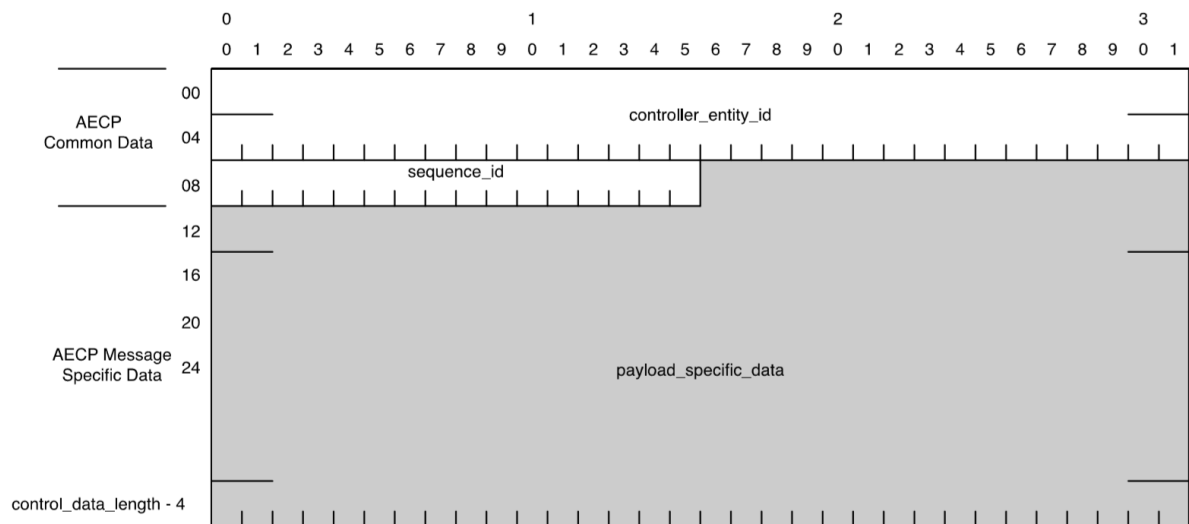


Table 3: The common AECPU layout.

The AECPU payload specific data is used to send specific control and enumeration requests, which are found in the 1722.1 specifications (IEEE, 2018). The research in this thesis is particularly concerned with the SET_MIXER AECPU command¹¹ using the descriptor type of MIXER. This command is currently used by ImmerGo to control the endpoint devices. The datagram used by this command is shown in **Table 4**.

The implementation in this project makes use of the various descriptor indexes to address parameters which control an audio mixer located within the endpoint listener devices. This process will be elaborated on in the following section and later chapters.

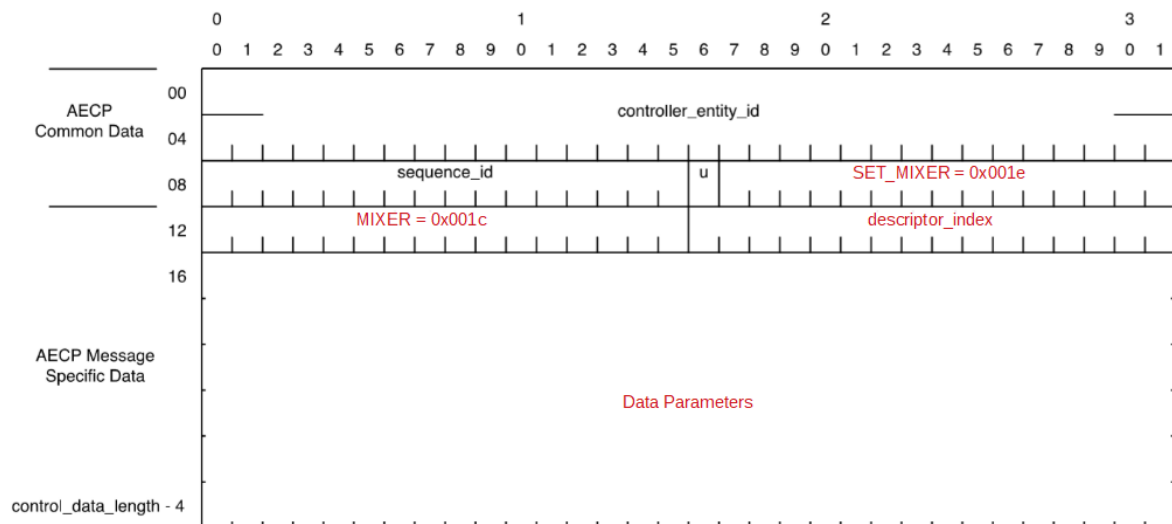


Table 4: The AECPU command layout.

¹¹ Pg213 of (IEEE, 2018)

2.2 ImmerGo

ImmerGo is an immersive sound rendering and control system that utilizes Ethernet AVB to transmit both audio and control data to endpoint devices on a network. It has a client-server-based approach to spatialization control and is designed to run in parallel with any audio software that can output multichannel audio to the Ethernet AVB network. It can work on a Macintosh OSX workstation or a Windows 10 workstation with an AVB capable network card.

2.2.1 ImmerGo Overview

The ImmerGo server has been built using a cross-platform NodeJS-based ¹² framework for desktop applications called NWJS ¹³ and runs as a lightweight desktop application. The server is able to receive track location updates from a mobile client device as well as connect, and control, multiple AVB endpoint devices on the network. It also provides the capability for the client device to interact with a DAW. ImmerGo's client devices use web sockets to communicate various spatialization and DAW transport requests to the server. The server uses these requests to control the DAW, or perform immersive audio rendering and send audio mixer updates to endpoints. It also uses a metadata configuration file to automatically manage stream connections between the endpoint devices. These endpoint devices are AVB-enabled audio processors that are attached to speakers, and perform audio mixing according to AECP commands.

Fig. 6 shows a typical system setup. In this diagram, the ImmerGo server is hosted on a single Apple Macintosh computer. It sends AVDECC requests to endpoint devices on the AVB network using Apple's native virtual AVB entity. It also communicates with a DAW via an internal MIDI port. The web client is hosted on any device that has a web browser and can connect to the same network that the server is on. On the server host, an XML file within the same directory as the server application contains loudspeaker configuration information that consists of each device's media access control (MAC) address, the number of speakers and their physical positions in the room. Upon initialization, this file is loaded into program memory and parsed into its own room model. These models are able to be used by the spatial audio renderer within the server to output speaker mix values, as well as enable the server to send AVDECC messages to the correct MAC addresses of each endpoint device. The AVDECC messages provide the control and AVB stream connection management between each endpoint device and the Mac virtual AVB entity.

¹² <https://nodejs.org/en/>

¹³ <https://nwjs.io/>

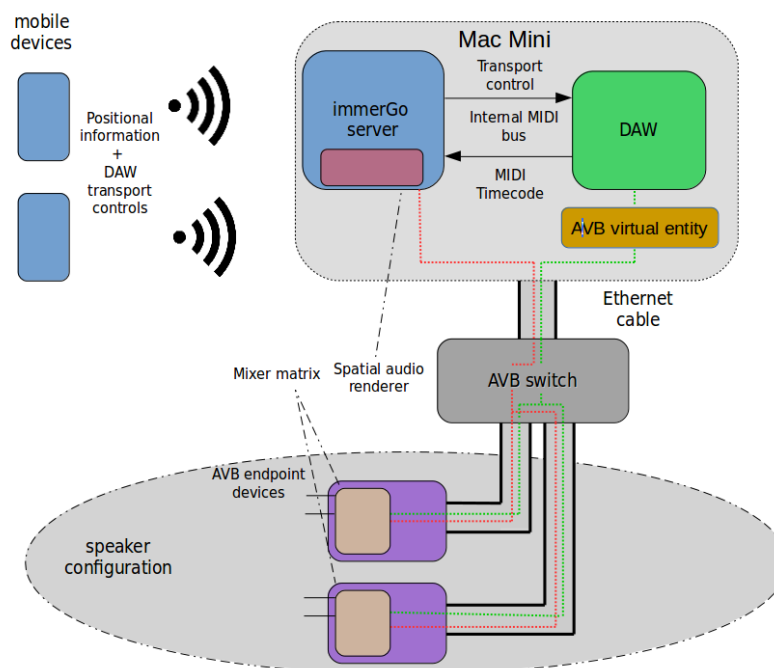


Fig. 6: A typical ImmerGo setup. The red dotted lines show control messages being sent to the mixer matrix. The green dotted lines show audio being sent from the DAW.

ImmerGo uses the MAC addresses within that loudspeaker configuration XML file to send ACMP messages (IEEE, 2013) to the appropriate endpoint devices to automatically set up relevant device connections prior to streaming audio. This removes the need for the audio engineer to set them manually.

The spatial audio renderer is split into two main processing tasks as shown in **Fig. 6**. The first is the rendering of gain values within the server according to its rendering algorithm. Which is followed by the audio sample mixing according to the renderer's output and digital to audio conversion of the audio signals to the speakers connected to each endpoint. Ethernet AVB AECp provides the protocol for the control of each processor's mixer matrix based on mixer values output from the server's rendering algorithm.

All the real-time rendering communication and rendering calculations from client to server to endpoint devices happens according to a clock. This clock may be internal to the ImmerGo server, or according to MIDI timecode (MTC) from the DAW via the internal virtual MIDI bus. The latter is the method used in this project. The clock ensures that track object location updates happen synchronously, whilst also allowing their locations to be recorded according to a timestamp so they can be played back at a later stage. The server also uses the internal virtual MIDI bus to send messages to the DAW which control transport functions such as play and pause.

The project described in this thesis modifies the communication mechanism between the server and the endpoints to send track object coordinates instead of mixer updates using AECp. It also modifies each endpoint processor to have its own spatial audio renderer that may use these track object coordinates with the spatial audio renderer to mix the audio appropriately to speaker outputs. The following subsection will describe the server NW.js environment to provide a context for the modifications done for this project.

2.2.2 The NW.JS Environment

NW.JS is built from three different web technologies, namely the NodeJS framework, Chromium and a small web API. It allows for desktop applications to be made using web technologies.

NodeJS comes with its own package manager called the Node Package Manager (NPM). The package manager acts as a repository that third-party developers from the NodeJS community use to upload libraries of code that can then be easily incorporated within other applications. It allows for the linking of modules, which are hosted on the online repository¹⁴ by simply using the NPM command line tool to install the package into the project directory. From there, the modules may be incorporated into the project using the ‘require’ keyword within the codebase as shown in **Listing 1**.

```
134 | const fs = require('fs'); // for file reads/writes
```

Listing 1: Module linking in Javascript

ImmerGo uses the following NPM packages,

- **fs** – to read/write to a file
- **xml2js** – used to parse .xml files
- **mac** – used for parsing MAC addresses
- **bufferEqual** – used to compare Ethernet buffers
- **path** – used to read directory paths
- **midi** – used to interface with midi drivers
- **pcap** – used to capture and send Ethernet packets
- **network** – provides an interface to the Ethernet network
- **http** – provide a server to handle HTTP requests
- **socket.io** – provides web socket interfaces to the application
- **xmldom** – used to parse XML files into a document object model (DOM)
- **underscore** – Javascript helper functions

The project in this thesis made particular use of the midi package and the cap package. The midi package is used to create a virtual input and output port between ImmerGo and the DAW which sends timecode from the DAW, as well as various transport controls between the DAW and ImmerGo. The cap package is used to capture and send AVDECC packets to control the endpoint devices on the network.

2.2.3 The ImmerGo Client

The client provides a user with control over track spatialization using a GUI that is served within a web browser. **Fig. 7** shows the GUI.

¹⁴ <https://www.npmjs.com/>

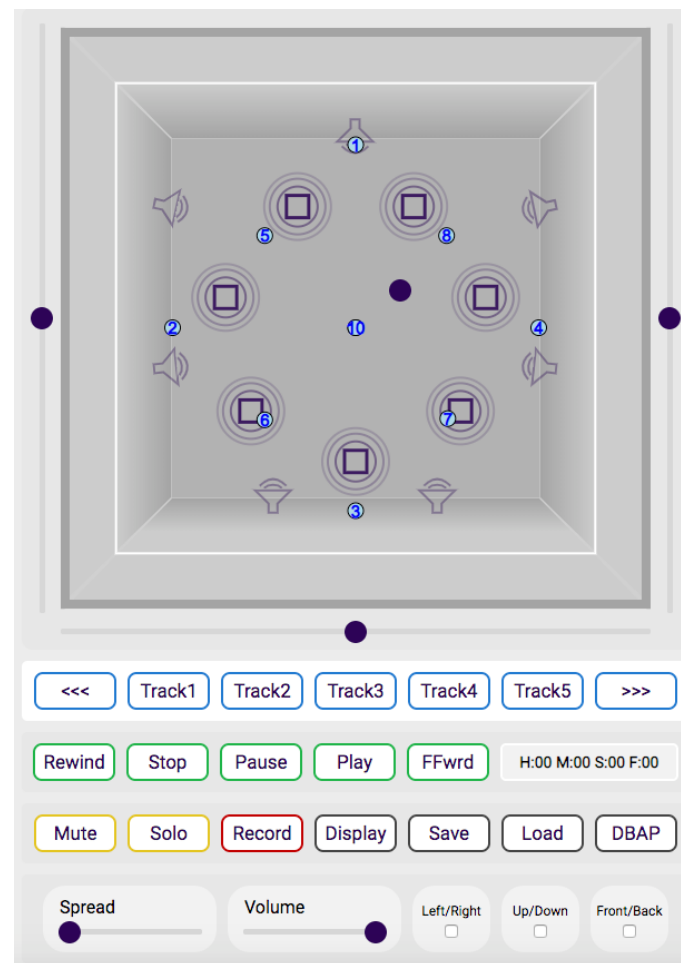


Fig. 7: The Immergo UI.

The upper half of the interface is the track object control window and holds visual representations of speaker objects and track objects. It shows a room with the physical speaker layout and walls surrounding the speakers. Speakers on the roof are denoted as squares with circles surrounding them and speakers to the side are denoted as a typical speaker graphic, with speakers lowest in the room being more opaque than speakers on the top. The virtual source movements for a selected track are controlled by moving the purple dot. The smaller blue dots show the position of the other tracks when they are selected for display. The white line on the wall shows the current height of the track object in the room. The slider on the side of track object control may be used to control the height of the object.

The lower half of the interface holds all the track selection buttons, as well as the track's volume, spread and the DAW transport control. Users are able to view other selectable tracks by clicking on the arrows to either side of the track selector, and then able to select a track by pressing on a track number. Once selected, the large purple dot denotes the track position at the current time and is used to position track objects which then have their metadata sent to the server and fed into the spatial audio renderer pipeline. The checkbox in the bottom right of the controls can be selected so that a mobile controller device can be tilted to provide a height change. There are mute and solo buttons which may be selected on any track. The solo button mutes all tracks except the current track. Transport controls for play, pause, stop, fast forward and rewind interact with the DAW using MIDI. The current time position of the DAW is displayed to the

right of the transport controls. It also has a record, save and load button that is used to record, save and load track position automation. A user can select a track and press the display button, which will then display the track's position in the sound object control window.

The client itself comprises two files, an ImmergoClient.html and main.js file. The HTML file provides the structure for the web browser application and the .js file provides the functionality. These files are served to a web browser upon an HTTP request to the URL specified by the ImmerGo server.

2.2.4 ImmerGo Renderer Definition Models

The ImmerGo system treats an immersive audio scene as having two main components, the track objects and the loudspeaker objects. The model's contents may be passed as parameters to spatial audio rendering algorithms to solve for localization mixing parameters. Each model has its own metadata that describes its locality, identification and object-specific parameters. Collectively they are called the "Audio Definition Models".

2.2.4.1 Room Configuration Model

The listening room configuration model is shown in **Table 5** below. The model holds each endpoint processor used by the system. These endpoints have associated with them both their Ethernet MAC address and the loudspeakers attached to it, with their locations. This model is held within an XML file within the ImmerGo project directory that is read in by ImmerGo at startup. The XML file holding the speaker configuration is given in **Listing 2**. Here we see an endpoint processor described within the "amplifier" tag. This tag has associated with it an object variable "macadr" that holds the MAC address for that endpoint device. Within the tag are further "speaker" tags that hold the location of each speaker that is attached to the endpoint processor.

Room Configuration Model	
-	Endpoint Processors
-	MAC Address
-	Loudspeaker Locations

Table 5: Room Configuration Model

In the ImmerGo runtime, the loudspeakers from the room configuration XML are read into a loudspeaker model that holds the loudspeaker location and the mix levels for each of the tracks being localized. The loudspeaker coordinates are used by the renderer to calculate mix levels for each track position and for display on the client UI. This is shown in **Table 6**.

Loudspeaker Model	
-	Position (x,y,z)
-	Mix levels for each track

Table 6: Loudspeaker model

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <config>
3   <roomheight> 320 </roomheight>
4   <amplifier macadr = "70:b3:d5:12:f3:c1">
5     <speaker number = "1" xpos = "-77" ypos = "0" zpos = "0 "> </speaker>
6     <speaker number = "2" xpos = "-63" ypos = "0" zpos = "0 "> </speaker>
7   </amplifier>
8   <amplifier macadr = "70:b3:d5:12:f3:b9">
9     <speaker number = "1" xpos = "-49" ypos = "0" zpos = "0 "> </speaker>
10    <speaker number = "2" xpos = "-35" ypos = "0" zpos = "0 "> </speaker>
11  </amplifier>
12  <amplifier macadr = "70:b3:d5:12:f3:b6">
13    <speaker number = "1" xpos = "-21" ypos = "0" zpos = "0 "> </speaker>
14    <speaker number = "2" xpos = "-7" ypos = "0" zpos = "0 "> </speaker>
15  </amplifier>
16  <amplifier macadr = "70:b3:d5:12:f3:b7">
17    <speaker number = "1" xpos = "7" ypos = "0" zpos = "0 "> </speaker>
18    <speaker number = "2" xpos = "21" ypos = "0" zpos = "0 "> </speaker>
19  </amplifier>
20  <amplifier macadr = "70:b3:d5:12:f3:ba">
21    <speaker number = "1" xpos = "35" ypos = "0" zpos = "0 "> </speaker>
22    <speaker number = "2" xpos = "49" ypos = "0" zpos = "0 "> </speaker>
23  </amplifier>
24  <amplifier macadr = "70:b3:d5:12:f3:b4">
25    <speaker number = "1" xpos = "63" ypos = "0" zpos = "0 "> </speaker>
26    <speaker number = "2" xpos = "77" ypos = "0" zpos = "0 "> </speaker>
27  </amplifier>
28 </config>

```

Listing 2: The room configuration XML.

2.2.4.2 ImmerGo Track Object Model

ImmerGo uses a track object model for spatialization. This model is created for each track and updates are made to it whenever the client moves a track. This model is shown in **Table 7** below.

ImmerGo Track Object Model	
- Position (x,y,z)	<i>Dynamic</i>
- Spread	<i>Dynamic</i>
- Volume	<i>Dynamic</i>
- Track ID	<i>Static</i>
- Mute	<i>Mix Control</i>
- Solo	<i>Mix Control</i>
- Recording	<i>Mix Control</i>
- Selected	

Table 7: ImmerGo Audio Object Model

The model can be split into static, dynamic and mix control components.

A static component is one that is unchanged, in this instance, the track ID. It is associated with each output audio channel from the DAW, and when a track is selected a user is able to control the model components and selected the track to have some of these components recorded for later playback.

The dynamic components, such as the positions, spread and volume, can be changed by a user over time and saved according to a timestamp from MIDI Timecode. This allows for track movement and mixing parameters to be recorded and then played back at a later stage.

The mix control components are the solo, mute and recording variables.

These definition models provide the information for a spatial audio renderer to generate mix values for tracks being mixed to loudspeakers.

2.2.5 ImmerGo's Rendering Algorithm – Distance Based Amplitude Panning

The definition models described in the chapter above are used by the Distance Based Amplitude Panning (DBAP) algorithm implemented within the server. DBAP was introduced in (Lossius, Baltazar, & Hogue, 2009) as a way of using object-based audio in more irregular and sparse loudspeaker arrangements. It is well suited for a speaker array where the listening "sweet spot" is not restricted to a specific position. For this reason, DBAP can be used in a variety of real-world situations such as concerts, stage productions, installations and museum sound design (Lossius, Baltazar, & Hogue, 2009). It is still a relatively new technique for sound localization and only one formal evaluation has been done (Kostadinov, Reiss, & Mladenov, 2010). This section will explain the theory behind DBAP and how it has been implemented within ImmerGo. Most of this chapter's explanation has been taken from (Lossius, Baltazar, & Hogue, 2009) and to avoid confusion the notation used is similar to the notation in (Lossius, Baltazar, & Hogue, 2009). The work done in this thesis is concerned with implementation of three further immersive audio rendering algorithms within ImmerGo alongside DBAP, and therefore their technical theory will be discussed alongside their implementation in later chapters.

2.2.5.1 DBAP Theory

DBAP extends the principle of equal intensity panning from a pair of loudspeakers to a loudspeaker array of any size with no prior assumptions about their positions in space or relationship to each other (Lossius, Baltazar, & Hogue, 2009) and can be used in both two dimensions and three dimensions.

Shown in **Fig. 8** below are the measurements and geometry associated with the DBAP calculation in three dimensions.

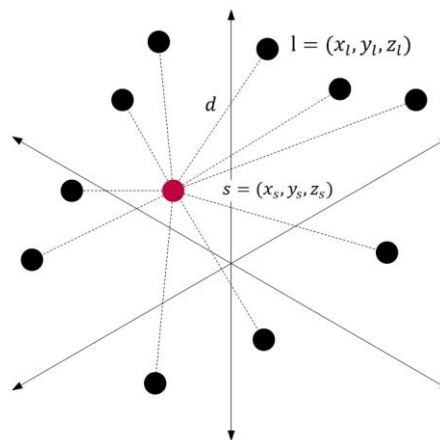


Fig. 8: Geometry for DBAP.

As shown in **Fig. 8**, the Cartesian coordinates of where the virtual source is to be placed are given by $s = (x_s, y_s, z_s)$. For a loudspeaker array with N loudspeakers where the position of the i th speaker is given by $l_i = (x_i, y_i, z_i)$, the distance d_i from the source to each of the loudspeakers is given by the familiar equation:

$$d_i = \sqrt{(x_i - x_s)^2 + (y_i - y_s)^2 + (z_i - z_s)^2} \quad (2.1)$$

for $1 \leq i \leq N$

DBAP then makes two assumptions. The first is that the intensity of the virtual sound source is to be constant regardless of its position in the sound scene (Kostadinov, Reiss, & Mladenov, 2010). This can be explained by the principle of constant intensity panning in a stereo set up whereby the centre panning a sound signal means both loudspeakers are playing the signal at the same volume. As such, if the amplitude of the i th speaker is g_i such that $0 \leq g_i \leq 1$ and I is the total intensity of the loudspeaker array, then

$$I = \sum_{i=1}^N g_i^2 = 1 \quad (2.2)$$

Secondly, it is assumed that all the loudspeakers are to be active at all times and the gain factor of a j th speaker is related to the distance of the virtual source to that speaker. This is shown by the equation (2.3)

$$g_j = \frac{c}{d_j^a} \quad (2.3)$$

where c is a normalization constant that takes into account the equal intensity panning and a is a coefficient calculated from the rolloff R in decibels per doubling of distance, given by

$$a = \frac{R}{20 \log_{10} 2} \quad (2.4)$$

A rolloff of $R = 6$ dB is equivalent to the inverse distance law for sound propagating in a free field. For closed or semi-closed environments R will generally be lower, somewhere in the range of 3-5 dB, and depends on the reflections and reverberations of the listening space (Everest & Pohlmann, 2014).

Combining equation (2.2) and equation (2.3) we find that

$$c = \frac{1}{\sqrt{\sum_{i=1}^N \frac{1}{d_i^{2a}}}} \quad (2.5)$$

So, the gain g_j of the j th loudspeaker can be found to be

$$g_j = \frac{1}{d_j^a \sqrt{\sum_{i=1}^N \frac{1}{d_i^{2a}}}} \quad (2.6)$$

This is repeated for each of the loudspeakers in the array to solve their gains as well.

DBAP can also compensate for virtual sources located outside the field of the loudspeakers with some additional calculations (Lossius, Baltazar, & Hogue, 2009). First, the convex hull of the loudspeakers (i.e

smallest convex space that contains each loudspeaker) needs to be calculated. Once calculated, the virtual source needs to be identified as being inside or outside of the convex hull. If it is inside the convex hull or on the boundary, the DBAP calculation is performed as has been described. If it is outside of the convex hull, a projection of the virtual source onto the boundary of the convex hull is calculated by determining the shortest possible distance between the source and the convex hull. From here, the projection of the virtual source is substituted for its actual position outside of the loudspeaker array (Lossius, Baltazar, & Hogue, 2009).

The distance value of the exterior virtual source to the boundary can be saved for further spatial processing such as gain attenuation, air filtering, the Doppler effect or gradual introduction of reverb (i.e more reverb added the greater the distance) (Lossius, Baltazar, & Hogue, 2009).

ImmerGo does not account for sources outside the array and so the above calculations are the one used in its implementation.

2.2.5.2 Spatial Blur

If a virtual source is panned to the exact position of one of the loudspeakers, equation (2.6) will cause a division by zero. Squaring equation (2.6) we have

$$g_j^2 = \frac{1}{\sum_{i=1}^N \frac{d_j^{2a}}{d_i^{2a}}} \quad (2.7)$$

and from this, we can see that

$$\lim_{d_j \rightarrow 0} g_i = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \quad (2.8)$$

In other words, when the virtual source is panned to exactly the position of one of the loudspeakers, only that speaker will be emitting the sound. This may cause unwanted changes to the spread and colouration of the virtual source. In order to account for this, a spatial blur constant $r_s \geq 0$ is added to the distance formula in equation (2.1). This gives us

$$d_i = \sqrt{(x_i - x_s)^2 + (y_i - y_s)^2 + (z_i - z_s)^2 + r_s^2} \quad (2.9)$$

In two dimensions, spatial blur can be understood as a vertical displacement between the virtual source and loudspeakers (Lossius, Baltazar, & Hogue, 2009). If r_s is large, the source won't be able to gravitate as easily towards one speaker when being panned. It has been shown in (Litovsky, Colburn, Yost, & Guzman, 1999) that there is a certain limit to the amount of spatial blur that can be added before the precedence effect starts to become a problem. The precedence effect is perceived as unwanted gravitation of the source towards the loudspeakers closest to the listener.

The spatial blur constant (r_s) is scaled for the size of the room using a factor determined by the covariance of the distance from the centre of the loudspeaker rig to all the loudspeakers. The use of this covariance

factor enables the perception of the blur to be less sensitive to changes in the loudspeaker layout size (Lossius, Baltazar, & Hogue, 2009). The covariance factor for a set of N loudspeakers is given by,

$$\text{Cov} = \sqrt{\frac{1}{N-1} \sum_{i=0}^N (x_i - \bar{x})^2 + (y_i - \bar{y})^2 + (z_i - \bar{z})^2} \quad (2.10)$$

Where $(\bar{x}, \bar{y}, \bar{z})$ is the mean of all the positions of the loudspeakers.

2.2.5.3 Weighting sources to chosen loudspeakers

DBAP can be extended to enable a virtual source to be weighted to a subset of the total loudspeakers. This provides several artistic possibilities as it allows virtual sources to be restricted to certain areas within a listening environment, or it can be more intense in certain places within the speaker array. For instance, one virtual source channel could be in a particular room while another is free to move around all the rooms (Lossius, Baltazar, & Hogue, 2009).

It is a relatively easy extension and only requires a weight $0 < w_j \leq 1$ to be added to equation (2.3), where w_j is the weighting of a virtual source for loudspeaker j such that,

$$g_j = \frac{cw_j}{d_j^a} \quad (2.11)$$

Thus, each loudspeaker requires a weighting for the virtual source which modifies equation (2.6) to

$$g_j = \frac{w_j}{d_j^a \sqrt{\sum_{i=1}^N \frac{w_i^2}{d_i^{2a}}}} \quad (2.12)$$

These weightings can be used for multiple virtual sources which can be used to differentiate the spatial audio spread of each loudspeaker and enable individual zones and topologies for the different virtual sources to move through (Lossius, Baltazar, & Hogue, 2009).

2.2.5.4 DBAP Algorithm

The DBAP algorithm is as follows, assuming there is one source to be localized between N loudspeakers,

1. Determine distances of the source to all loudspeakers.
2. For each loudspeaker, perform the inversion distance calculation, add to the power normalization coefficient and save the inverse distance calculation in memory.
3. Take the square root of the normalization coefficient and for each loudspeaker multiply the normalization coefficient by its own unique inverse distance of source to speaker to get the final gain factor for that speaker.
4. Create a copy of the signal for each loudspeaker and weight it according to its gain factor.

This algorithm leaves us with a gain value that is used to control the level of the source going to that loudspeaker, where $L_i(t)$ is the signal going to the loudspeaker at sample t as shown in equation

(2.13) below:

$$\begin{bmatrix} L_1(t) \\ L_2(t) \\ \dots \\ L_N(t) \end{bmatrix} = s(t) * \begin{bmatrix} g_1 \\ g_2 \\ \dots \\ g_N \end{bmatrix} \quad (2.13)$$

2.2.6 Panning Sources in ImmerGo

Fig. 9 shows the ImmerGo application flow from when the speaker configuration file is read into the server application to localizing a track source.

1. The speaker configuration is first loaded into memory,
2. The covariance factor for the speaker array is solved ¹⁵.

At this point, ImmerGo is ready to begin panning.

1. The user may then make requests to the server over the websocket using a mobile device's web browser.
2. The server responds with the ImmerGo UI.

Once the user has the client interface within their browser they may begin interacting with the server.

1. When a track source is moved by a client, the coordinates are transmitted to the server.
2. Once received, these are used by the DBAP algorithm to calculate the mix levels housed in the loudspeaker model.
3. The server then updates the speaker model mix levels for that track.
4. These mix levels are then sent to the endpoints every time an MTC frame is received from the DAW.

¹⁵ For more detail see Appendix A A1

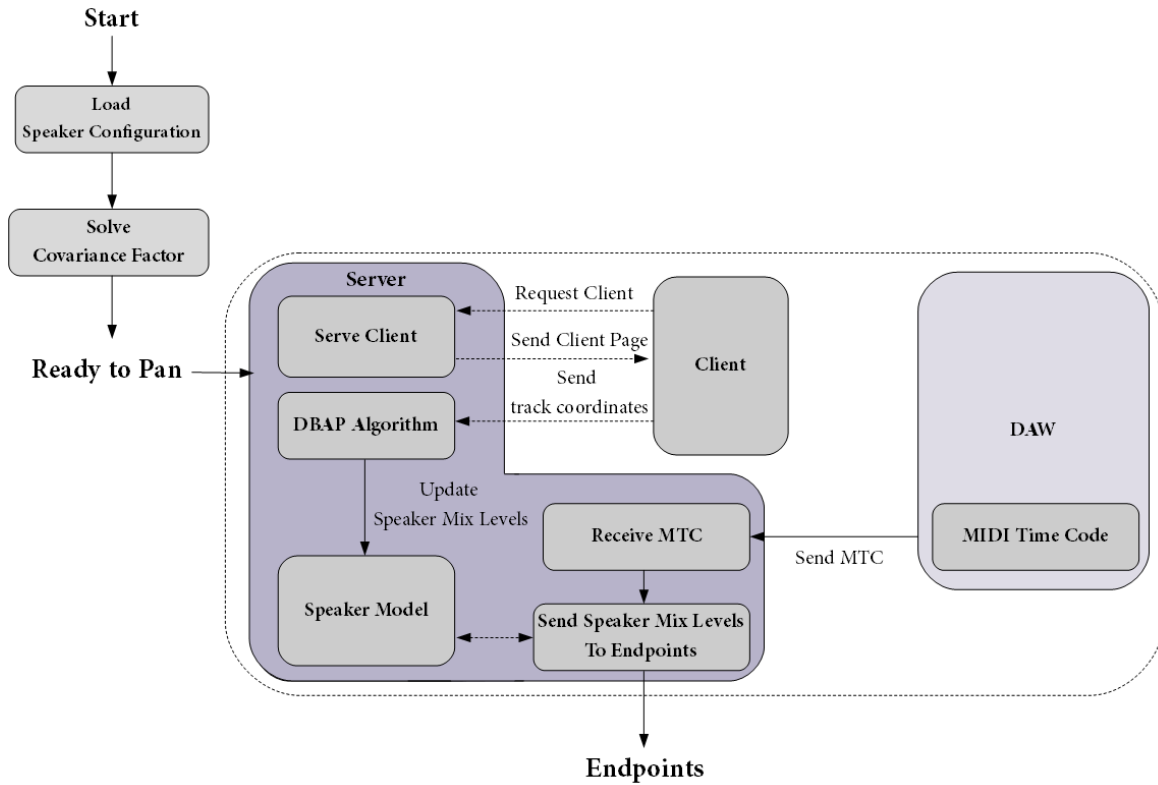


Fig. 9: Diagram showing DBAP setup and panning mechanism

2.2.6.1 Receiving Track Updates from the Client

The client device is able to send track updates once every 8ms. This value was set so to correspond to the intervals of the MTC quarter frame and was found experimentally to provide the best balance between accuracy and responsiveness.

Listing 3 shows the function used to receive the track objects into the server object models. When the socket receives the “coords” message, it updates the correct track object model, and then speaker mix levels for that track according to DBAP within the *updateSpeakerMixLevels* function.

```

2680 // Within the ImmerGo server
2681 // 3D coords for a given track are sent from client
2682 socket.on('coords', function(xCoord,yCoord,zCoord,trackNo) {
2683
2684     if (!(tracks3D[trackNo].mute)) {
2685         // update the mix levels for each speaker for the given track
2686         updateSpeakerMixLevels(trackNo, xCoord, yCoord, zCoord);
2687
2688         if (MTCOn == false) {
2689             // If there is no timecode, send updates to endpoints
2690             sendMixLevels();
2691         }
2692     }
2693 });

```

Listing 3: Receiving track object updates within the server.

The DBAP algorithm implementation is given in **Listing 4**. Here the distance between the source and each loudspeaker is solved according to the algorithm described in section 2.2.5.4. Once every speaker’s gain value has been solved, and the normalization coefficient solved, the mix levels of that track for each speaker

is set by multiplying the gain value by the normalization coefficient. Each loudspeakers array holding the mix levels for the track being panned are updated by transforming the floating point calculation into an 8 bit char with the *mixLevelAdjust* variable and scaling that by the track's volume.

```

2316     let mixLevelAdjust = 255; // adjusts the mix level from 0-1 to 0-255 for mixer matrix
2317     let radius = 0.25;      // Spatial blur constant
2318
2319     // In updateSpeakerMixLevels...
2320     // Perform DBAP with source
2321     for (let i=0; i<speakerConfig.amplifiers.length; i++) {
2322         dPower[i] = [];
2323         dInversePower[i] = [];
2324
2325         for (let j=0; j<speakerConfig.amplifiers[i].speakers.length; j++) {
2326             xl = speakerConfig.amplifiers[i].speakers[j].x;
2327             yl = speakerConfig.amplifiers[i].speakers[j].y;
2328             zl = speakerConfig.amplifiers[i].speakers[j].z;
2329
2330             xlminusxs = xl-xs;
2331             ylminussy = yl-ys;
2332             zlminuszs = zl-zs;
2333
2334             sumofsquares = Math.pow(xlminusxs,2)+Math.pow(ylminussy,2)+Math.pow(zlminuszs,2);
2335             spatial_blur = (radius*covariance);
2336             spatial_blur *= spatial_blur;
2337             // distance from virtual source to speaker
2338             d = Math.sqrt(sumofsquares + spatial_blur);
2339             dPower[i][j] = d;
2340             dInversePower[i][j] = 1/dPower[i][j];      // Inverse power
2341             // Normalization Coefficient
2342             powerTotal = 1*powerTotal+Math.pow(dInversePower[i][j],2);
2343         }
2344     }
2345 }
2346
2347 /* Set track Levels */
2348 // For each amplifier calculate track mix Levels for each speaker
2349 for (var i=0; i<speakerConfig.amplifiers.length; i++) {
2350     for (var j=0; j<speakerConfig.amplifiers[i].speakers.length; j++) {
2351         speakerConfig.amplifiers[i].speakers[j].mixLevels[trackNo] =
2352             (dInversePower[i][j]
2353              *Math.sqrt(powerTotal))
2354              *mixLevelAdjust
2355              *(tracks3D[trackNo].volume/100);
2356     }
2357 }

```

Listing 4: The DBAP algorithm and updating the speaker mix levels within *updateSpeakerMixLevels*.

2.2.6.2 MIDI Synchronization from the DAW

All the real-time transmission of the endpoint mixer updates are synced to MIDI Time Code (MTC) received from the DAW. Typically, an MTC frame rate of 30 frames per second is used. A full MTC frame is represented in 32 bits and sent as 4 separate 8 bit bytes at quarter frame intervals of roughly 8.3ms.

Fig. 10 shows how the MTC is received within the NWJS application. MTC values are sent as 2 byte messages with the most significant byte containing the type of MIDI message received and the least significant byte containing the actual value.

First, the MIDI input port is opened using the NPM MIDI library previously mentioned. Whenever the MIDI input port receives a message, it determines what type of message it is. If the first byte is 0xF1 then

the server knows it's receiving a quarter frame of MTC and begins building the frame, otherwise it does nothing. The second byte of the message indicates the MTC type: frames, hours, minutes and seconds. Using a switch statement with the MTC type, the server knows where to start filling the MTC frame should the project be started somewhere between frames. Once a full frame is received, the server sends the speaker mix levels within the loudspeaker model to the endpoints and sends the MTC value back to the client for visual feedback of the DAW timeline.

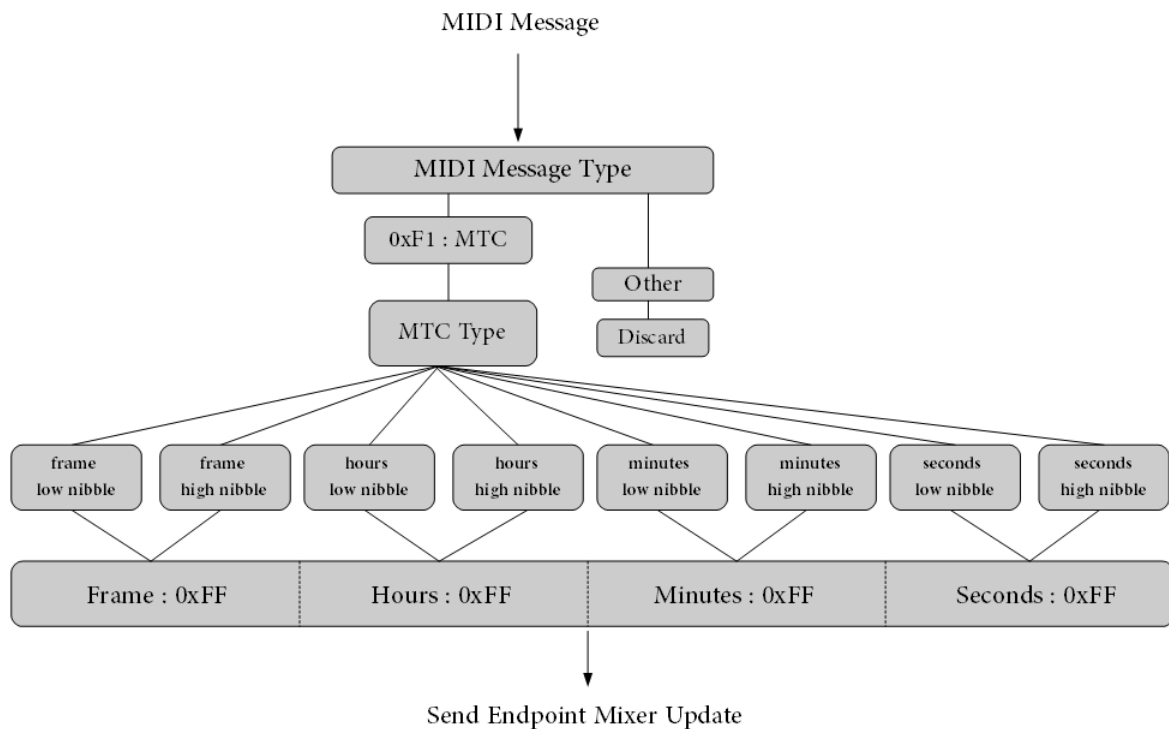


Fig. 10: Receiving an MTC frame.

2.2.6.3 Recording Track Object Updates as Automation

There are two ways that track objects may move. The first way is by a user selecting a track and moving it. The other way is by playing back recorded track movements which allows for multiple tracks to move simultaneously at every MTC quarter frame. An overview of this process is presented since it is used by all the localization techniques discussed in this thesis. When a track is selected for recording, its positions are saved into a track node according to the current quarter frame received and added to a linked list of nodes for that track. The track node structure is shown in Table 8.

ImmerGo Track Recording Node
- Quarter Frame Counter
- Position (x,y,z)

Table 8: ImmerGo's automation track node.

When the recording button is pressed, the server enters the recording state. Fig. 11 describes the recording process while the server is in the recording state, from the point of receiving an MTC message:

1. When a quarter of the MTC frame is received, the quarter frame is incremented.
2. If a track is selected, its current localization position is checked against the last saved position in the linked list.
3. If the position is changed, then a new track node is made with the current count of quarter frames.
4. This node is then saved to the end of the localization position list for that track.

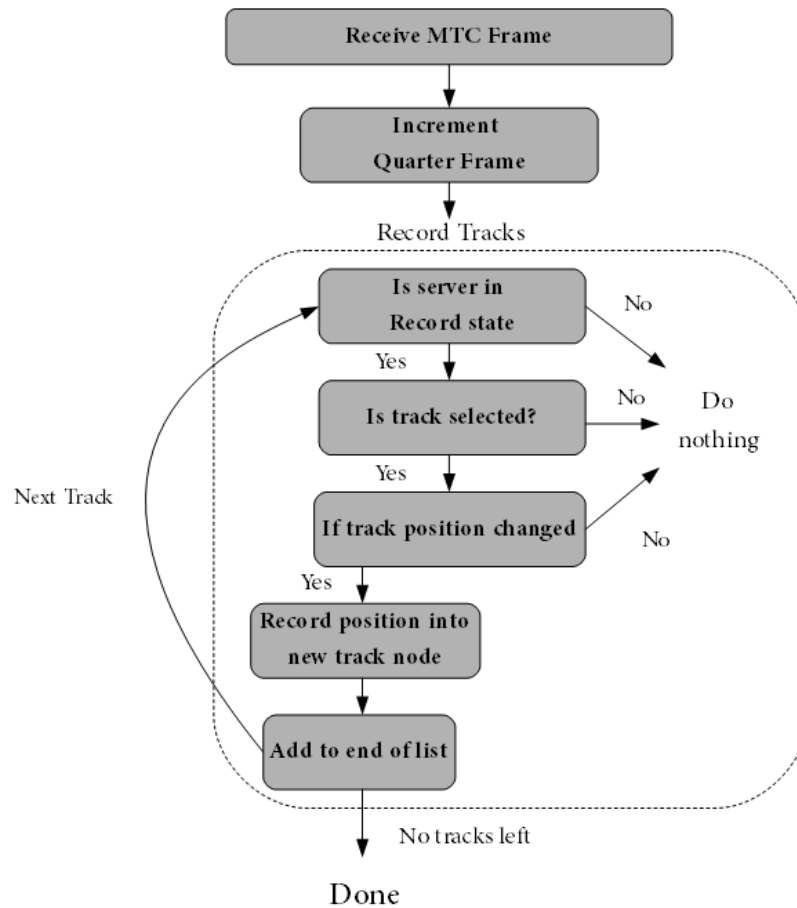


Fig. 11: Recording automation.

Once the recording is completed, it may be stored in an XML formatted “.imm” file to be played back at a later stage using the load button on the client interface.

2.2.6.4 Automation Playback

The automation playback process is similar to the recording process. Fig. 12 below shows how automation that has been recently recorded, or read from a saved recording, may be played back.

1. First, an MTC quarter frame is received.
2. If the server is in playback state and the track is not selected, it will update the speaker mix levels for the track according to its coordinates at that quarter frame.
3. This is repeated for every track before the next quarter frame is received.
4. If track nodes do not exist beyond the current quarter frame, the track retains its coordinates as set by the user or a quarter frame from earlier in the automation.

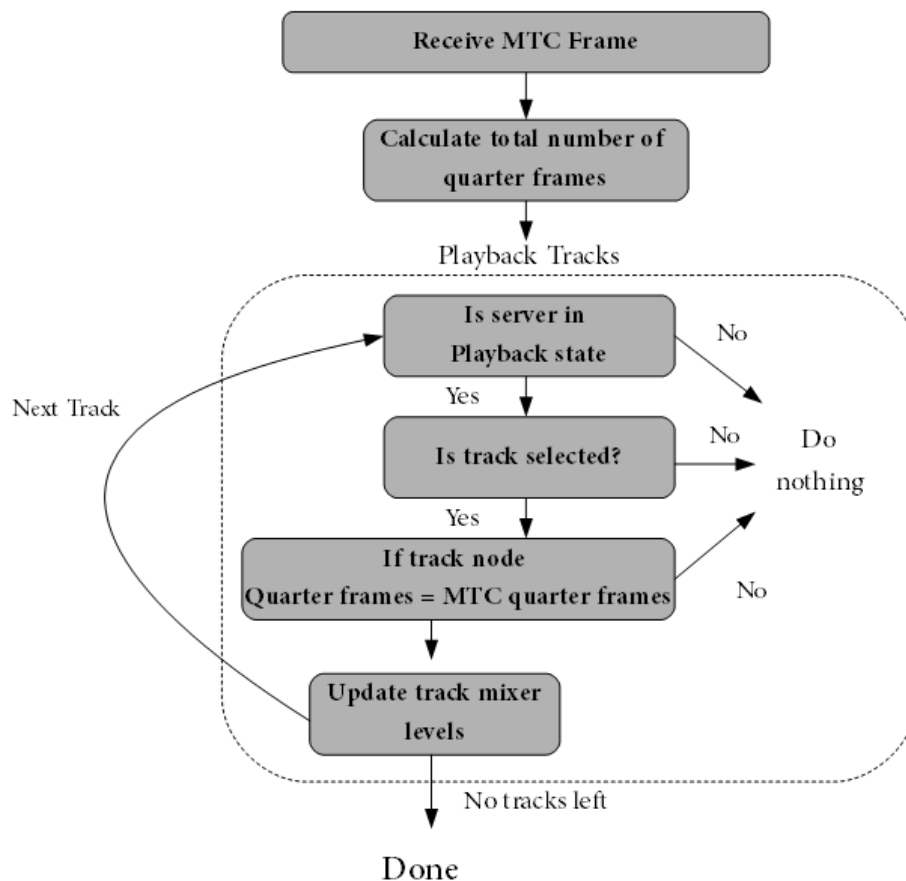


Fig. 12: Playing automation.

2.2.7 ImmerGo Speaker Endpoint Processors

ImmerGo is currently able to control a number of different AVB devices which are built by miniDSP ¹⁶. The SPK-4Ps have been provided by miniDSP for this research and are the devices that will be used in this project. The endpoints themselves will be discussed in greater detail in the following chapter, but for now it is enough to know that the SPK-4Ps are able to drive 2 loudspeakers, one active speaker and a second passive speaker with its passive output. It is able to receive an AVB stream of up to 32 channels. Each of these devices has a SHARC DSP chip and an XMOS chip programmed with firmware that includes a mixer matrix and AVDECC handler for control messages sent from ImmerGo. The SHARC DSP chip may be controlled using the XMOS processor although this has not been done prior to this project.

¹⁶ NDAC-8 - <https://www.immersivedsp.com/products/n-dac8>

NDAC-2 - <https://www.immersivedsp.com/products/ndac-2-poe>

SPK-4P - <https://www.immersivedsp.com/products/spk-4p>



Fig. 13: The SPK-4P active speaker.

2.2.8 Controlling the Speaker Endpoint with ImmerGo

Each endpoint mixer is controlled by sending an AVDECC AECP SET_MIXER packet that has a descriptor index of 0x0060 followed by the mixer data. **Table 4** shown earlier describes the template AECPDU packet that was used by the sending mechanism. The sending mechanism is illustrated in **Fig. 14** below. When ImmerGo sends mixer updates, the AECPDU is copied and its destination and source headers are changed according to the endpoint MAC for the SPK-4P it's sending to. The length of the data field is updated according to the length of the parameters used to control the mixer matrix. For a 32x2 mixer matrix, the length would be 16 + 64 bytes, with the 16 bytes used for the header fields and the 64 bytes needed for each crosspoint of the mixer matrix. The mixer matrix parameters are added sequentially to the data parameters field. Once added, the packet is sent to the endpoint using pcap¹⁷ and the process starts again for the next endpoint until they have all been updated.

¹⁷ <https://www.npmjs.com/package/pcap>

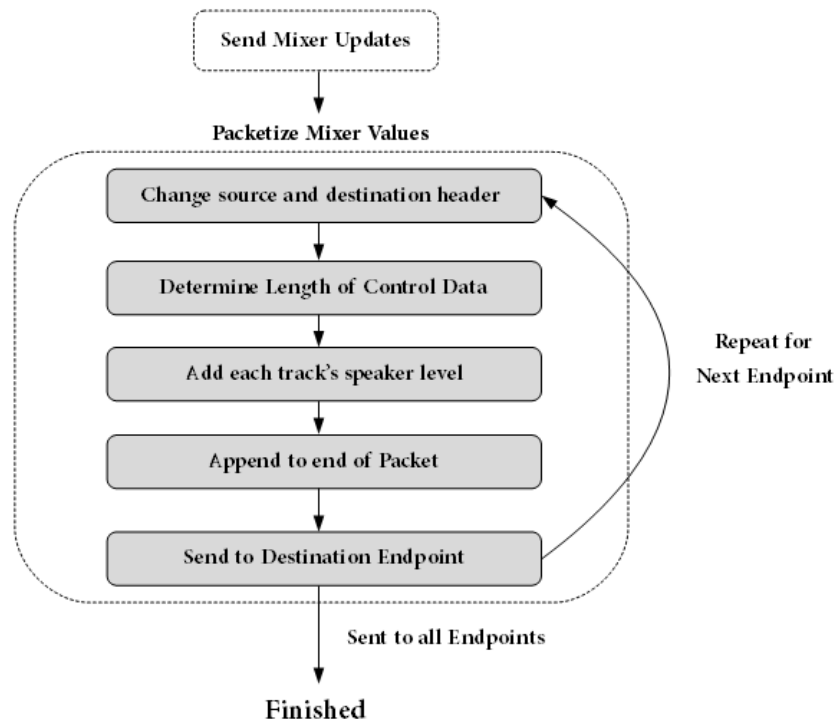


Fig. 14: Sending endpoint mixer updates from ImmerGo.

2.3 Summary

This chapter discussed the ImmerGo system and how it utilizes Ethernet AVB to control endpoint processors on the network. It covered the spatial audio rendering algorithm, DBAP, that is used within the ImmerGo server, as well as the mechanism used to synchronize the sending of mix levels to the endpoint processors for track localization. The following chapter will discuss how the endpoint processors were able to mix the audio as well as route and unpack the mixer control commands sent by ImmerGo.

Chapter 3

Speaker Endpoint Processors

The endpoint processors used in this project required an interface to the Ethernet AVB network and the capability to perform arithmetic and DSP operations. As described in the previous chapter, one such device is the SPK-4P built by miniDSP. Each SPK-4P has an internal configuration that can be seen in **Fig. 15**. The main components are the XMOS and SHARC DSP processors, as well as a Power Over Ethernet Plus (POE+) module that allows the device to be powered with an AVB capable POE+ switch. The XMOS processor has been programmed as an AVB listener. It manages the buffering of audio that is received from the network as well as any AVDECC commands. Attached on the same PCB is the SHARC DSP chip which is controllable using commands given from the XMOS chip passed over the I²C data line. The XMOS receives audio from the AVB network and then passes two channels to the SHARC DSP over an I²S bus with two data lines. The audio signals are then passed through the SHARC chip and sent out to the two class D amplifiers. One of the amplifiers is dedicated to the active speaker, the other is attached to an output port that is used to power an external speaker. Prior to this project, ImmerGo only made use of the XMOS processor to perform the mixing required to localize track sources, whilst bypassing the SHARC DSP chip. This chapter will discuss how this was achieved, and in doing so, will provide information about all the necessary mechanisms for an extension to deeper levels of distribution.

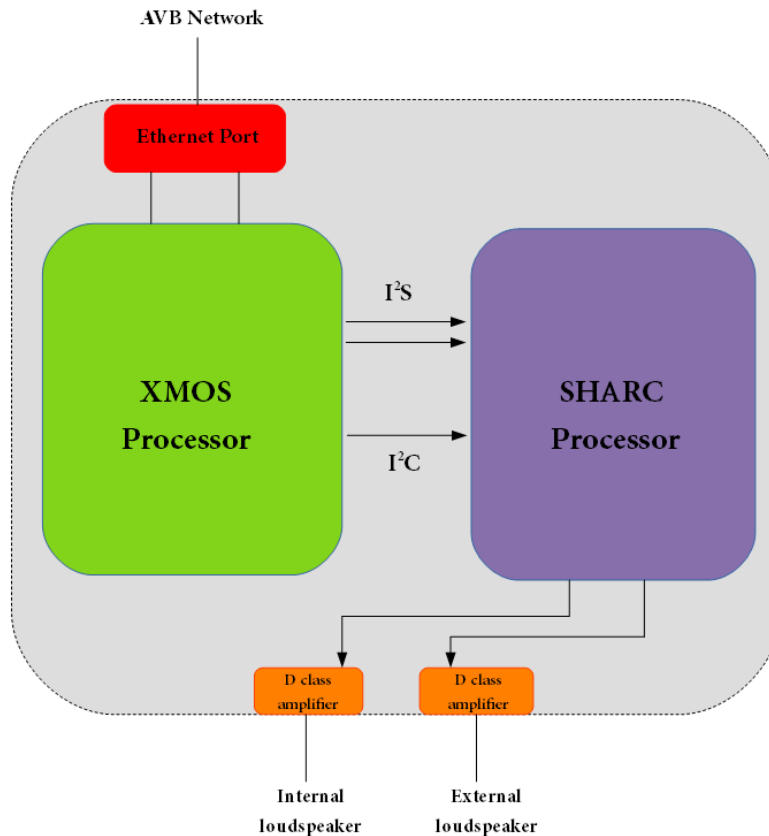


Fig. 15: SPK-4P internal configuration.

3.1 XMOS Processor Architecture

The XMOS processors found in the SPK-4P have the XCORE-200 XE architecture¹⁸, which is designed to allow the programming of deterministic and concurrent, real-time applications. The processor has hardware operating system components known as xTIME technology. The xTIME technology incorporates the process scheduler, timer and memory manager, and is designed to allow real-time parallel processing applications to be predictable. Fig. 16 shows the layout of the processor architecture.

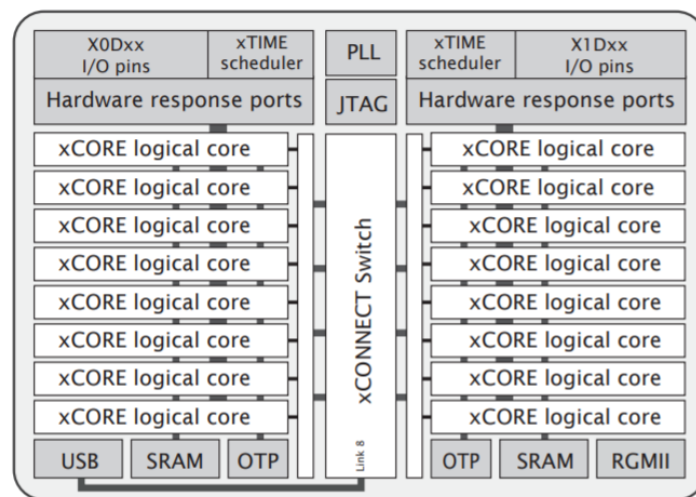


Fig. 16: XMOS XCORE-200 processor architecture.

As shown in Fig. 16, the XCORE-200 processor is split into two processing tiles connected with the xCONNECT switch. Each tile holds 8 logical cores that share the tile's processing resources and memory (SRAM) but also have their own registers. As well as this, each tile also has its own I/O pins that are accessed through the hardware response ports. The JTAG provides a header that can be used to upload firmware from the XMOS SDK and the one-time programmable (OTP) memory contains the board specific information. The universal serial bus is available to be used but is not currently used within ImmerGo. The RGMII (Reduced Gigabit Media Independent Interface) is used to received data coming from the Ethernet AVB network.

The processor runs 500MIPS at a cycle rate of 500Mhz and the xTIME timing and synchronization technology ensures that each logical core gets up to 125MIPS of computing bandwidth without blocking the other cores. For programmers, this means that each of the logical cores appears as its own separate parallel processor capable of running multiple tasks simultaneously. tasks on different tiles use the xCONNECT switch to pass data to one another. The xTIME scheduler on each tile ensures that tasks on each tile, and their communication, have their timing predictable according to the phase-locked loop clock frequency (PLL). This makes them very suitable for robust real-time application development and production.

¹⁸ <http://www.xmos.com/products/silicon/xcore-200/xs-series>

Every XMos chip that is manufactured has its architecture information provided within an XML formatted .xn file. This file is used by the program compiler to build the firmware correctly for the processor. The SPK-4P architecture model is given in the “D4U-miniDSP2x4.xn” file within the XMos SDK project. As shown in **Listing 5**, this file holds the chip specific information with the name, location and function of each port, as well as the clock cycle settings and information.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <Network xmlns="http://www.xmos.com" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.xmos.co
3   <Type>Board</Type>
4   <Name>XS2 MC Audio</Name>
5   <Declarations>
6     <Declaration>tileref tile[2]</Declaration>
7     <Declaration>tileref usb_tile</Declaration>
8   </Declarations>
9   <Packages>
10    <Package id="0" Type="XS2-UnA-512-FB236">
11      <Nodes>
12        <Node Id="0" InPackageId="0" Type="XS2-L16A-512" Oscillator="24MHz" SystemFrequency="500MHz" referencefrequency="100MHz">
13          <Boot>
14            <Source Location="bootFlash"/>
15          </Boot>
16          <Tile Number="0" Reference="tile[0]">
17            <!-- QSPI for internal Flash -->
18            <Port Location="XSI_PORT_1B" Name="PORT_SQI_CS"/>
19            <Port Location="XSI_PORT_1C" Name="PORT_SQI_SCLK"/>
20            <Port Location="XSI_PORT_4B" Name="PORT_SQI_SIO"/>
21
22            <!-- SPI -->
23            <Port Location="XSI_PORT_1A" Name="PORT_SPI_MISO"/>
24            <Port Location="XSI_PORT_1C" Name="PORT_SPI_CLK"/>
25            <Port Location="XSI_PORT_1D" Name="PORT_SPI_MOSI"/>
26
27            <!-- Audio Ports -->
28            <Port Location="XSI_PORT_1E" Name="PORT_I2S_ADC0"/>
29            <Port Location="XSI_PORT_1F" Name="PORT_I2S_DAC0"/>
30            <Port Location="XSI_PORT_1G" Name="PORT_I2S_ADC1"/>
31            <Port Location="XSI_PORT_1H" Name="PORT_I2S_DAC1"/>
32            <Port Location="XSI_PORT_1I" Name="PORT_I2S_LRCLK"/>
33            <Port Location="XSI_PORT_1O" Name="PORT_I2S_BCLK"/>
34            <Port Location="XSI_PORT_1P" Name="PORT_MCLK_IN"/>
35
36            <!-- TDM Ports Same as I2S -->
37            <Port Location="XSI_PORT_1E" Name="PORT_TDM_ADC0"/>
38            <Port Location="XSI_PORT_1F" Name="PORT_TDM_DAC0"/>
39            <Port Location="XSI_PORT_1G" Name="PORT_TDM_ADC1"/>
40            <Port Location="XSI_PORT_1H" Name="PORT_TDM_DAC1"/>
41            <Port Location="XSI_PORT_1I" Name="PORT_TDM_FS"/>

```

Listing 5: Device architecture structure contained within the D4U-miniDSP2x4.xn file.

3.1.1 XMOS tasks

As mentioned, XMOS tasks are dedicated to a logical core on either of the tiles. To help understand this, a basic layout of the XMOS program tasks and their underlying relationship to the architecture is given in **Fig. 17**. Each Individual processing task is allocated its own logical core that is housed on a tile. The tile is connected to its associated I/O ports and timers. The tasks pass data to one other regardless of whether they are on the same tile, using communication abstractions (xCONNECT) known as channels and interfaces. The I/O pins, which are sometimes referred to as General Purpose I/O (GPIO) pins, may be used to send data in and out of the chip through the hardware response ports. There are several timers which may be instantiated and used by processing tasks. tasks react to events such as data received from I/O pins, messages from other tasks or timer events.

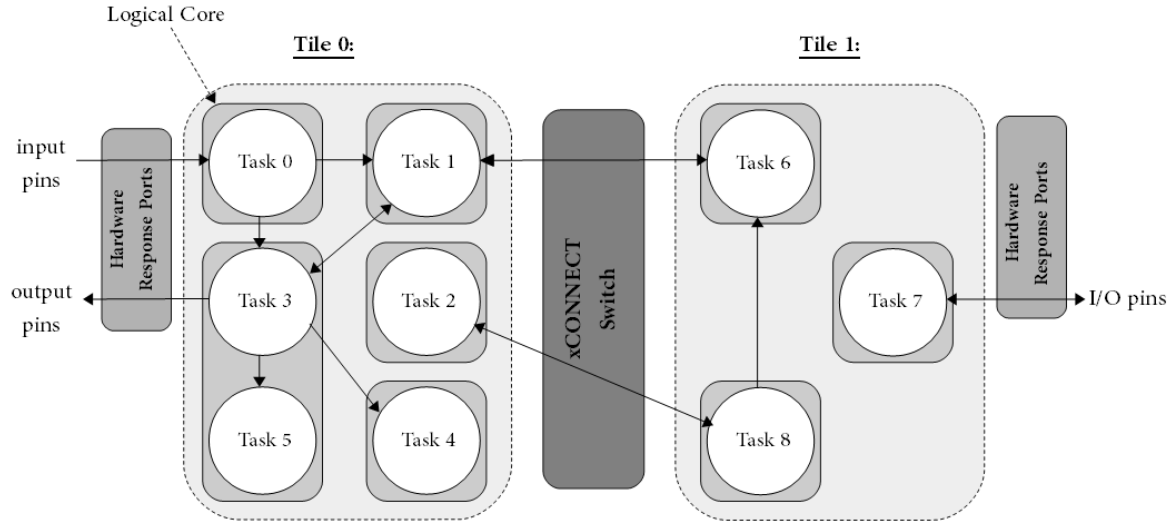


Fig. 17: Tiles and tasks.

The XMOS development kit uses the xC language and its associated xcc compiler. The language xC is similar to C but enforces certain restrictions and implements new data types that are required for the concurrent architecture. The xC tasks get compiled into a binary before being flashed to the processor memory using a JTAG connection. The following sections will describe various functionality provided by the XMOS processors. As shown in Fig. 17, each task runs as an independent function that performs processing within its own scope and can be passed data from the scope of other processing tasks. The limit to how many tasks can be created is based on the logical cores available in each tile, and this limit is shown within the XMOS SDK.

3.1.2 XMOS Pin Allocation

I/O pins for both tiles are addressable as “in ports” or “out ports”. Timers may also be instantiated, of which some are dedicated to clocked ports, or “buffered ports”. A buffered port allows for data to be stored in a buffer and sent out according to a timer, so the processor can continue with other functions. This functionality is particularly useful for the transmission of audio samples which use a fixed sample rate.

Listing 6 shows some of the important ports that are instantiated within the main file of the xC project used by the endpoints in this thesis. The buffered ports are used for audio sample transmission to and from the I²S driver. The clocks on tile 0 are used to transmit the master and bit clock signals out of the processor to synchronize the SHARC processor. The in port is used to receive the clock signal from the XMOS processor master clock. The out ports on tile 1 are used to control the LEDs connected to the SPK-4P.

```

1 on tile[0]: out buffered port:32 p_aud_dout[AVB_NUM_MEDIA_OUTPUTS/CHANNEL_PER_WIRE] = {PORT_TDM_DAC0, PORT_TDM_DAC1};
2 on tile[0]: in buffered port:32 p_aud_din[AVB_NUM_MEDIA_INPUTS/CHANNEL_PER_WIRE] = {PORT_TDM_ADC0, PORT_TDM_ADC1};
3 on tile[0]: clock clk_tdm_bclk = XS1_CLKBLK_3;
4 on tile[0]: clock clk_tdm_mclk = XS1_CLKBLK_4;
5 on tile[0]: in port p_tdm_mclk = PORT_PLL_MCLK;
6 on tile[1]: out port p_leds_row = XS1_PORT_4C;
7 on tile[1]: out port p_leds_column = XS1_PORT_4D;

```

Listing 6: Port declaration within xC.

3.1.3 XMOS Inter-task Communication

Inter-task communication is handled using channels or interfaces. Channels connect parallel tasks and may be either bidirectional or unidirectional. Channels typically pass data synchronously using a handshaking method, which generally slows code on the receiving end. This handshaking process may be avoided on every transaction by using streaming channels. A streaming channel creates a permanent connection between two tasks at startup. This is so the XMOS processor knows data will be streamed continuously on the channel. **Listing 7** shows a streaming channel created and given to two tasks that are allocated to a single core. The infinite while loop in task 1 creates an integer with a value 200 and sends it to task 2 using the streaming channel. task 2 receives this integer using a case within a switch statement and prints the integer value.

```

1 void task1(streaming chan c_basic){
2     int send_int = 200;
3     while(1){
4         // Send ints
5         c_basic <: send_int;
6     }
7 }
8
9 void task2(streaming chan c_basic){
10    int received_int;
11    // Receive an int
12    select{
13        case c_basic :> received_int:
14            printintln(received_int);
15            break;
16    }
17 }
18 }
19
20 int main(){
21     streaming chan c_basic;
22     par {
23         task1(c_basic);
24         task2(c_basic);
25     }
26     return 0;
27 }

```

Listing 7: Channels between two tasks within xC.

Channel communication can become confusing when passing various types of data between the two processing tasks. The interface type attempts to solve this by providing a more verbose method of passing data unidirectionally. An interface is defined with a set of functions, and is given to both tasks intending to send data to one another. The process receiving data is given the role of a server and the process sending data is given the role of the client. This is demonstrated in **Listing 8** below. The interface functions are defined at the top of the file. This interface is then instantiated within the main function. task 1 is given the client side of the interface and task 2 is given server-side. task 1 makes a call first to the *pass_integers* function and then the *pass_float* function of the interface client to send integers 5 and 10 and float 0.5 to the server on task 2. task 2 then prints the received integers and float using a case statement within the interface server.

```

1 interface i_pass_parameters {
2     void pass_integers(int x, int y);
3     void pass_float(float x);
4 };
5
6 void task1(client interface i_pass_parameters i_client)
7 {
8     i_client.pass_integers(5, 10);
9     i_client.pass_float(0.5);
10 }
11
12 void task2(server interface i_pass_parameters i_server)
13 {
14     select {
15         case i_server.pass_integers(int x, int y):
16             printf("Received: %d, %d\n", x, y);
17             break;
18         case i_server.pass_float(float x):
19             // handle the message
20             printf("Received: %f\n", x);
21             break;
22     }
23 }
24
25 int main() {
26     interface i_pass_parameters i_tasks;
27     on tile[0]: task1(i_task);
28     on tile[1]: task2(i_task);
29
30     return 0;
31 }

```

Listing 8: Interface between two tasks within xC.

3.1.4 XMOS Analysis Tools

The XMOS integrated development environment (IDE) has tools necessary for visualizing and profiling processor firmware. The timing analyzer tools are aptly named XMOS Timing Analysis (XTA) tools. These tools allow for various functions and data routes to be traced and timed in a simulation. XTA provides the diagnostics to ensure the processor's application code is able to meet real-time demands prior to being flashed onto the chip. As shown later in this thesis, the XTA tool is used extensively to profile the spatial audio rendering calculations that are implemented within the XMOS processor.

3.2 ImmerGo's XMOS Processor

The SPK-4P XMOS processors contain a version of an openly available AVB implementation that is modified to contain a mixer matrix. This mixer is addressable using the AVDECC AECP SET_MIXER commands sent by ImmerGo. Shown in **Fig. 18** is the XMOS AVB firmware used by ImmerGo. The control path is highlighted by the finely dashed, purple arrows and the audio path is highlighted by the dashed, green arrows. The ImmerGo controlled mixer matrix is located within the Audio Buffer Manager task. The audio is routed into the SHARC audio input, however, all SHARC processing has been bypassed. Tile 1 manages the interface from the physical port to the Ethernet Media Access Control (MAC) whilst tile 0 holds the various components of the AVB implementation and audio processing.

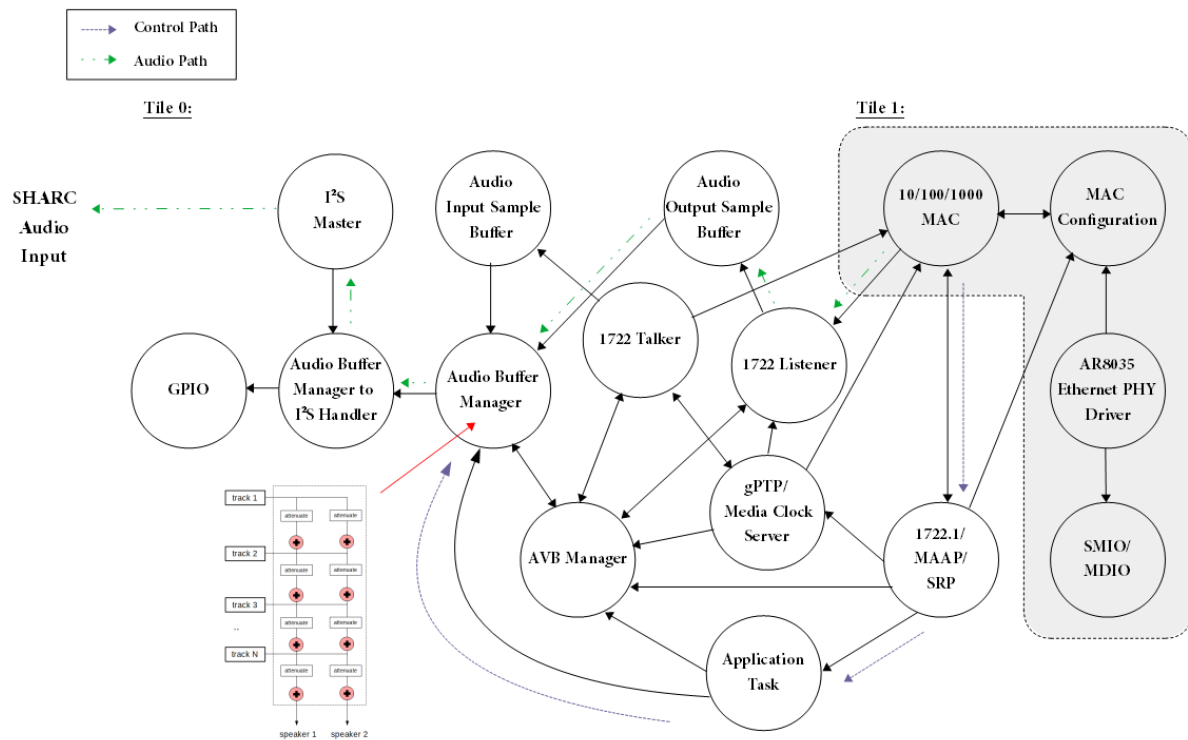


Fig. 18: ImmerGo XMOS AVB firmware with Mixer Matrix.

The audio may be buffered onto the I²S data lines using either the I²S driver or the TDM driver. The I²S buffer allows 2 channels to be transmitted, while the TDM driver allows 32 channels which is important for the SHARC processing of all channels received from the AVB stream.

This project's development process was concerned primarily with modifications to the XMOS mixer matrix and the mechanism that controlled it. As such, in depth discussion into the other components will be omitted, however, the reader is directed to the documentation that describes the XMOS AVB implementation if they'd like a complete understanding of each component ¹⁹.

3.2.1 The Audio Path

Fig. 19 shows the path of the audio samples from the AVB network through the Audio Buffer Manager to the I²S lines. Audio samples are received as AVTP packets by the Ethernet MAC task. These are then passed to the 1722 Listener task which unpacks the AVTP packet and pushes the samples into the media FIFOs (First in First out) buffers held within the Audio Sample Output Buffer task. The Audio Buffer Manager task then runs an infinite loop which retrieves the audio samples from the FIFOs and then sends them to the I²S driver using the Audio Buffer Manager to I²S handler. The mixer matrix is inserted into the Audio Buffer Manager task between where the samples are pulled from the FIFO and sent to the I²S master.

¹⁹ <https://www.xmos.com/developer/published/avb-design-guide?page=4>

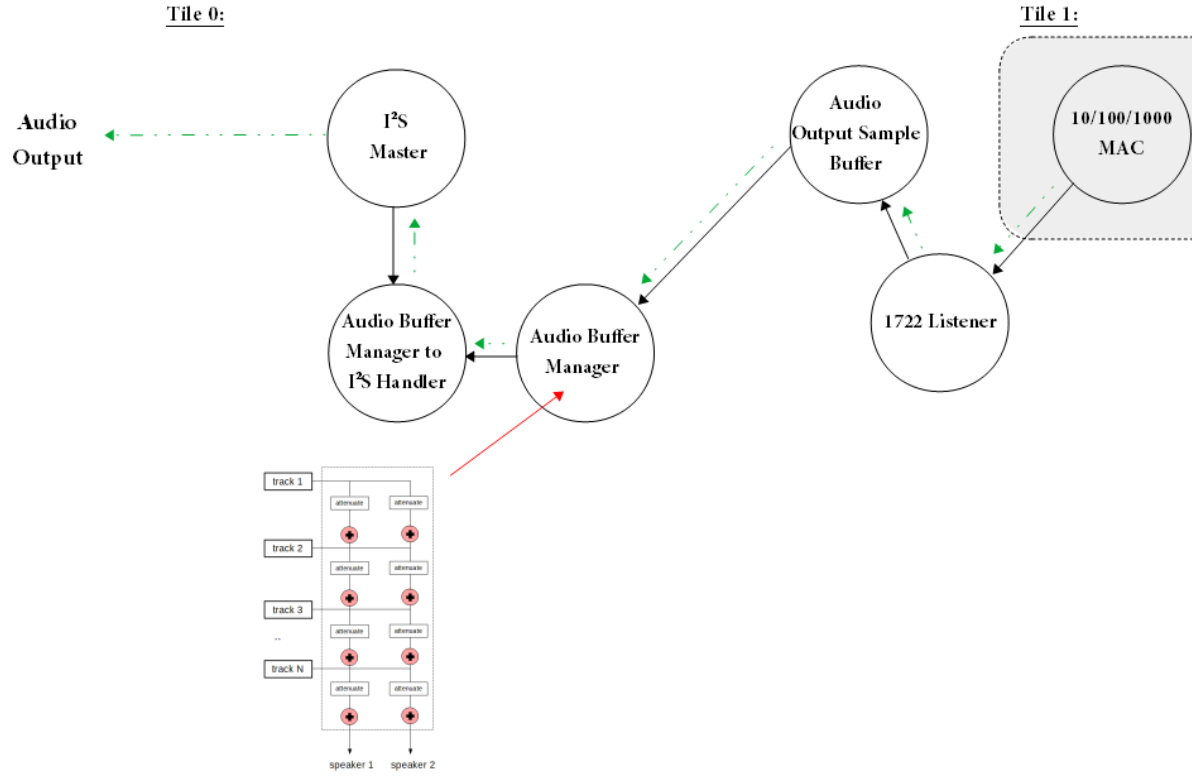


Fig. 19: Path of audio through the Audio Buffer Manager task.

Before discussing how the mixer matrix is incorporated, the method used by Audio Buffer Manager to pull samples from the FIFO is discussed using an example of an 8 channel AVB stream described in Fig. 20. The 1722 Listener task fills the media FIFO within the Audio Output Sample Buffer task with 24 bit audio samples from the 8 channels according to their timestamp t_n . The channel samples within the FIFOs are interleaved such that the odd channel samples are filled in first, and the even channel samples filled second. The Audio Buffer Manager task runs an infinite while loop which then pulls the audio samples for each channel and fills a buffer with samples in their correct order. These samples are then passed to the I²S master. Since only 2 pins are available to transfer the channels, only the first 2 channels from the audio stream will be played with channel 1 going to loudspeaker 1 and channel 2 to loudspeaker 2.

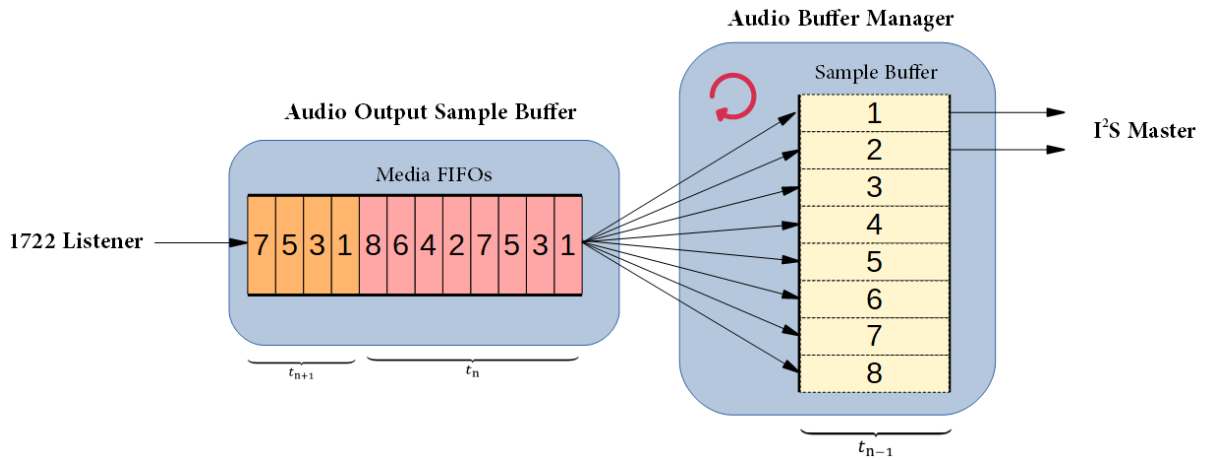


Fig. 20: Pulling samples from the FIFO and passing to the I²S master.

3.2.2 The ImmerGo Mixer Matrix

ImmerGo uses an attenuation mixer matrix that allows for channels from an AVB stream to have their volumes for each loudspeaker controlled prior to being summed together. As shown in **Fig. 21**, the inputs of the mixer matrix are set according to the number of audio channels the processor is able to receive while the outputs are set according to the two speakers attached to the SPK-4P. **Fig. 19** in the previous page shows where the mixer matrix lies within the Audio Buffer Manager task: in between the point that the samples are pulled from the FIFOs and sent to the I²S driver.

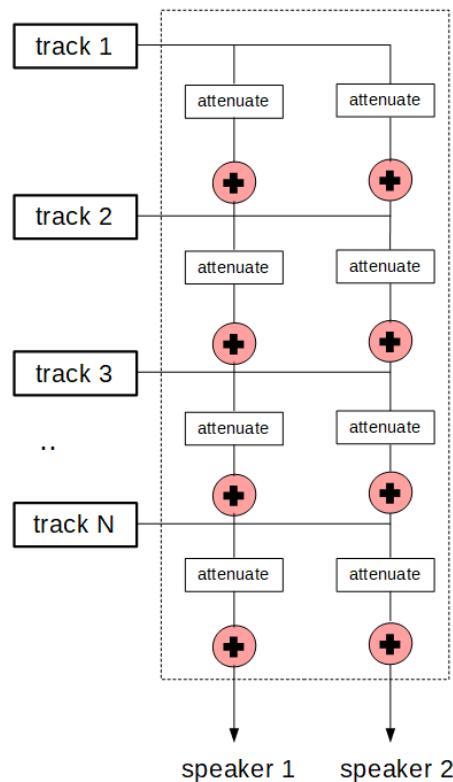


Fig. 21: Attenuate mixer matrix.

Listing 9 shows the function header for the Audio Buffer Manager task with the channels and interfaces that allow for the transmission of audio from the 1722 listener to the Audio Buffer Manager and then I²S data lines. The interfaces and channels are listed below, with the interfaces important for the audio mixing modification being bolded,

- **c_audio** – a channel that sends audio samples to the I²S master.
- audio_input_buf – Pushes audio samples onto double buffer to be sent to the AVB network.
- **audio_output_buf** – Pulls samples from the media FIFO.
- c_media_ctrl – Used to set sample rate and control options for the Audio Buffer Manager
- **audio_io_type** – tells the Audio Buffer Manager to use the I²S or time division multiplexing (TDM) as shown in **Listing 10**.

```

59 void audio_buffer_manager(streaming_chanend c_audio,
60                          client_push_if audio_input_buf,
61                          client_pull_if audio_output_buf,
62                          chanend c_media_ctrl,
63                          const audio_io_t audio_io_type);

```

Listing 9: Function header for the Audio Buffer Manager task.

```

56 typedef enum audio_io_t
57 {
58     AUDIO_I2S_IO,
59     AUDIO_TDM_IO
60 } audio_io_t;

```

Listing 10: I²S line settings.

Listing 11 shows the mixer matrix buffers for the input and output channels. These are defined within the Audio Buffer Manager task as 24 bit samples held within a 32 bit word. Also shown are the mixer matrix parameters which hold the attenuation amounts for the XMOS mixer.

```

129 // ... in audio buffer manager task
130 // XMOS mixer gain parameters
131 signed char XMOSMixerVals[MAX_SPEAKERS][MAX_CHANNELS]={0};
...
144 // Mixer matrix inputs
145 int32_t sample_out_buf[AVB_NUM_MEDIA_OUTPUTS] = {0};
146 // Mixer matrix outputs
147 int32_t samples_out[MAX_SPEAKERS+1] = {0};

```

Listing 11: XMOS mixer matrix variables within the Audio Buffer Manager task.

Listing 12 below shows how the mixer matrix was implemented within the XMOS firmware. First, the interleaved samples are pulled from the media FIFO which fills the sample buffer. These samples are then shifted right to create headroom for the mixer. The samples are then sent into the mixer matrix which scales them according to the values within the *XMOSMixerVals* array and sums to the correct speaker output.

```

181 // Infinite While Loop within Audio Buffer Manager ...
182
183 default:
184     unsafe {
185         if (audio_io_type == AUDIO_I2S_IO) {
186             /* Pull samples from Media FIFOs */
187             #pragma loop unroll
188             for (int i=0; i<AVB_NUM_MEDIA_OUTPUTS; i+=2) {
189                 sample_out_buf[i] = audio_output_fifo_pull_sample(h_out, i,
190                                                                 timestamp);
191                 sample_out_buf[i] = sample_out_buf[i] >> 8;
192             }
193             #pragma loop unroll
194             for (int i=1; i<AVB_NUM_MEDIA_OUTPUTS; i+=2) {
195                 sample_out_buf[i] = audio_output_fifo_pull_sample(h_out, i,
196                                                                 timestamp);
197                 sample_out_buf[i] = sample_out_buf[i] >> 8;
198             }
199
200             /* Mix the samples */
201             #pragma loop unroll
202             for(int i=0; i<MAX_SPEAKERS; i++)
203             {
204                 samples_out[i] = 0;
205                 #pragma loop unroll
206                 for(int j=0; j<MAX_CHANNELS; j++)
207                 {
208                     samples_out[i] += sample_out_buf[j]*XMOSMixerVals[i][j];
209                 }
210             }
211
212             c_audio <: (int32_t *unsafe)&samples_out;
213         }

```

Listing 12: XMOS attenuation-only mixer matrix in the Audio Buffer Manager task.

3.2.3 Controlling the XMOS Mixer Matrix

The mixer matrix parameters are updated according to the AVDECC AECP SET_MIXER commands sent by the ImmerGo server as described in the previous chapter. Fig. 22 shows how the AECP messages are routed through the firmware tasks after being received by the Ethernet MAC.

The AECP SET_MIXER command has its headers parsed within the 1722.1 task till the 0x0060 descriptor index. It then passes this descriptor index and the packet data field to the Application task using the *i_1722_1_entity* interface. The Application task has a set of switch statements which can have events triggered according to the descriptor index. In this case, the descriptor index 0x0060 tells the Application task to update the XMOS mixer matrix according to the data value within the packet. An internal interface *i_1722_1_mixer* between the Application task and the Audio Buffer Manager is then used to update the XMOS mixer matrix parameters.

The *i_1722_1_mixer* is an *avb_1722_1_mixer* interface which has a function that sets the XMOS mixer matrix values as shown in Listing 13.

```

19 interface avb_1722_1_mixer {
20     void set_mixer_values( unsigned short num_mix_values, char mix_Values[]);
21 };

```

Listing 13: Internal interface used to update the XMOS mixer matrix.

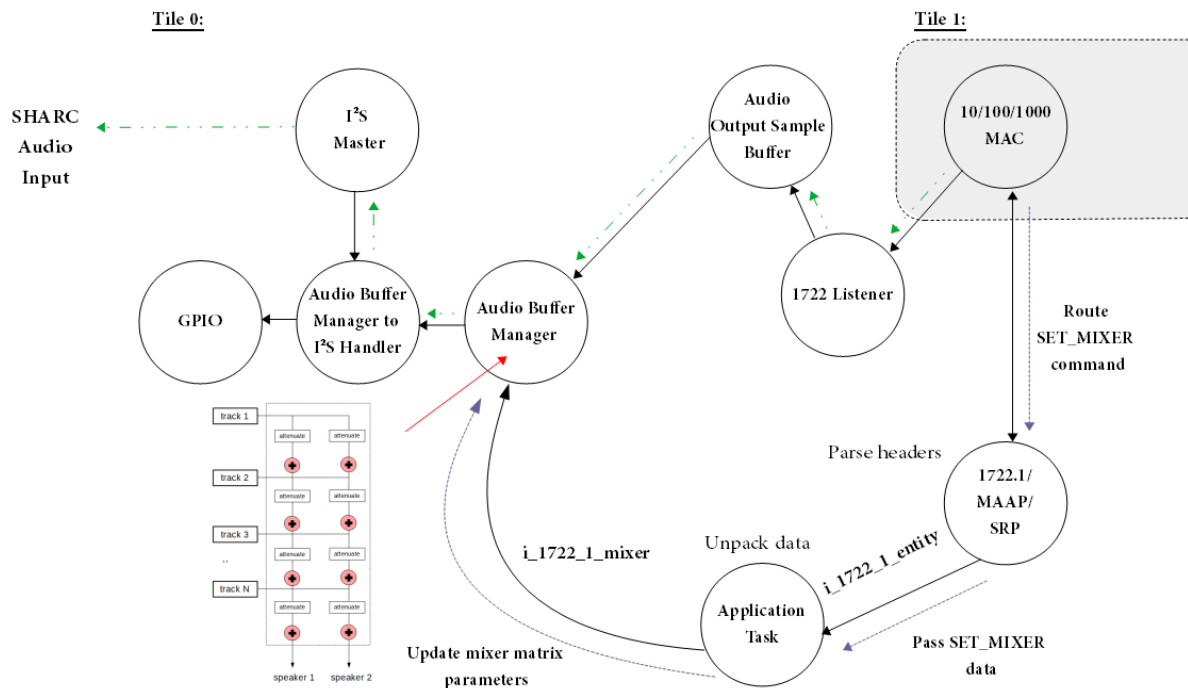


Fig. 22: Controlling the XMOS mixer matrix.

As shown in Listing 14, when the Application task receives the data, it unpacks all the mixer matrix values and then makes a call containing the mixer values with the `i_1722_1_mixer` client. This is received by the `i_1722_1_mixer` server within the Audio Buffer Manager task which receives the values and updates the XMOS mixer matrix values as demonstrated in Listing 15.

```

495 // In the application task ...
496 case 0x60: { // Set XMOS Mixer Matrix
497     char mix_Values[64];
498     for (int i = 0; i < values_length; i++) mix_Values[i] = va;
499     i_1722_1_mixer.set_mixer_values(values_length, mix_Values);
500     return_status = AECP_AEM_STATUS_SUCCESS;
501     break;
502 }

```

client →

Listing 14: Passing the mixer matrix values from the Application task to the Audio Buffer Manager.

```

369 // In the Audio Buffer Manager infinite while loop ...
370 case i_1722_1_mixer.set_mixer_values(unsigned short num_mix_values, char mix_Values[])
371     for (int i=0; i < MAX_SPEAKERS; i++) {
372         for (int j = 0; j < MAX_CHANNELS; j++) {
373             XMOSMixerVals[i][j] = mix_Values[index++];
374         }
375     }
376     break;
377 }

```

server →

Listing 15: Receiving the mixer matrix updates.

3.3 Summary

This chapter reviewed the architecture of the endpoint processor used within the ImmerGo system. A discussion of the style of programming required to program the XMOS processor provided information on the mechanisms used by ImmerGo to implement the first level of endpoint distributed processing. This discussion, in particular, described from the endpoint perspective how the mixer matrix was updated with

values outputted from the DBAP algorithm. To do so, a brief overview of the AVB firmware that is used by the SPK-4P was given. This highlighted key areas within the XMOS processor that were necessary for the implementation within this thesis' project. The following chapter will discuss how the mechanisms used by ImmerGo were extended to distribute more processing to the speaker endpoints. These extensions provide a framework that allows for more algorithms to be implemented.

Chapter 4

Modifications to ImmerGo

Modifications were required for ImmerGo to incorporate further spatial audio rendering algorithms, and to then have those algorithms distributed to the endpoints. The object-based representation and AECN control mechanism used to control the endpoints served as a conduit for controlling these algorithms and enabling them to be more deeply distributed. To achieve the deeper levels of distribution, a new task was introduced within the SPK-4P firmware called the “Spatial Audio Renderer” (SAR) task. This task acts as a container within the XMOS processor that responds to AECN control messages containing ImmerGo’s renderer definition models.

Since DBAP had already been implemented within the ImmerGo server, it was the first to have its calculations distributed. This chapter will discuss how the new XMOS SAR task implements the DBAP algorithm. First, there is a description of how the ImmerGo server was modified to send the speaker and track object positions to the endpoint processor instead of the speaker mix levels. This is followed by an overview of the new firmware developed for the XMOS, with a description of how this firmware is able to receive the speaker and track locations and use them to perform the DBAP algorithm internally. Finally, there is a description of how further distribution was achieved utilizing the SHARC DSP processor attached to the XMOS.

4.1 Modifying the ImmerGo Host to Send Audio Definition Models

It was found that the DBAP calculations could become very processor intensive on the host as the number of speakers and tracks being localized increased. In order to mitigate this, the calculations were moved to the speaker endpoints. ImmerGo is now modified to send not only the mixer levels, but also the speaker positions, track data, and as later discussed, algorithm specific configuration data to the SAR incorporated within the XMOS. To achieve this, new packet templates were created for these data structures collectively termed the “Audio Definition Models”.

As discussed in chapter 2, there are two types of Audio Definition Model used by ImmerGo,

- The Room Definition Model
- The Track Object Model

This section will describe how these were sent from the ImmerGo server to the endpoints.

4.1.1 The ImmerGo Object Model Transmission Template

ImmerGo is required to send data from the server to the SAR within the endpoint processors. This is done using the NPM cap package, which is able to send AVDECC commands to the endpoint processors. The server uses the general packet structure for the AECp SET_MIXER command shown in **Table 9** below. This packet layout is created as a template within the server code. This template is then able to be copied and modified according to what needs to be sent to each endpoint device.

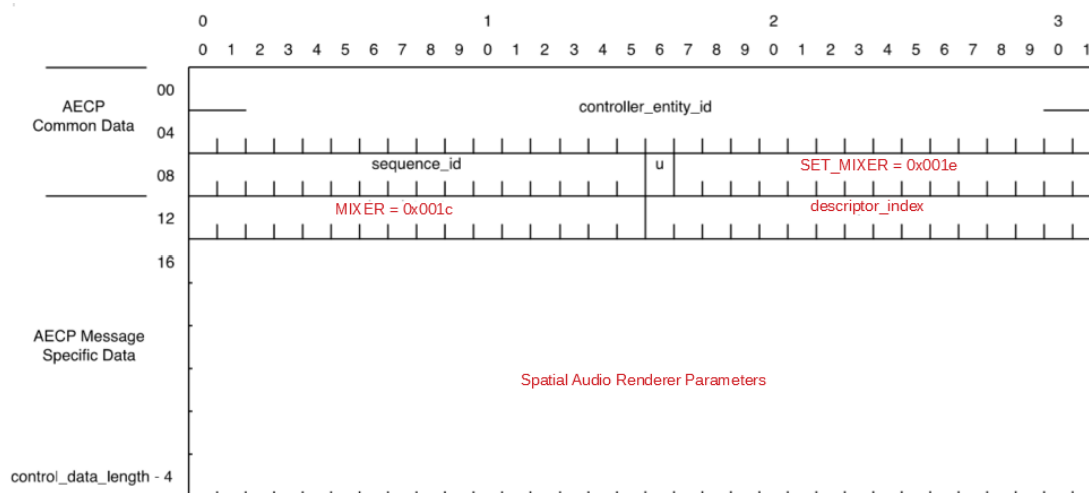


Table 9: AECp packet structure for the spatial audio renderer.

4.1.2 Sending the ImmerGo Room Definition Model to Endpoints

The Room Definition model is sent to the endpoint processors after each endpoint device in the speaker configuration XML has been discovered. **Fig. 23** below shows a typical speaker configuration that might be associated with the XML file. Each processor is addressed with its particular MAC address and an ID associated with the speakers that are connected to it. When calculating an algorithm internally, the speaker ID ensures that the mixer matrix is updated with the correct value pertaining to that speaker.

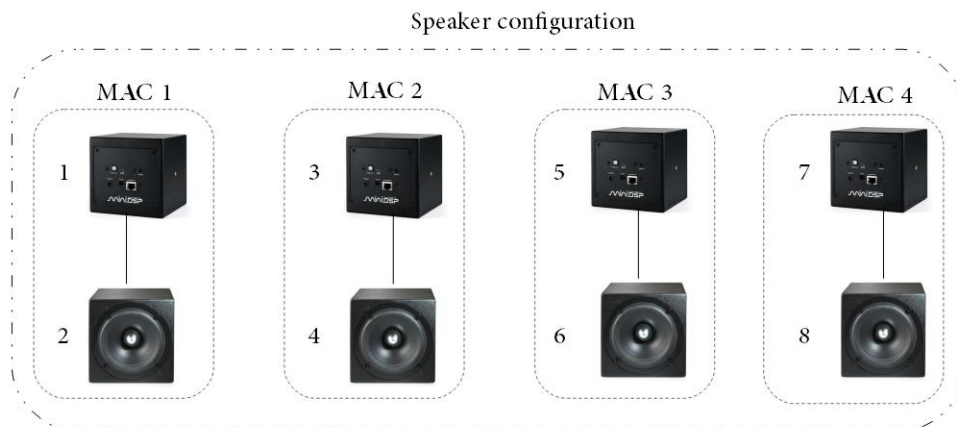


Fig. 23: Speaker Configuration with their associated MAC address and unique Speaker IDs.

The AECp SET_MIXER packet that is used to send the speaker configuration is shown in **Table 10** below. In the first two bytes, the IDs associated with the speakers attached to the MAC are given. The speaker

configuration that may be sent is limited to 255 speakers and the coordinates of each of these speakers need to be to each endpoint to be used by the DBAP calculation. Each coordinate has its (x, y, z) elements represented as 32 bit integers which give a range of 2^{32} cm for each coordinate and is packetized into 4 bit bytes, one after the other, with a final terminating integer of 0xFFFFFFFF. Each endpoint is then sent the configuration, along with the IDs of the two speakers it's attached to, within an AECP SET_MIXER command. The descriptor index is set to 0x0010. This descriptor index is important, as it tells the endpoint to set up the endpoint processor's internal speaker configuration with the correct speaker positions.

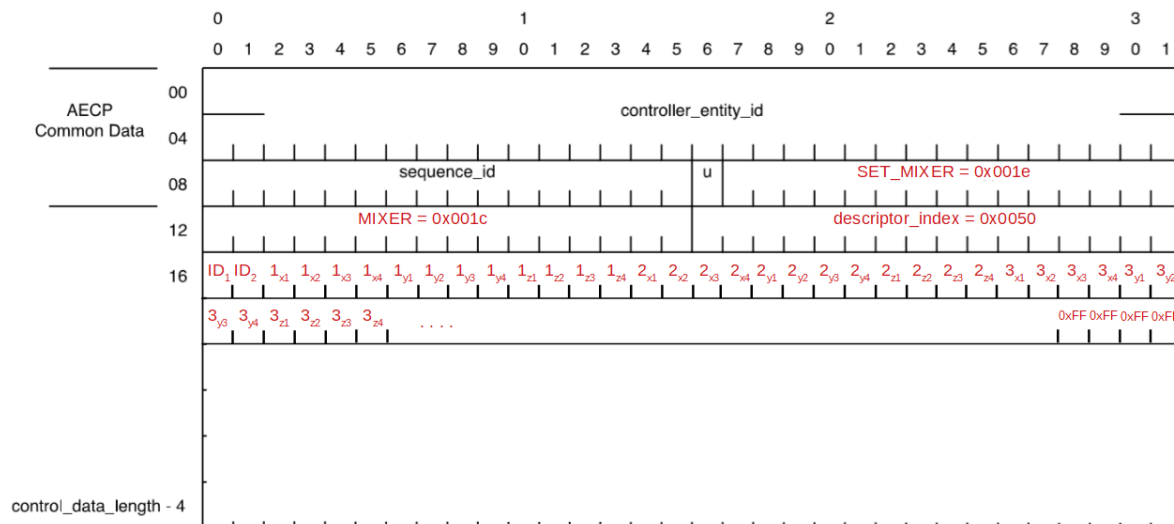


Table 10: AECP SET_MIXER packet holding the Speaker Configuration

Fig. 24 shows how this speaker configuration is sent and the code for this process is described in greater detail in Appendix A A3. The speaker configuration is first loaded into the server, and the endpoint devices in the configuration are discovered. Once discovered, the configuration may then be sent to each device. The speaker positions are first packetized into a buffer with each coordinate converted to 4 bytes. Then this data is added to an AECP packet with the IDs of the speaker connected to that endpoint.

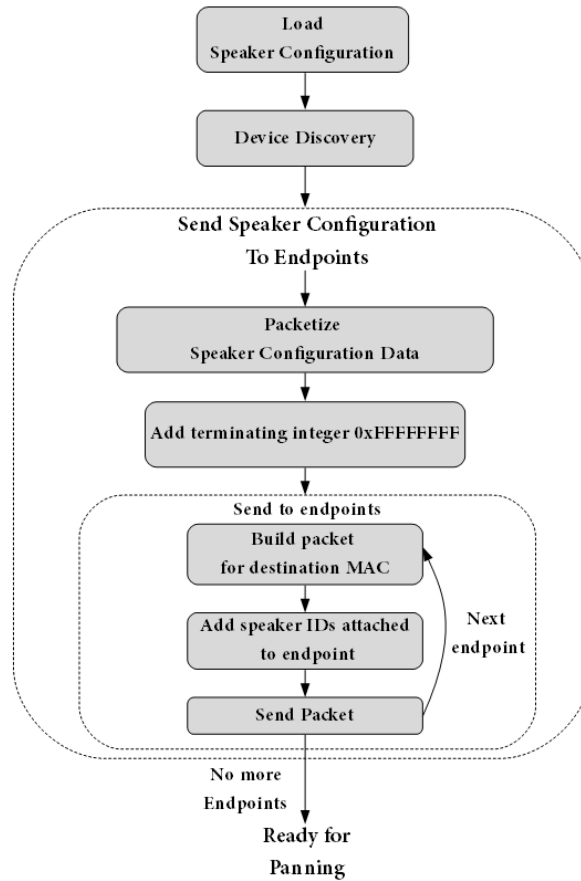


Fig. 24: Sending the Room Definition Model to the Endpoint processor

4.1.3 Sending the ImmerGo Track Model to Endpoints

A track queue has been created in the server to enable the updating of multiple sound sources according to the automation track as described in chapter 2. The queue ensures that track object updates are sent to the endpoints at every quarter frame of MTC. For every new quarter frame of MTC, there will be different tracks within the queue needing to have their object model updated since some track objects will be moved while others are not. This is shown in Fig. 25 below. When a track position is updated by either a user, or automation, its ID is added to the queue at the current quarter frame. IDs in the queue are then used by the sending mechanism to send the track object coordinates associated with the IDs to the endpoints for that quarter frame. This process is then continued for subsequent MTC quarter frames.

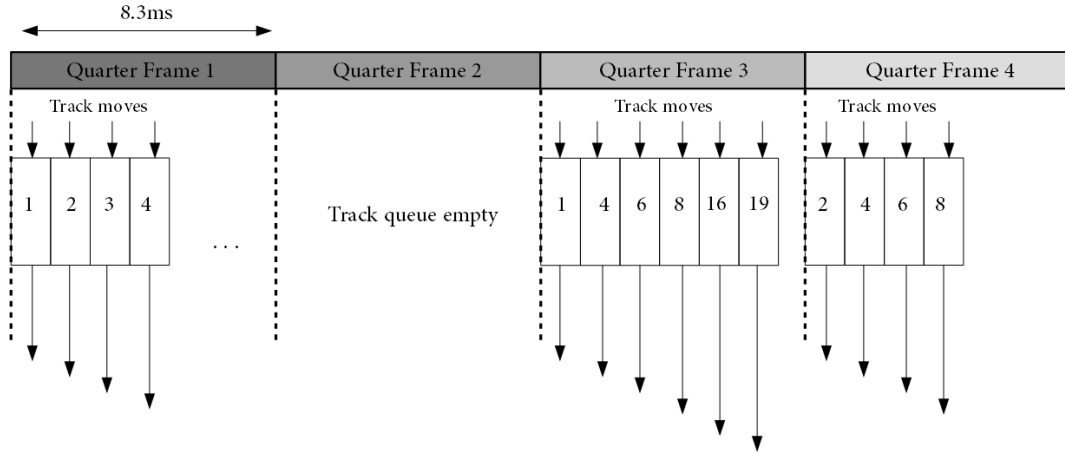


Fig. 25: Track queue updates for every interval

Fig. 26 shows the mechanism used to send track objects that are scheduled for updates to the endpoints and code is explained in greater detail in Appendix A A4. When a track source is moved either by a user action or automation, it has its object model updated with the new coordinates followed by its ID is added to an update queue. Now, every time the server receives an MTC frame the track update queue is checked. If it holds any track IDs, regardless of how many, the track IDs and coordinates associated with those IDs are bundled into a single packet. The first value of the packet holds the total number of tracks to be updated at that quarter frame. From here they are sent to each endpoint processor.

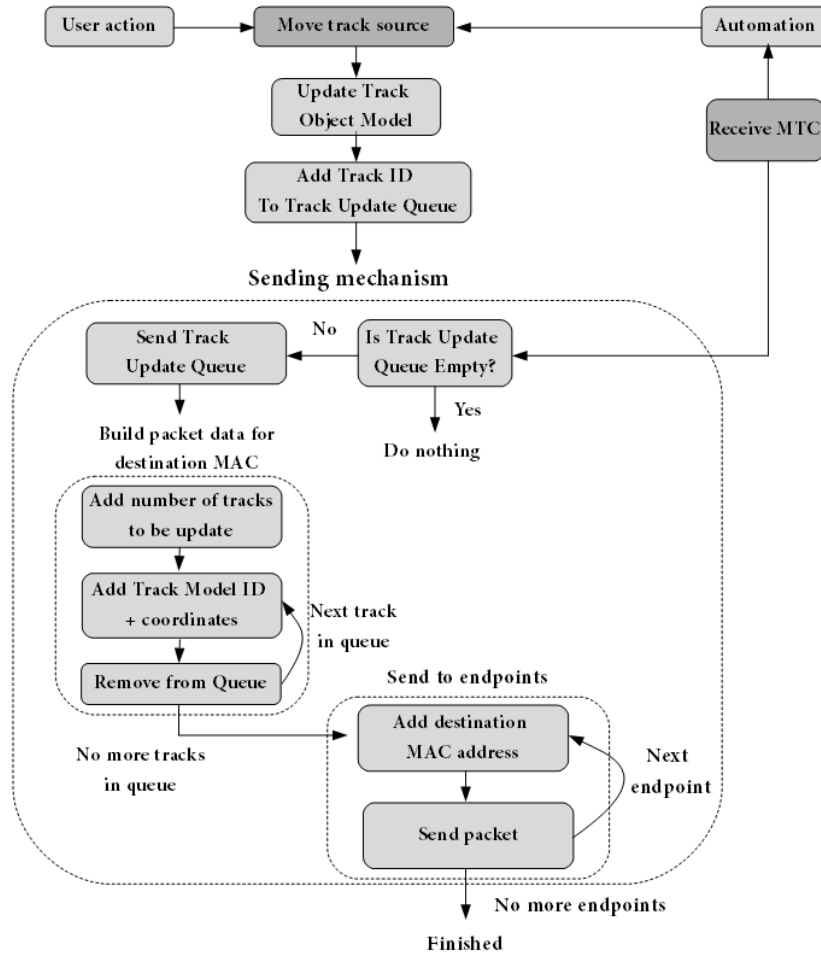


Fig. 26: Sending track object updates to the processor from the ImmerGo server.

4.2 Introduction to the Spatial Audio Renderer task

The remainder of this chapter will discuss the incorporation of the Spatial Audio Renderer (SAR) task which allows for deeper levels of distribution to be achieved with the XMOS processor firmware. The SAR acts as a container that holds all the calculations for the DBAP algorithm, as well as the other algorithms that are introduced later in this thesis.

Fig. 27 shows the layout of the modifications to the XMOS processor to incorporate the SAR. The green arrows show the path of the audio through the XMOS firmware. The purple arrows show the path of the control. For the second level of distribution, the SAR task was required to calculate the XMOS mixer matrix values and then set the mixer within the Audio Buffer Manager task using an internal interface. For the third level of distribution, the firmware was modified to mix the audio using the SHARC DSP mixer.

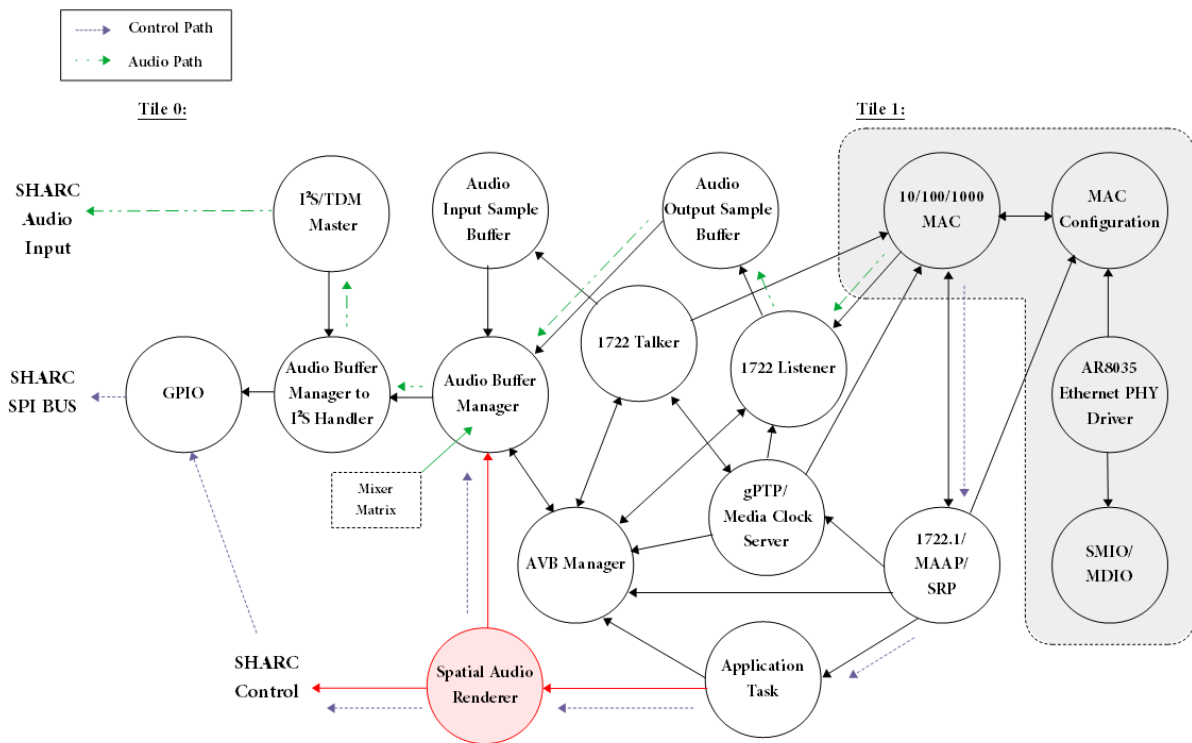


Fig. 27: SPK-4P XMOS AVB firmware layout with modifications in highlighted in red.

4.2.1 The Spatial Audio Renderer (SAR)

The SAR task fulfils three core functions,

1. Interprets the AVDECC AECF SET_MIXER commands received from ImmerGo.
2. Calculates mixer matrix values according to a specific spatial audio rendering algorithm.
3. Controls both the XMOS and SHARC mixer matrices.

The SAR task was implemented on tile 0 alongside the Audio Buffer Manager task as shown in Listing 16.

```

402     on tile[0]: audio_buffer_manager(c_audio, i_imr, i_audio_in_push, i_audio_out_pull, c_media_ctl[0], AUDIO_I2S_IO)
403
404     on tile[0]: spatial_audio_renderer(i_1722_1_control, i_imr);

```

Listing 16: Spatial Audio Renderer declaration.

The following subsections will describe how each of these components were implemented and used for the distribution of the DBAP algorithm.

4.2.1.1 Functions within the Spatial Audio Renderer

The SAR has functions which are controlled by ImmerGo using different descriptor indices shown in Table 11. Here the descriptor indices 0x0001 and 0x0004 allow for the XMOS processor to continue receiving mixer levels and responding to them. The descriptor index 0x0001 is used to set the XMOS mixer matrix and, provided the SHARC mixer firmware is enabled, 0x0004 sets the SHARC mixer matrix. For DBAP to be calculated internally within the SAR, first the speaker configuration positions are set with the descriptor index 0x0010, and then the DBAP algorithm can be used with descriptor index 0x0020 with the track object metadata within the packet.

Descriptor Index	Function
0x0001 : Set XMOS Mixer Gain Values	Sets the XMOS mixer matrix gain parameters
0x0004 : Set SHARC Mixer Gain Values	Sets the SHARC mixer matrix gain parameters
0x0010 : Set Speaker Configuration	Set the Spatial Audio Renderer Speaker Configuration
0x0020 : Perform DBAP	Performs DBAP for track source positions
0x0060 : Track Volume	Controls volume for a particular track
0x0061 : Track Mute	Control mute for a particular track
0x0062 : Track Solo	Control solo for a particular track

Table 11: AECP SET_MIXER descriptor indices for DBAP operations within the SAR.

4.2.1.2 The Spatial Audio Renderer Structure

The SAR task was required to hold variables that are used by the DBAP algorithm to solve for mixer values internally. These are listed below.

- *Speaker Configuration* – Holds the speaker locations for the entire speaker array.
- *The ID of Speakers* – 2 element array holding the IDs of the speakers attached to the endpoint.
- *Track Sources* – Array holding each track source position.
- *Track Mutes* – Array holding the mute state of each track.
- *Track Solos* – Array holding the solo states of each track.
- *Track Volumes* – Array holding the volumes of each track.
- *Mixer Gain Parameters* – Array holding the floating point gain values of each track to each speaker.
- *XMOS Mixer Parameters* – An array holding the gains the signed 8 bit integer format for the XMOS mixer matrix
- *The distance of source to each speaker* – An array holding the distance of a source to each speaker used for the DBAP algorithm.

The SAR task is required to communicate internally with the Application task to receive commands from the ImmerGo server to perform both mixer and audio definition model updates. As well as this, it is required to control XMOS mixing within the Audio Buffer Manager and, for the deepest level of distribution, control SHARC mixing over the SPI bus. The *i_1722_1_entity* interface was used along with newly introduced interfaces. The newly introduced interfaces that connect these tasks are shown in Fig. 28. These interfaces are,

- *i_1722_1_control* – Interface which makes specific calls to the SAR task from the Application task.
- *i_imr* – Interface between the SAR and Audio Buffer Manager to control the XMOS mixer.
- *DSP_AddrParaLoad* – Function which loads parameters onto the SHARC DSP chip.

There are different versions of the firmware.

1. XMOS mixer implemented and SHARC processing disabled.
2. The SHARC mixer enabled with no mixing on the XMOS processor.

The diagram in Fig. 28 shows how the SAR would control both. The XMOS mixer is controlled using the *i_imr* interface introduced above. The SHARC mixer is controlled with the *DSP_AddrParaLoad* function which uses the physical I/O pins connecting the SPI bus between the XMOS and SHARC. This will be discussed in more depth in the following section.

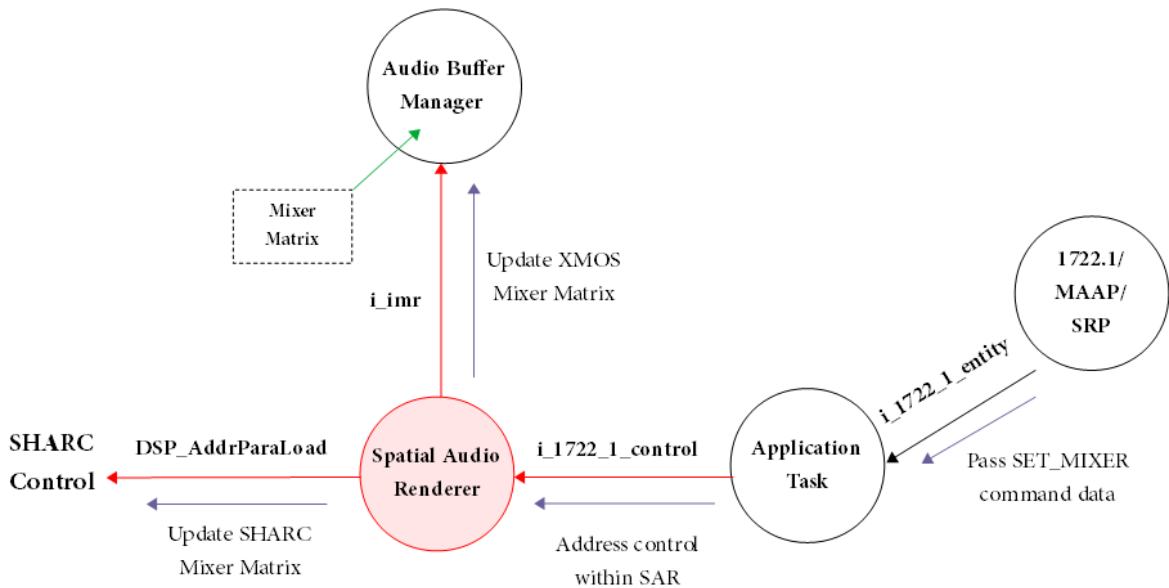


Fig. 28: Interfaces between the SAR and other tasks.

As shown in Fig. 28, when AECP control commands are sent from the ImmerGo server they are passed from the Ethernet MAC to the 1722.1 task according to their headers. The 1722.1 task then makes a callback to the Application task which extracts the data and descriptor indices from the SET_MIXER command. The *i_1722_1_entity* interface is used by the 1722.1 task to pass this data, and has the functions specifically for sending command specific data.

Fig. 29 shows how each descriptor is routed from the Application task to the SAR. When the Application task receives a SET_MIXER callback, it makes calls to the SAR based on the descriptor index within the command. The calls are managed through the *i_1722_1_control* interface client which is an

avb_1722_1_control interface. As shown in **Listing 17** below, it has functions that correspond to those listed in Table 11.

```
1 interface avb_1722_1_control {
2   void set_speakerConfig(unsigned char ids[], unsigned char config[]);
3   void perform_DBAP(unsigned char noSrcs, char src[]);
4   void set_gainXMOSMixerVals( unsigned short num_values, char gain_Values[]);
5   void set_gainSHARCMixerVals( unsigned short num_values, char gain_Values[]);
6   void mute(unsigned char id_src, unsigned char enabled);
7   void solo(unsigned char id_src, unsigned char enabled);
8   void volume(unsigned char id_src, unsigned char volume);
9};
```

Listing 17: The control interface between the application task and SAR.

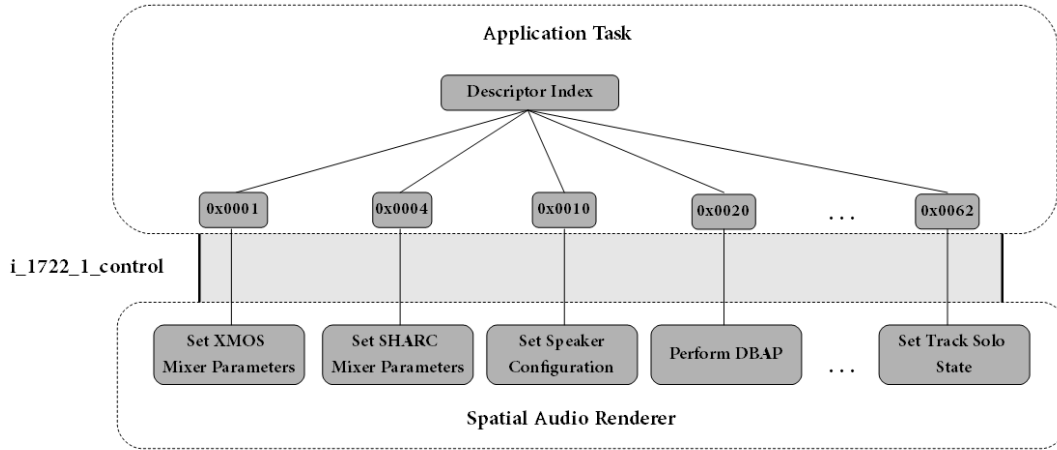


Fig. 29: Routing the descriptor indices to functions within SAR using the *i_1722_1_control* interface.

The SAR sets the values of the XMOS mixer matrix using the *i_imr* interface. This interface is similar to the *i_1722_1_mixer* described in **Listing 18** of the previous chapter and has a function which sets the mixer matrix within the Audio Buffer Manager. The method of setting the SHARC mixer matrix will be discussed later in this chapter.

4.2.1.3 Handling ImmerGo Commands

The SAR is required to receive data from AVDECC AECp SET_MIXER commands as discussed in the previous section. This subsection will discuss how the data is unpacked and used by the SAR, starting with simple updates to the XMOS mixer matrix as discussed in the previous chapter, and then moving further into the ImmerGo Audio Definition Models.

4.2.1.3.1 Controlling the XMOS mixer matrix

The *i_imr* interface is not unlike the *avb_1722_1_mixer* interface discussed in the previous chapters and, as mentioned, is used by the SAR to control the XMOS mixer matrix., however, ImmerGo now controls the XMOS mixer directly by sending an AECp SET_MIXER command with a 0x0001 descriptor index along with the mixer crosspoint parameters. When received by the SAR, the parameters are unpacked into the data structure that holds the XMOS mixer values. Once unpacked and stored, a call with the *i_imr* interface client is made to the Audio Buffer Manager task's *i_imr* server which updates the mixer matrix values.

Listing 18 below shows the unpacking and sending process within the SAR.

```

461 // In the SAR task ...
462 case i_1722_1_control.set_gainXMOSMixerVals(unsigned short num_mix_values, char mix_Values[]): {
463     /* Set XMOS mixer gain parameters */
464     indx = 0;
465     #pragma loop unroll
466     for (int i = 0; i < MAX_SPEAKERS; i++){
467         #pragma loop unroll
468         for(int j = 0; j < MAX_CHANNELS; j++){
469             XMOSMixerVals[i][j] = mix_Values[indx++]; // Update XMOS mixer gain values
470         }
471     }
472     i_imr.send_XMOSgain_vals(ptr_XMOSMixerVals); // Send to Audio Buffer Manager
473     break;
474 }
475

```

Listing 18: Updating the XMOS mixer matrix gain values.

4.2.1.3.2 Unpacking the Speaker Configuration

For the second level of distribution, the SAR is required to receive every speaker's coordinate to solve for DBAP. As shown earlier in Fig. 23 each speaker has an ID associated with it. The IDs of the speakers attached to the endpoint are used to determine which speakers in the entire configuration are attached to that endpoint.

When the SAR receives the speaker configuration, the unpacking process begins. This process is described in greater detail in Appendix A A5 and shown succinctly by the diagram in Fig. 30. When the set speaker configuration command is called, first the IDs of the speaker attached to the endpoint are stored. Then each byte of the speaker coordinate is unpacked into a 32 bit integer and cast into a floating point value for subsequent use in calculations. If a value that is unpacked is 0xFFFFFFFF, the end of the configuration has been reached and the loop terminated. After all the positions are unpacked, the XMOS mixer crosspoints are reset to 0.

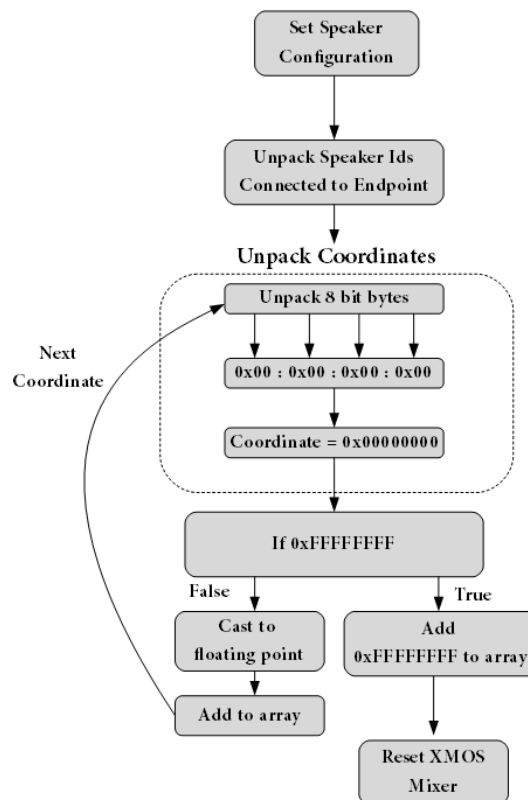


Fig. 30: Unpacking the speaker configuration into the SAR.

4.2.1.3.3 Unpacking Track Sources

In order to perform DBAP, and any other of the spatial audio algorithms, track source IDs and their coordinates need to be routed to the correct function within the SAR. For DBAP, the descriptor index for this routing is 0x0020. When this descriptor index is received, the raw data from that packet is routed to the DBAP algorithm within the SAR. As discussed in section 4.1.3, track source coordinates are sent as 4 bytes prepended with their ID. When received by the SAR, the raw data needs to be cast into a floating point and stored for use by the DBAP calculations. Once the source coordinates are unpacked, the DBAP algorithm is performed for the track and the mixer value is stored. This procedure is then repeated for the next track. Once all the track sources are updated and new mixer values saved, the mixer matrix is set according to the new values. This process is described in Fig. 31 and shown in greater detail in Appendix A A6.

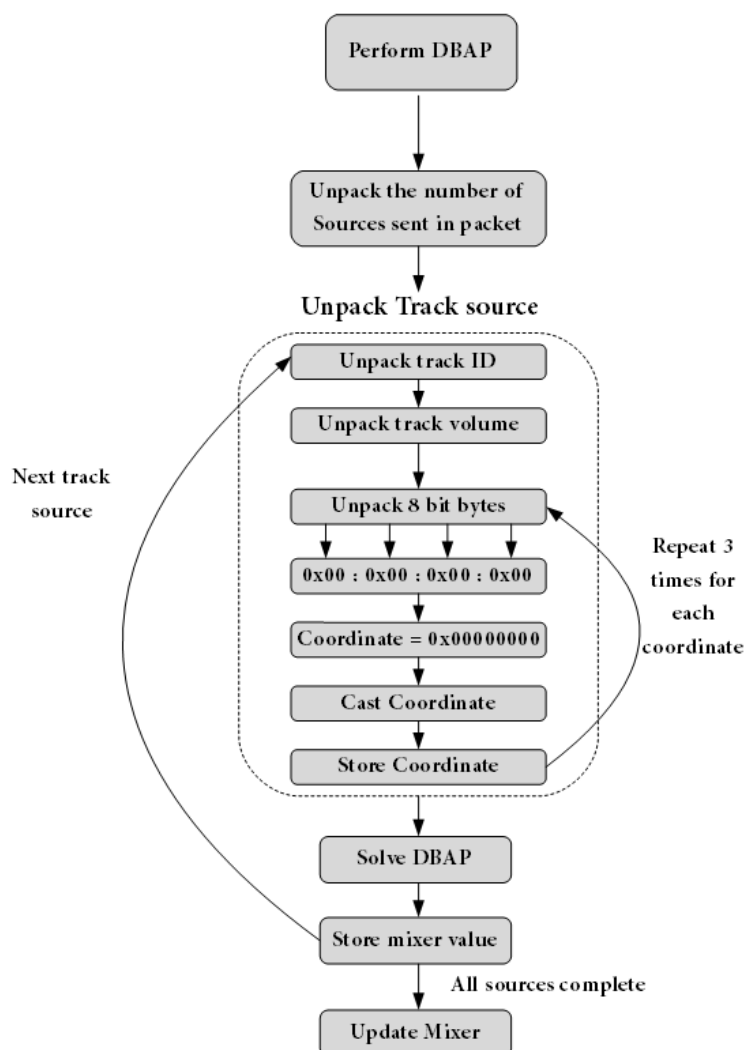


Fig. 31: Unpacking the track source for DBAP calculation within the SAR.

4.2.2 DBAP within the SAR

An overview of how the DBAP algorithm is implemented in the SAR is shown in Fig. 32 below. When an AECN SET_MIXER command is sent with the descriptor index 0x0020, it routes the packet data holding

each track ID and positions at that MTC timestamp to the function which calculates the DBAP algorithm. Then,

1. For each track, their coordinate is unpacked as shown in the previous subsection.
2. The distance of the source to each speaker is calculated and stored in an array for use in the remainder of the function.
3. The normalization factor is solved.
4. The distance of the source to the speaker connected to the processor is selected with the speaker ID.
5. This is used to solve for the inverse distance for the speaker connected to the endpoint.
6. The normalization factor and inverse distance are multiplied together to get the gain value for the correct speaker and stored within the SAR.
7. This is repeated for the next speaker.
8. If there are further track sources, this entire process is repeated with the new track.
9. Once the gains for each track are solved, they are passed to the XMOS mixer matrix over the *i_imr* internal interface.

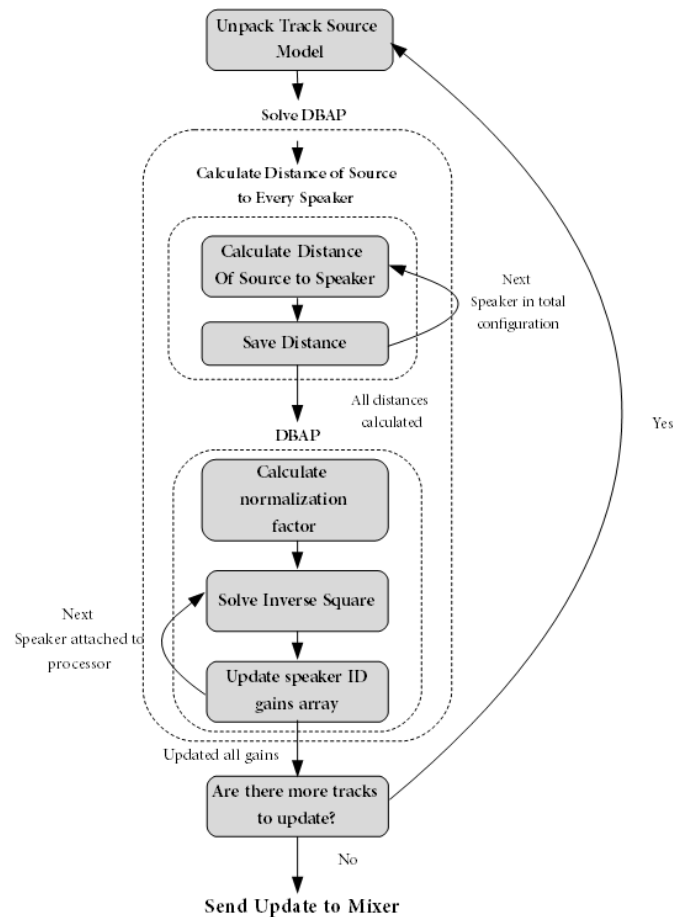


Fig. 32: DBAP algorithm within the XMOS processor.

The output of the DBAP algorithm is required to be scaled from floating point into a signed char format and added to the XMOS Mixer parameters structure for use by the XMOS mixer matrix. These values are then passed to the Audio Buffer Manager over the *i_imr* client interface as described in section 4.2.1.3.1.

The following subsection will describe how the deepest level of distribution is achieved by using the SHARC mixer matrix instead of the XMOS mixer matrix.

4.3 DBAP with the SHARC DSP Mixer Matrix

The SHARC processor has been programmed by engineers at miniDSP. It requires the TDM driver to transfer 32 audio channels from the XMOS chip over the two I²S lines to its mixer matrix and as such, the firmware using the SHARC mixer matrix omits the *i_imr* interface and the XMOS mixer matrix within the Audio Buffer Manager. Shown in Fig. 33, samples that are retrieved by the 1722 listener are passed through the Audio Buffer Manager to the TDM master, which sends all 32 channels to the SHARC mixer matrix. The mixer matrix is then controlled by the SAR using the *DSP_AddrParaLoad* function provided by miniDSP. This function sends parameters to the SHARC over the SPI bus using the physical I/O ports connected the XMOS to the SHARC.

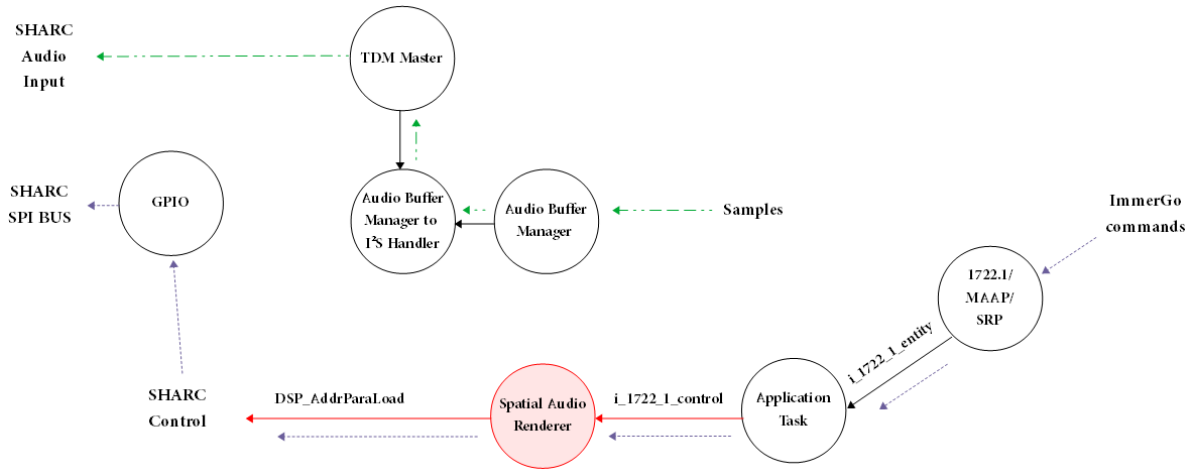


Fig. 33: Audio path and control path from the host computer to SHARC DSP.

Listing 19 below shows that the spatial audio renderer now omits the *i_imr* interface. The *audio_buffer_manager* task is given the TDM *audio_io_type* which ensures the samples are buffered onto the data line correctly.

```

425     on tile[0]: audio_buffer_manager(c_audio, i_audio_in_push, i_audio_out_pull, c_media_ctl[0], AUDIO_TDM_IO);
426
427     on tile[0]: spatial_audio_renderer(i_1722.1_control);

```

Listing 19: XMOS project with the SHARC mixer matrix.

As shown in Fig. 34 below, the SHARC contains a mixer matrix that has a process block for sample attenuation and delay operations at each matrix cross point. After the summation, before sending to the speaker outputs, there is a low pass IIR filter which helps to prevent tearing and popping when changing delay parameters. This is followed by an amplitude smoothing filter which is used to prevent large changes in volume as the mixing values are changed. This mixer matrix is programmable to have up to 32 inputs and 10 output with each delay process having a delay-line of 1024 samples. The SPK-4P is only connected to 2 outputs, so the mixer is set up as a 32x2 matrix and is the matrix size that will be used throughout the remainder of the thesis.

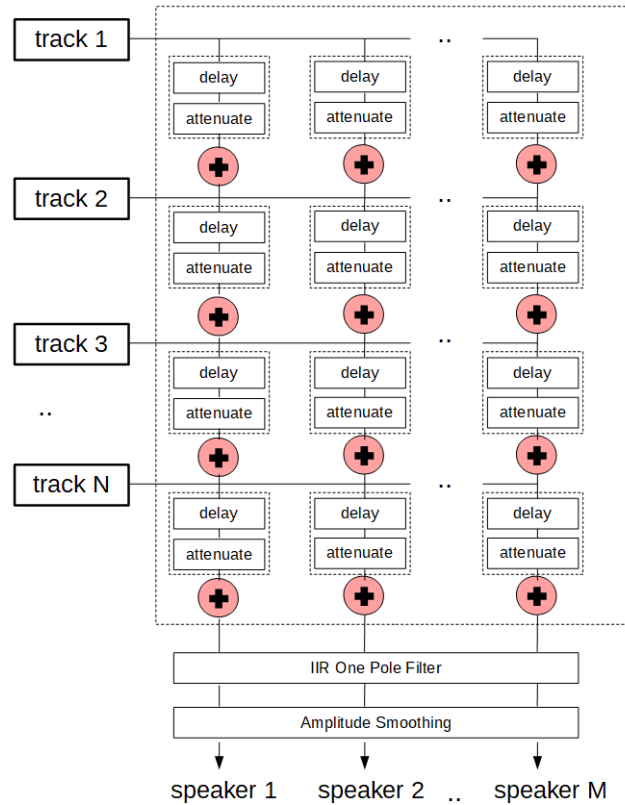


Fig. 34: The MIMO matrix found in the SHARC processor.

4.3.1 SHARC DSP Control

The SHARC mixer is controlled by sending parameters as a stream of bytes over the SPI bus. These parameters change the parameters of the processing blocks within the mixer matrix. The technique used to send the bytes over the SPI bus is beyond the scope of this thesis, however, the methods used by the SAR to update the SHARC DSP will be discussed.

4.3.1.1 The SHARC Mixer matrix Parameters

The SHARC mixer matrix has 3 processes that are addressed with the following parameters,

- Gain and sample polarity control using a floating point Q1.15 representation (Oberstar, 2007).
- Delay line control in samples.
- IIR control and amplitude smooth filter using coefficients between 0 and 1.

The processor blocks are addressed according to the schema shown in Fig. 35. The mixer matrix crosspoint processing operations are addressed using indices 0-64, with 0-31 being the first column cross points and 32-63 being the second column cross points. The delay and attenuation blocks share a single address index, and the filters have their own addresses located at 65 and 66.

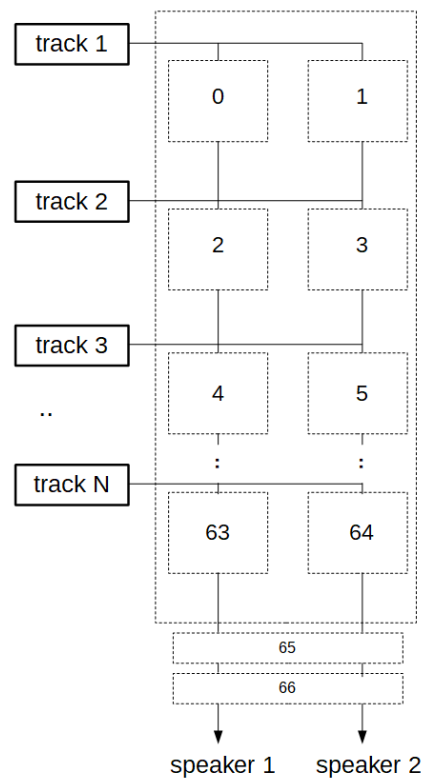


Fig. 35: Address schema for the processing blocks within the SPK-4P SHARC mixer matrix.

4.3.1.2 Setting SHARC DSP Mixer matrix Crosspoints

The gain and sample polarity control are set using parameters in the Q1.15 representation which is used by the SHARC to represent a floating point number as a 16 bit fixed point integer. This allows for 16-bit resolution where every step is $\frac{1}{2^{15}}$ between -1 and 1. As shown in Fig. 36, the Q1.15 representation incorporates a polarity flip indicated by a 1 or 0 in the most significant bit of the 16-bit number.

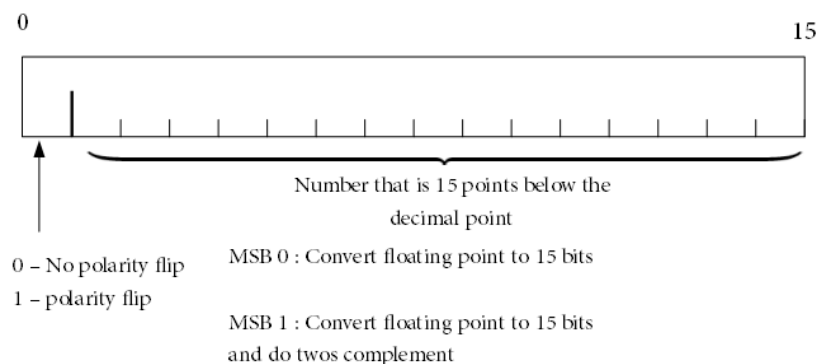


Fig. 36: Q1.15 representation.

In order to transform the floating point mixer values into Q1.15 format, a conversion is required and is shown in Listing 20 below. Here, a gain value is received as a 64 bit floating point value. If it is less than zero, meaning the polarity of the sample should be flipped, the sign bit is set to 1 and the absolute value of the floating point is transformed into a 15 bit integer, which has a 2s complement conversion applied to it. This is returned and sets the remaining 15 bits. If the value is positive, then the most significant bit is set to

0 and the floating point is transformed into a 15 bit number without the 2s complement conversion and added to the remaining bits.

```

1 // in SAR task ...
8 #define HALFSIXTEENBIT 32767

...
157 unsigned short fs_to_Q15(double fs_gain){
158 // 2s complement of a double val
159 // Convert 1 = 0x7FFF, -1 = 0x8001
160     unsigned short output;
161     unsigned short sign;
162     unsigned short fractional;
163
164     if (fs_gain < 0){
165         sign = 1;
166         // Do twos complement
167         fractional = (unsigned short) (ceil(((1*fs_gain)*HALFSIXTEENBIT)));
168         fractional = ~fractional+1;
169         output = (sign << 16) | fractional;
170     }
171     else {
172         fractional = (unsigned short) (ceil((fs_gain*(HALFSIXTEENBIT))));
173         output = fractional;
174     }
175
176     return output;
177 }
178 }

```

Listing 20: Code that converts floating point into Q1.15 representation.

The delay parameter is given simply as a delay in samples. The DBAP algorithm does not require any delay parameters to be changed so it is always given a value of 0.

As shown in **Fig. 37**, a matrix crosspoint's gain and delay parameters are set using a single 32-bit word with the 16 most significant bits reserved for the Q1.15 gain value and the 16 least significant bits reserved for the amount of delay.

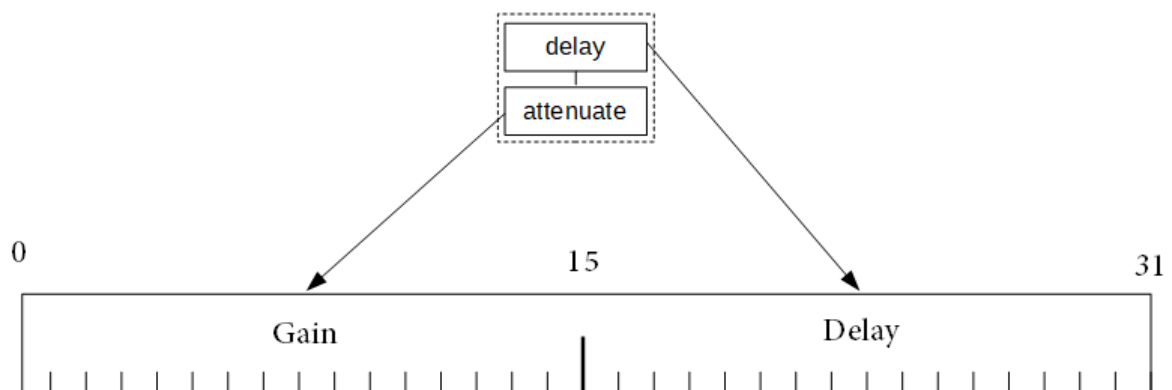


Fig. 37: Mixer matrix crosspoint parameter.

Multiple crosspoints may be set in one command by sending a block of parameters to the SHARC, with a starting address and a message count parameter as shown in **Fig. 38** and **Fig. 39**. **Fig. 38** has loudspeaker crosspoint values set for track 2 and track 3. **Fig. 39** shows track 2's crosspoints being set for the loudspeakers.

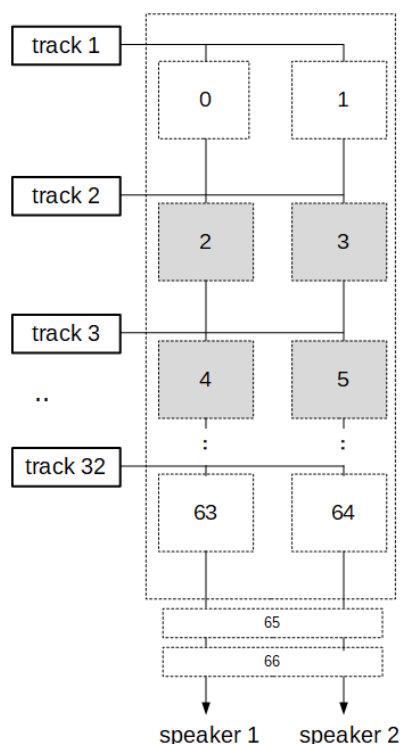


Fig. 38: Setting crosspoints 2-5.

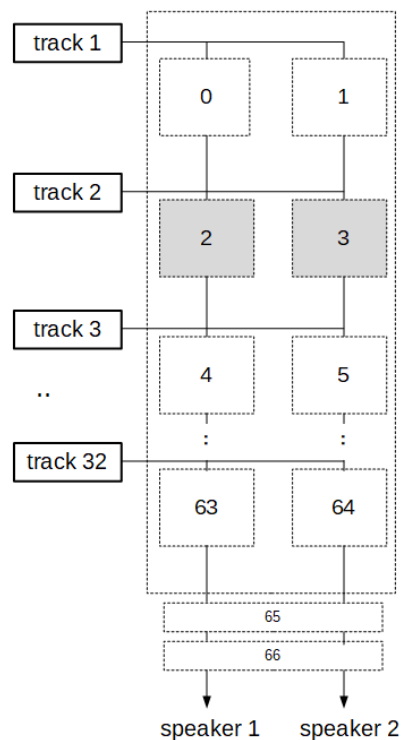


Fig. 39: Setting crosspoints 2-3.

When performing the DBAP calculations, the gain values of loudspeakers connected to a single track are set directly after being solved, followed by any successive track updates. This is shown in Fig. 40 on the following page and described in greater in Appendix A A7. When the track source is moved, the mixer crosspoints for that track are solved according to DBAP. These parameters are then loaded into the parameter buffer with special care to convert the gain values to Q1.15. Every time parameters are added to the buffer, the message counter is incremented. A pointer to the first element in the parameter buffer is sent to the SHARC using the *DSP_AddrParaLoad* function with additional arguments for the address of the first matrix crosspoint and the message counter which is used to set further crosspoints according to subsequent parameters within the buffer.

4.4 Endpoint Clock Synchronization

Spatial audio rendering requires sample accurate delivery of samples to the speakers playing the audio. Ethernet AVB provides device word clock synchronization using the generalized Precision Time Protocol. The XMOS and SHARC processors are synchronized to the same word clock when on the network. The deterministic architecture provided by the XMOS processors guarantees that all the calculations and audio routing happens synchronously within each individual processor according to the word clock.

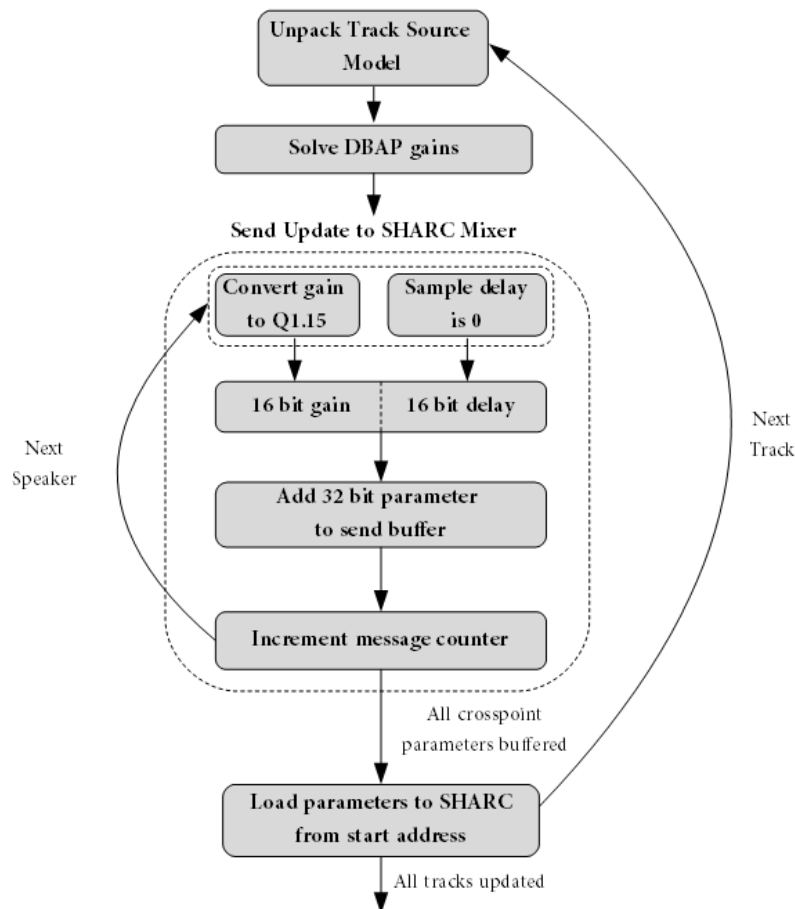


Fig. 40: Setting the SHARC mixer with mixer values from DBAP.

4.5 Summary

This chapter has discussed modifications that were done to the existing ImmerGo application in order to distribute the DBAP calculations to the XMOS processor. The Spatial Audio Renderer (SAR) was introduced as a new processing task within the XMOS processor and was able to respond to AECP commands from ImmerGo. The SAR allowed for ImmerGo to calculate mixer values within the server and send the mixing parameters out to the endpoint mixers as it has done. As well as this, the SAR was also able to perform the DBAP calculations internally. To do this, ImmerGo was required to be able to packetize and send its audio definition models for the speaker configuration and track objects. An overview of how the SAR performed the DBAP calculations and control both the internal XMOS mixer and the SHARC mixer to the output was discussed. This chapter showed how two versions of firmware were developed, one where the SAR controlled the XMOS mixer whilst the SHARC processor was bypassed and another where the XMOS mixer matrix was removed and the SHARC mixer was enabled.

Chapter 5

Vector Base Amplitude Panning

Vector base amplitude panning (VBAP) was the first new rendering algorithm added to the ImmerGo system. This chapter will discuss the theory that underlies the algorithm and how it was implemented within the ImmerGo system at each level of endpoint distribution.

VBAP is an object-based, amplitude panning technique developed by Ville Pulkki and his team in the late 90s (Pulkki V. , 1997). As stated in the paper, it was "developed as an approach to meet the computer music composers' lack of a tool to mix multiple input channels to multiple loudspeakers placed around and above the listener" (Pulkki V. , 1997). It has since gained popularity in commercial cinema and broadcasting and is a prominent part of rendering suite provided in the MPEG-H Object Renderer (Herre, Hilpert, Kuntz, & Plogsties, 2015). It has also been implemented as a renderer for DTS:X's MDA Object-based Immersive Audio Specification (ETSI, 2015) used in applications like Avid's MDACreator, as well as the EBU Audio Definition Model Renderer (EAR) described in the EBU Tech 3388 (European Broadcast Union, 2018).

VBAP represents the panning area as a vector space using a vector base formed from vectors connecting the listener position to each loudspeaker position. It then uses these basis vectors to localize an audio position between multiple speakers. Its 2D version can be proven to coincide with stereophonic panning laws, but the use of a vector base has the advantage of extending to many loudspeakers that can be irregularly placed around a listener. As such, this section will begin with an overview of stereophonic panning followed by 2D VBAP, then 3D VBAP with multiple speakers. This will be followed by the implementation within ImmerGo at the different levels of distribution.

5.1 The Theory Behind VBAP

5.1.1 Stereophonic Panning

Stereophonic panning is considered a psychoacoustic method of panning because it uses a psychoacoustic illusion to localize a source between two speakers. The formulation assumes that two loudspeakers emit coherent signals to a listener situated at a sweet spot between them. These signals may have different amplitudes which is what creates the illusion that there is a single auditory event taking place in the area between the loudspeakers.

As shown in **Fig. 41**, two loudspeakers are positioned symmetrically about an angle φ_0 with respect to the x plane. The signals coming from these loudspeakers are controlled using two gain factors g_1 and g_2 respectively. A virtual source is able to be positioned between these loudspeakers according to a panning law that determines the level of the signal from each loudspeaker.

The direction of a mono virtual source is determined according to the relationship between the amplitudes coming from both loudspeakers. If the virtual source is moving, it's loudness should remain constant and

so the gain factors should be normalized (Morrell & Reiss, 2009). The normalized sound power produced by the two gain factors g_1 and g_2 satisfies the normalization equation (5.1), such that:

$$g_1^2 + g_2^2 = C \quad (5.1)$$

Here $C > 0$ and corresponds to the overall sound volume of the source (Pulkki V. , 1997).

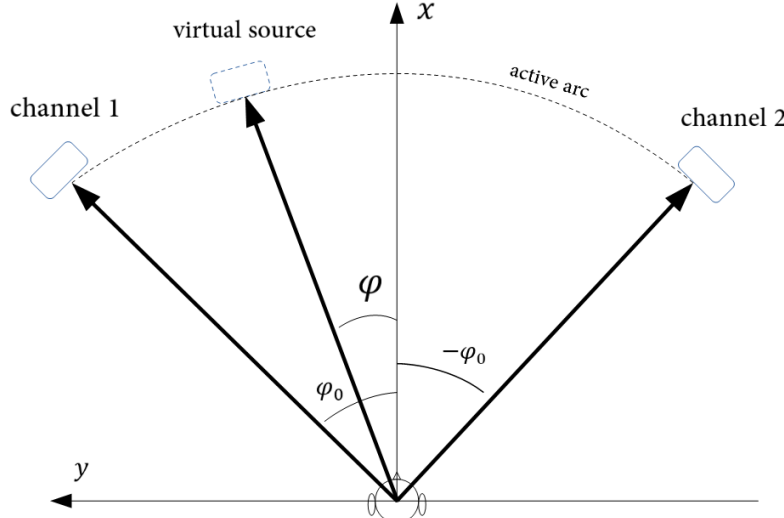


Fig. 41: Typical stereophonic setup.

5.1.2 Two-Dimensional VBAP

It is worth going through the formulation for VBAP in two dimensions before extending to three dimensions to help explain the concept clearly. Fig. 42 shows how a stereophonic loudspeaker configuration can be formulated as vectors $\mathbf{l}_1 = [l_{1x} \ l_{1y}]^T$ and $\mathbf{l}_2 = [l_{2x} \ l_{2y}]^T$ in the 2D plane, with each vector stemming equidistantly from a listener. The loudspeaker vectors allow a phantom source vector $\mathbf{p} = [p_1 \ p_2]^T$ to be constructed within their vector space. For immersive effect, a listener would be surrounded by multiple loudspeakers, each with their own vector basis that a phantom source may be projected with.

$\mathbf{p} = [p_1 \ p_2]^T$ is defined as the vector which points to a phantom source that is situated within the vector space. Therefore $\mathbf{p}$ can be represented as a linear combination of vectors $\mathbf{l}_1$ and $\mathbf{l}_2$. This leads to the construction of equation (5.2)

$$\mathbf{p} = g_1 \mathbf{l}_1 + g_2 \mathbf{l}_2 \quad (5.2)$$

where g_1 and g_2 are gain factors which should be treated as non-negative scalar values.

Equation (5.2) can also be shown as a set of linear equations such as

$$\begin{aligned} p_1 &= g_1 l_{1x} + g_2 l_{2x} \\ p_2 &= g_1 l_{1y} + g_2 l_{2y} \end{aligned} \quad (5.3)$$

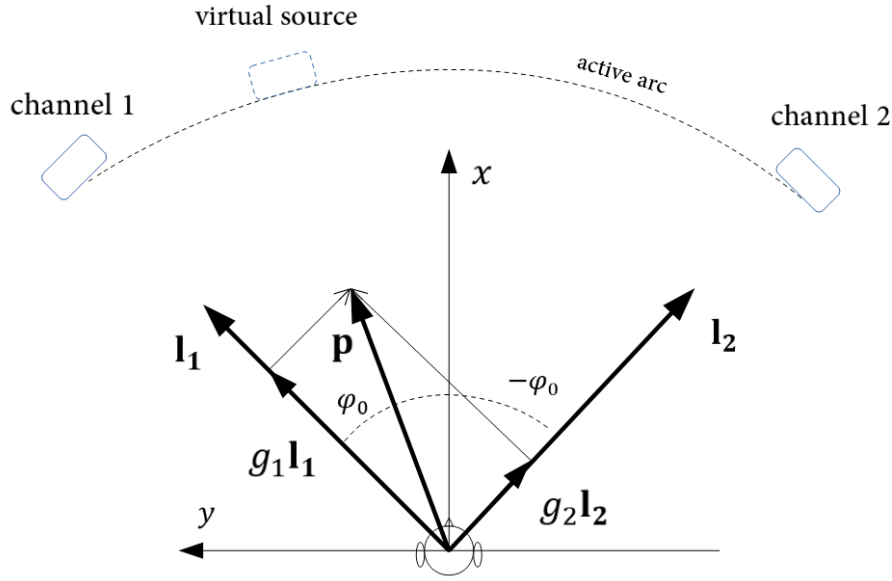


Fig. 42: VBAF formulation for two dimensions.

The matrix form of these equations may be represented as

$$\mathbf{p}^T = \mathbf{g} \mathbf{L}_{12} \quad (5.4)$$

where $\mathbf{g} = [g_1 \ g_2]$ and $\mathbf{L}_{12} = [\mathbf{l}_1 \ \mathbf{l}_2] = \begin{bmatrix} l_{1x} & l_{1y} \\ l_{2x} & l_{2y} \end{bmatrix}$.

This equation can be solved for, provided the inverse matrix $\mathbf{L}_{12}^{-1}$ exists,

$$\mathbf{g} = \mathbf{p}^T \mathbf{L}_{12}^{-1} = [p_1 \ p_2] \begin{bmatrix} l_{1x} & l_{1y} \\ l_{2x} & l_{2y} \end{bmatrix}^{-1} \quad (5.5)$$

where $\mathbf{g} = [g_1 \ g_2]$ and $\mathbf{L}_{12} = [\mathbf{l}_1 \ \mathbf{l}_2] = \begin{bmatrix} l_{1x} & l_{1y} \\ l_{2x} & l_{2y} \end{bmatrix}$.

The inverse matrix exists when $\mathbf{L}_{12} \mathbf{L}_{12}^{-1} = \mathbf{I}$ where $\mathbf{I}$ is the identity matrix which is the case when $\varphi_0 \neq 0^\circ$ and $\varphi_0 \neq 90^\circ$. These angles would only occur when the loudspeakers are in the same position or opposite each other which is typically uninteresting for accurate stereophonic reproduction.

If the loudspeakers are non-orthogonal (i.e $\varphi_0 \neq 45^\circ$) the gain factors need to be normalized (Pulkki V. , 1997). This can be achieved using the Gram-Schmidt normalization (Strang, 2016) process shown below in equation (5.6).

$$\mathbf{g}_{\text{scaled}} = \frac{\sqrt{C} \mathbf{g}}{\sqrt{g_1^2 + g_2^2}} \quad (5.6)$$

This normalization method can be used to render object-based spatial audio to standard surround sound loudspeaker setups like 5.1, 7.1 etc which are not orthogonal. Even though they are not orthogonal, each loudspeaker pair is able to form an active arc for a phantom source to be projected. The VBAP algorithm can then cycle through each active arc and determine the active arc that is suitable for the projection of the phantom source. The gain factors for those speakers will then be used while the other pairs of speakers will not emit any sound. This is elaborated on in the implementation section of this chapter.

5.1.3 Three-dimensional VBAP

By adding in a third loudspeaker that is equidistant but not in the horizontal plane we introduce a third loudspeaker vector as shown in **Fig. 43**. This vector coupled with the other two forms a three-dimensional vector basis that may be used for phantom source projections in three dimensions. The active arc becomes an *active triangle* and the task now is to solve for the gain values of that source for each of the three loudspeakers that form this triangle. This results in a more complicated matrix computation, and selection process when more than three loudspeakers are introduced.

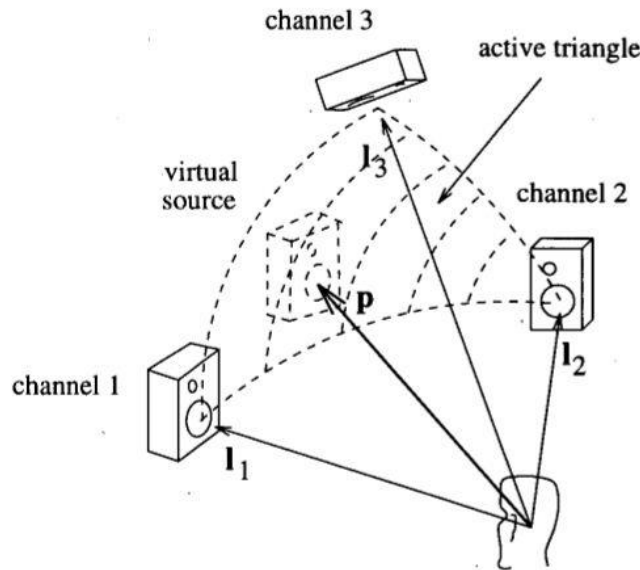


Fig. 43: VBAP formulation for three dimensions. *Source:* (Pulkki V. , 1997)

As with the two-dimensional setup, the gain factors for 3D VBAP are required to be normalized to an equal intensity. This is shown in equation (5.7) below.

$$g_1^2 + g_2^2 + g_3^2 = C \quad (5.7)$$

where $C > 0$ still relates to the overall volume.

In almost all cases, the volume of the source will be controlled prior to panning and as such we may simply set $C = 1$.

We can then extend the 2D gain equation (5.5) to the 3D equation shown in (5.8)

$$\mathbf{g} = \mathbf{pL}_{123}^{-1} = [p_1 \ p_2 \ p_3] \begin{bmatrix} l_{1x} & l_{1y} & l_{1z} \\ l_{2x} & l_{2y} & l_{2z} \\ l_{3x} & l_{3y} & l_{3z} \end{bmatrix}^{-1} \quad (5.8)$$

where $\mathbf{g} = [g_1 \ g_2 \ g_3]$.

Once the gain values are calculated we can multiply the signal S for each sample t by the gain corresponding to each loudspeaker for the active triangle.

$$\begin{bmatrix} L_1(t) \\ L_2(t) \\ L_3(t) \end{bmatrix} = s(t) * \begin{bmatrix} g_1 \\ g_2 \\ g_3 \end{bmatrix} \quad (5.9)$$

5.1.4 VBAP and the Convex Hull

Extending this technique to more than three loudspeakers requires the convex hull of the array of loudspeakers to be determined so multiple vector basis may be determined. The convex hull is the smallest perimeter that contains all of the loudspeakers and is determined as triplet vertices in three dimensions.

Fig. 44 below shows the convex hull of a 15.0 speaker array.

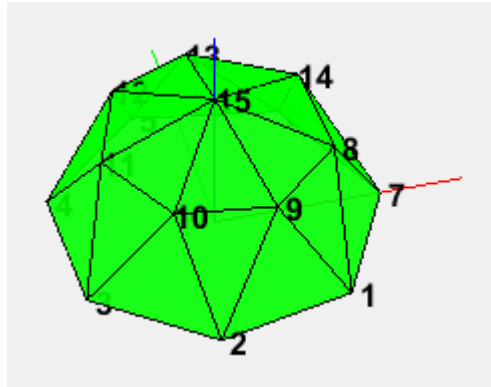


Fig. 44: Convex-hull of a 15.0 speaker array.

There are various algorithms that can solve the convex hull, but the most efficient is the Quickhull algorithm (Barber, Dobkin, & Huhdanpaa, 1996). Quickhull has also been used successfully in other VBAP systems (Choi, 2012) (Pulkki & Politis, 2015) and is the preferred choice in the implementation done in this thesis. The complete solution for Quickhull is omitted in this document as it can be found in (Barber, Dobkin, & Huhdanpaa, 1996), but it is important to note that the algorithm has an $O(n \log n)$ complexity which is faster than other convex hull algorithms, and it can be used to calculate the convex hull in three dimensions.

Also important to note is that once the speaker configuration's convex hull is determined, each loudspeaker triangle won't change. This allows for each triangle's inverse loudspeaker matrix to be computed prior to performing the localization calculations, which saves a lot of processing time.

5.1.5 VBAP Algorithm

The VBAP algorithm requires precomputation before localizing. These precomputations are implemented as steps. Assuming there is a set of N loudspeakers, each with their own locations

$(x_1, y_1, z_1), (x_2, y_2, z_2) \dots (x_n, y_n, z_n)$ equidistant from the listener, the precomputation is performed as follows:

1. Compute the convex hull for N loudspeaker vertices using the Quickhull.
2. Invert each of the loudspeaker matrices that are part of the convex hull.

The inverted loudspeaker matrices may be labelled as $\mathbf{L}_{qrs}^{-1}$ where q, r and s uniquely represent any of the $1 \leq N$ loudspeaker IDs. Each loudspeaker inverse matrix is then added to an array that is associated with loudspeakers IDs of a particular active triangle and may be used as a vector basis for projection.

Now an arbitrary projection vector may be defined such that it lies in any active triangle on the convex hull. If the projection vector lies outside of an active triangle, that active triangle will not be used. The algorithm for real-time panning is defined as a for-loop that runs through each loudspeaker matrix.

3. Perform the matrix calculation for the inverse loudspeaker matrix and get $\mathbf{g} = [g_1 \ g_2 \ g_3]$
4. If the minimum value of the vector $\mathbf{g}$ is greater than 0, then the projection vector lies in the vector space defined by the loudspeaker triangle, if not we discard that gain vector and do the matrix calculation for the next loudspeaker matrix.

5.1.6 Multiple Direction Amplitude Panning (MDAP)

VBAP works well for high precision three-dimensional panning. If a source is positioned directly on a speaker, it will only play from that speaker; if a source is positioned in the middle of the active triangle, it will play at equal volume from all three speakers. However, this high precision panning may not be favourable in situations where the virtual source is moving from one active triangle to the next because of the abrupt changes in amplitude when moving between the loudspeaker triangles. This is particularly noticeable where there are multiple small spanning active triangles. It also has the problem of producing the *detent effect* where the source is perceived as stationary coming from the closest loudspeaker. A solution to this would be to spread the virtual source such that it spans multiple active loudspeakers simultaneously. The spread of the source will remain more even as it is moved across the loudspeaker, and since it is coming from multiple loudspeakers the image will be perceived as wider. Pulkki proposed Multiple Direction Amplitude Panning (MDAP) as a method of spreading the source (Pulkki V. , 1999).

As can be seen in Fig. 45, MDAP is an extension of VBAP whereby a minimum of 8 secondary direction vectors are constructed from the origin to a ring, or multiple rings, surrounding the primary direction vector.

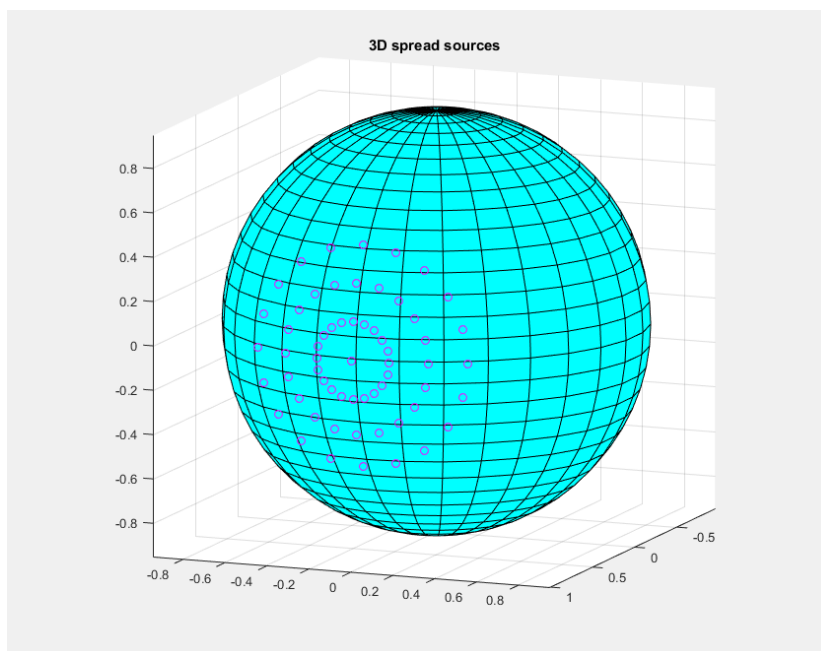


Fig. 45: MDAP spread sources. Spread angle is 60° with 3 rings containing 20 sources.

A method of constructing these rings has been described in (Pulkki V. , 1999) and implemented as a MATLAB simulation in (Politis & Pulkki, 2015). Each ring is made by constructing multiple vectors that are spaced equally within a circle and are perpendicular to the primary direction. These vectors are then squeezed into a ring surrounding the primary virtual source, using a chosen angle which is considered spread width. This can be repeated so that multiple rings are created at different spread widths. This may only prove useful in speaker arrays with many speakers, since the panning will have a finer resolution with the increased loudspeaker triangles.

5.2 VBAP Implementation

The VBAP calculations have been implemented in both the ImmerGo NWJS framework and the endpoint XMOS processors. As mentioned later in this subsection, processing timing constraints prevented MDAP from being implemented within the endpoint processors.

5.2.1 VBAP in NWJS

The VBAP algorithm is implemented as a separate Javascript module named “VBAP.js” that is used by the server code. This module’s functionality was split into two main parts,

1. Determine the convex hull and its associated inverted matrices required for the matrix calculation.
2. Perform VBAP on a virtual source and send mixer values to the XMOS endpoint processors.

The VBAP module used two NPM libraries to speed up the development process. These are,

- Mathjs ²⁰ – for matrix and vector operations.
- Quickhull3d ²¹ – for calculation of the convex hull of the loudspeaker array.

The VBAP module works as an API by exposing the core functions used by the ImmerGo server. The functions that are exposed from the VBAP library, and used by the ImmerGo server, are given in the diagram **Fig. 46** below. VBAP.js exports the functions shown in **Listing 21**. The VBAP.js module exports the *SetMesh* function which is called with the loudspeaker positions, *ls_pos*, and allows the server to determine the convex hull, *lsMesh*, and inverse loudspeaker matrices, *ls_inv_mtx*, required for the panning calculation. Once set, these variables may be retrieved from the module using the *get_invLsMtx* and *get_triFaces* functions.

The functions highlighted within the client in **Fig. 46** we modified for use by VBAP. The *setSrcCoords* function is used to set the track model within the *soundControlWindow*. This function was modified to force the virtual source to remain restricted to the loudspeaker convex hull. When the server receives coordinates, it sets them to a track using the *setCurrentCoords*. The spread angle of the source may also be set with the *setSpread* function which is used by the client to activate MDAP. VBAP.js exports the *vbap3* function which is called with the virtual source position and returns mixer values that are sent to the endpoints using the *updateSpeakerMixLevels* function.

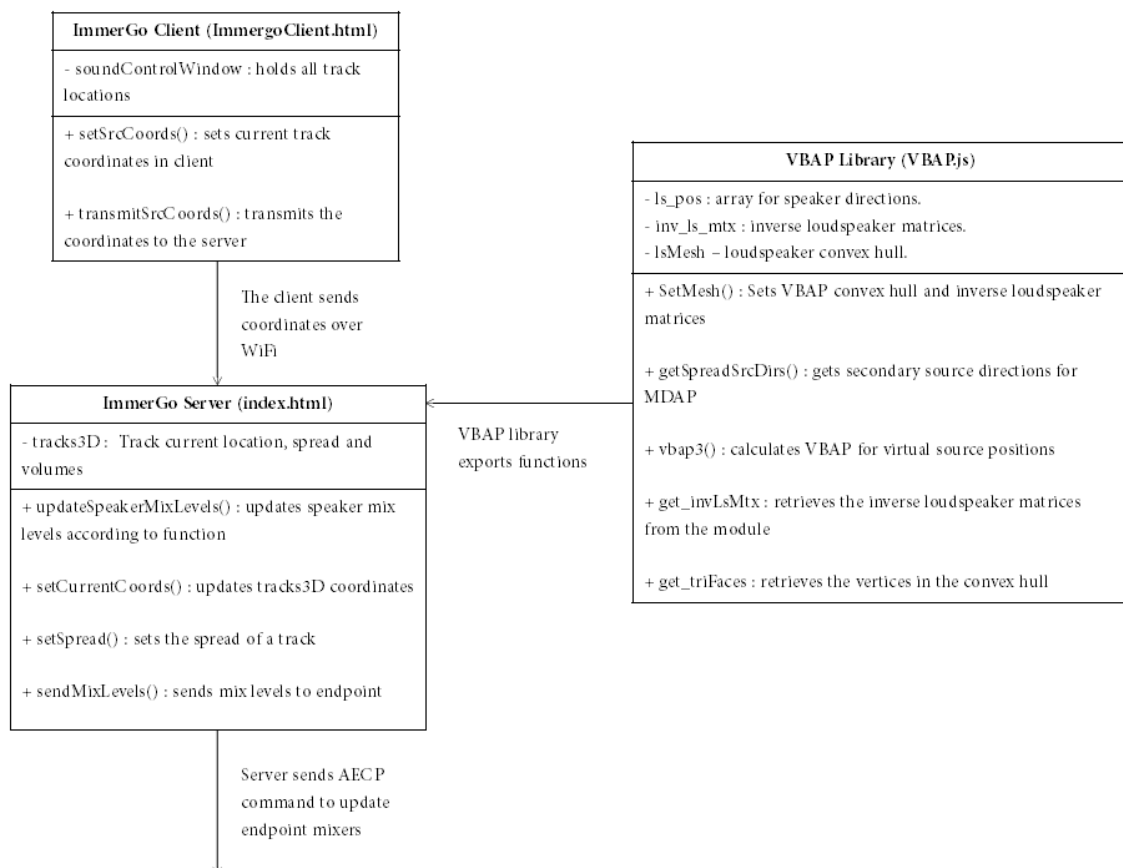


Fig. 46: Diagram showing functions exposed from to the ImmerGo server from the VBAP library.

²⁰ <https://www.npmjs.com/package/mathjs>

²¹ <https://www.npmjs.com/package/quickhull3d>


```

525 module.exports={
526   SetMesh:SetMesh,
527   getSpreadSrcDirs:getSpreadSrcDirs,
528   vbap3:vbap3,
529   get_invLsMtx : get_invLsMtx,
530   get_triFaces : get_triFaces };

```

Listing 21: Module exports within the VBAP.js library.

The VBAP library holds global variables pertaining to the VBAP configuration as shown in Listing 22.

```

12 var ls_mesh = new lsMesh();           // Convex hull mesh
13 var inv_ls_mtx = []                   // Inverse loudspeaker matrices
14
15 /***** Classes *****/
16 function lsMesh(faces, vertices){
17   /* Loudspeaker mesh */
18   this.faces = faces;                 // Each triangle faces vertices
19   this.vertices = vertices;           // Every vertex in the configuration
20 }

```

Listing 22: VBAP library global variables.

VBAP restricts virtual sources to the convex hull of the loudspeakers which are placed in a sphere, or hemisphere around the listener. The ImmerGo client UI was modified to give a visual representation of panning a virtual source around a hemisphere as shown in Fig. 47 below.

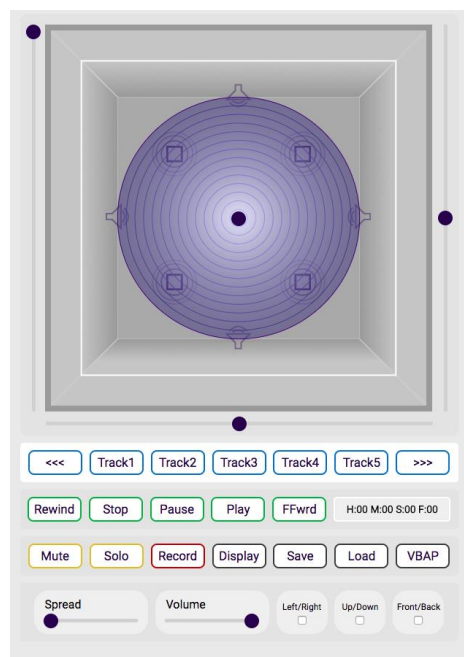


Fig. 47: ImmerGo VBAP client

When a client pans a source using VBAP, the Z coordinate is solved according to the x and y coordinates and the radius of the sphere using equation (5.10). This ensures that the virtual source remains on the sphere.

$$z = \sqrt{r^2 - x^2 - y^2} \quad (5.10)$$

5.2.1.1 Calculating VBAP configuration

The VBAP configuration holds the parameters that may be calculated prior to moving the source. These are specifically the inverse loudspeaker matrices with their associated faces in the convex hull. As shown in **Fig. 48**, after the speaker configuration XML is loaded into the server, the distance of each speaker to listener origin is checked to be within a 5cm limit, in other words, it lies on the surface of the sphere. If all are within bounds, then the VBAP configuration can be solved for, if not then VBAP is disabled and the server falls back to the DBAP algorithm. This procedure is described in greater detail in Appendix B B1.

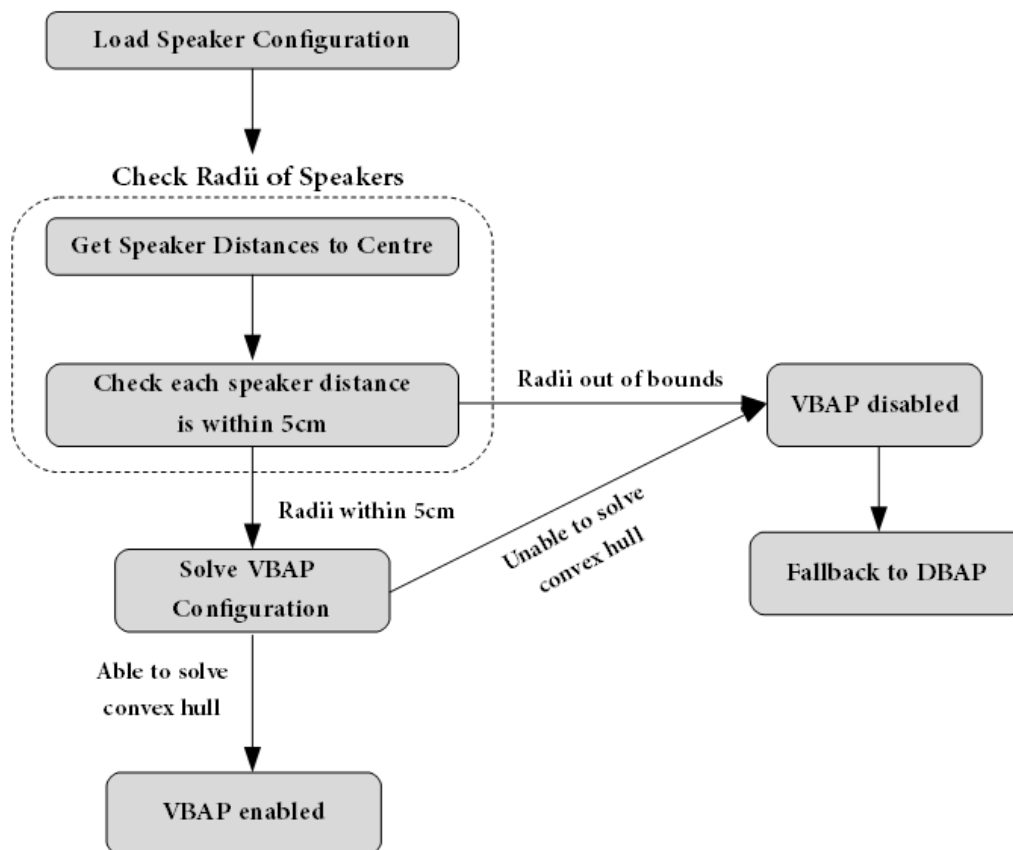


Fig. 48: Checking Speaker radii is in bounds.

The VBAP configuration is solved using the *SetMesh* function within the VBAP library. As shown in **Fig. 49**, this function solves for the convex hull of the speaker configuration and determines the loudspeaker inverse matrices associated with the faces in the hull. The convex hull function itself performs Quickhull on the vertices. If it is unable to define a hull due to there being too few vertices or vertices that lie on the same line, then VBAP is disabled. Once solved, convex hull faces that are orientated incorrectly are discarded.

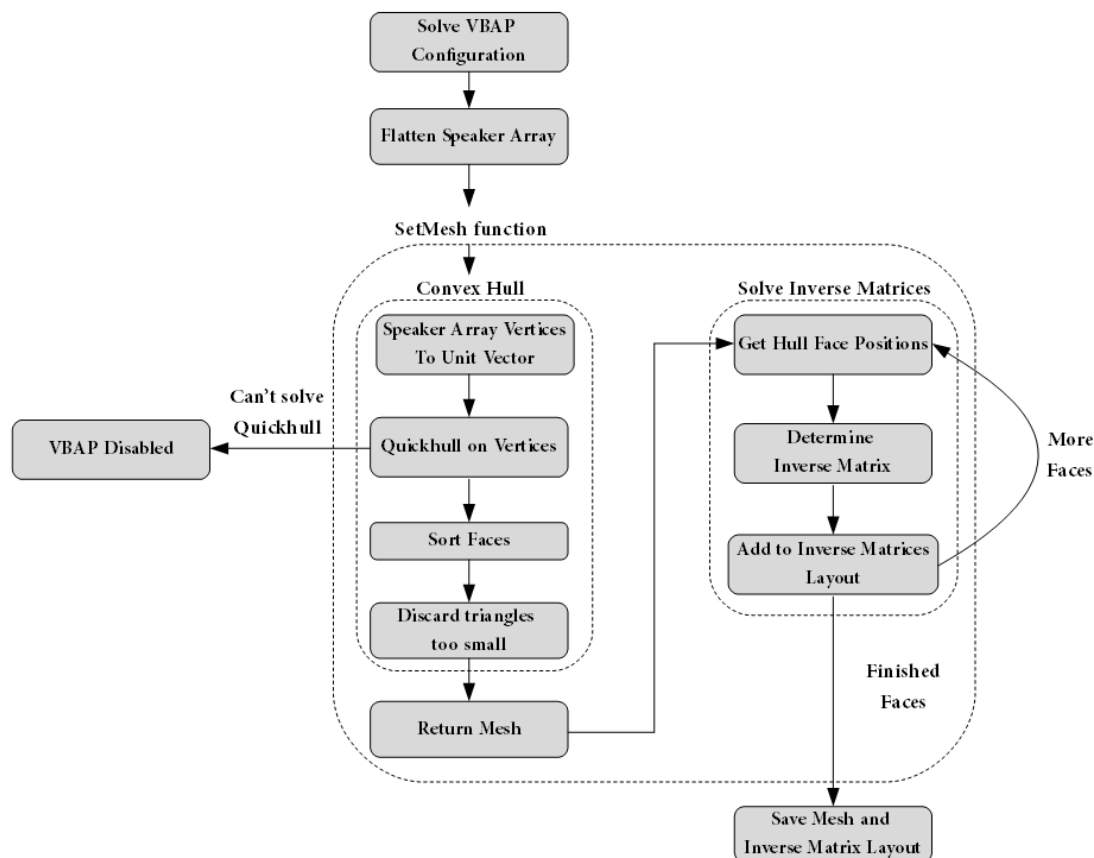


Fig. 49: Solving the VBAP configuration.

Faces are discarded using the *keepValidTriangles* function as shown in Listing 23. The function iterates over each triangle face and determines the angle between the surface normal of the triangle to the centroid of the triangle. If the angle is greater than $\frac{\pi}{2}$ then the face is pointing away from the listener and thus discarded. All triangles whose angles are less than $\frac{\pi}{2}$ are saved and returned by the function. This process is illustrated in Fig. 50 where the green face has the angle of the centroid to normal $\theta_1 < \frac{\pi}{2}$ and the red face has the angle $\theta_2 > \frac{\pi}{2}$ and so is discarded.

After the convex hull is solved, the inverse loudspeaker matrices are determined using the *mathjs* library. The convex hull points and inverse matrices are saved for the VBAP calculation.

```

165 function keepValidTriangles(faces,vert){
166   /* Function that discards invalid triangles */
167   let validfaces = [];
168   let n = faces.length;
169
170   for ( let i = 0 ; i < n; i++){
171     let currentface = faces[i];
172
173     let vec = [vert[currentface[0]], vert[currentface[1]], vert[currentface[2]]]; // Current face vectors
174     let triangle_normal = math.cross(math.subtract(vec[0], vec[1]), math.subtract(vec[1], vec[2])); // Triangle surface normal
175
176     let centroid = math.mean(vec, 0); // Centroid of triangle vectors
177
178     if (math.acos(math.dot(centroid,triangle_normal)) < Math.PI/2 ){
179       validfaces.push(faces[i]); // If orientated correctly add to valid triangles
180     }
181   }
182   return validfaces;
183 }

```

Listing 23: Determining which convex hull triplets to discard.

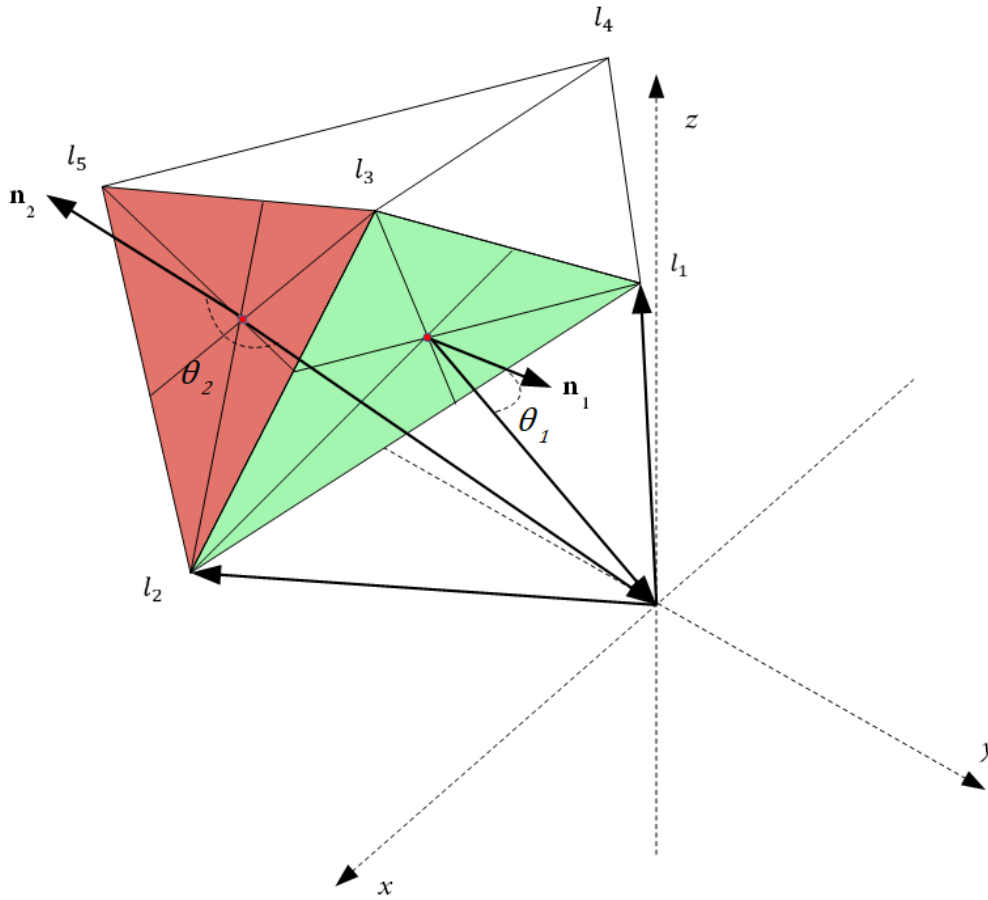


Fig. 50: Discarding triangles.

5.2.1.2 Performing VBAP within NWJS

Once the VBAP configuration has been set within the VBAP module, the VBAP calculation may be performed on a virtual source using the *vbap3* function. As shown below in Fig. 51, when a track source update is received, its coordinates and spread parameter is given to the *vbap3* function. If the spread value is greater than 0, then MDAP is performed, otherwise VBAP is performed. The VBAP algorithm performs the matrix calculation for each triangle in the convex hull. If each gain value output from the calculation is greater than 0, then the vector lies within that vector space and those gain values are outputted. These values are then normalized according to Gram-Schmidt normalization and the correct speaker IDs of the face have their mix levels updated for the track. Mixer updates are then sent to the endpoints with the updated speaker models when an MTC frame is received.

The VBAP algorithm calculation within the *vbap3* function is shown in Listing 24. There is a gain array that is used to hold the gain values for each speaker in the configuration. As shown in Fig. 51, the matrix calculation of the virtual source vector is performed for each inverse matrix until the correct projection face is found. Once found, the array of gains is filled with the output of the calculation. After all the gains have been filled, the array is normalized and returned.

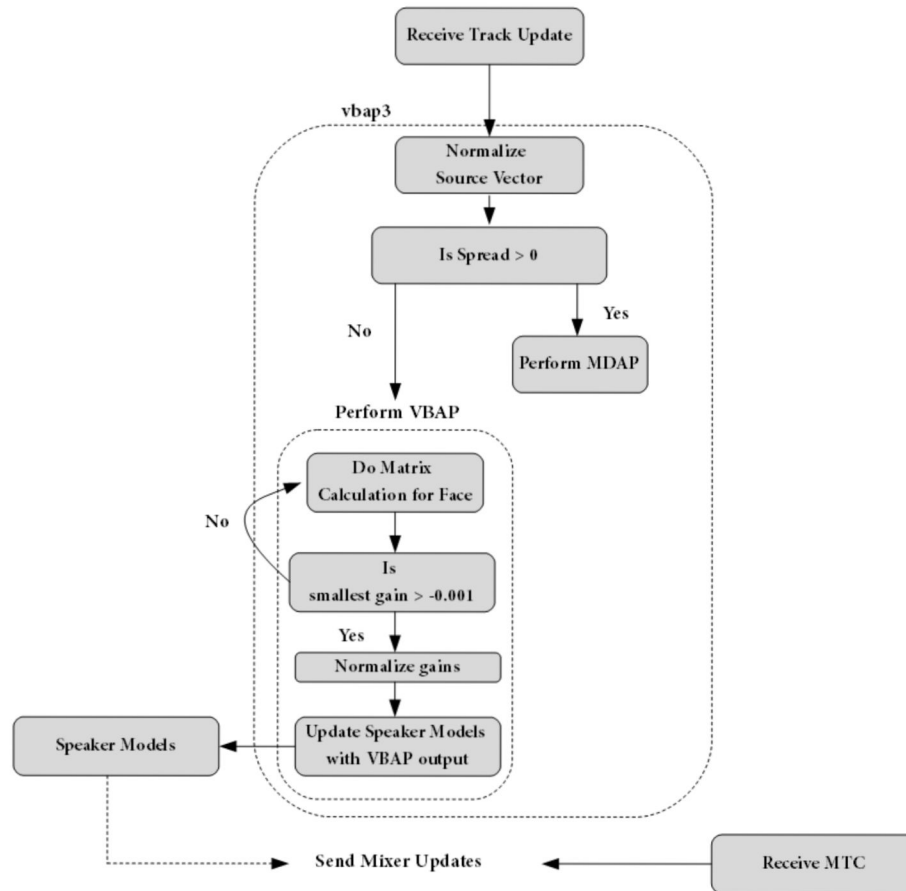


Fig. 51: Panning with VBAP in ImmerGo.

```

359  /* Compute VBAP */
360  for (let i = 0; i < ls_mesh.faces[0].length; i++){
361      let g_temp = [];
362      // Perform VBAP matrix calculation
363      g_temp[0] = ls_invMtx[i][0]*src_vector[0] + ls_invMtx[i][1]*src_vector[1] + ls_invMtx[i][2]*src_vector[2];
364      g_temp[1] = ls_invMtx[i][3]*src_vector[0] + ls_invMtx[i][4]*src_vector[1] + ls_invMtx[i][5]*src_vector[2];
365      g_temp[2] = ls_invMtx[i][6]*src_vector[0] + ls_invMtx[i][7]*src_vector[1] + ls_invMtx[i][8]*src_vector[2];
366
367      if (math.min(g_temp) > -0.001) { // Virtual source vector lies in this triangle matrix
368          for (let j = 0; j < ls_mesh.faces[i].length; j++){ // Iterate through the triangle face and set the gains for the matrix
369              gains[ls_mesh.faces[i][j]] = (g_temp[j] < 0) ? 0 : g_temp[j]; // If the gain is less than 0, limit it to 0
370          }
371          break;
372      }
373  }
374
375  /* Normalize gains */
376  gains = math.divide(gains, math.sqrt(math.sum(math.dotPow(gains, 2))));
377
378  /* Convert NaN values to 0 */
379
380  gains = gains.map(getNum);
381
382  return gains;
383
384  }

```

Listing 24: The VBAP calculation

5.2.1.3 Performing MDAP in NWJS

MDAP is activated when the spread parameter for a source is greater than 0. The spread value describes the squeezing angle of the MDAP secondary source rings. Fig. 52 shows how when the spread value is greater than 0, MDAP is enabled.

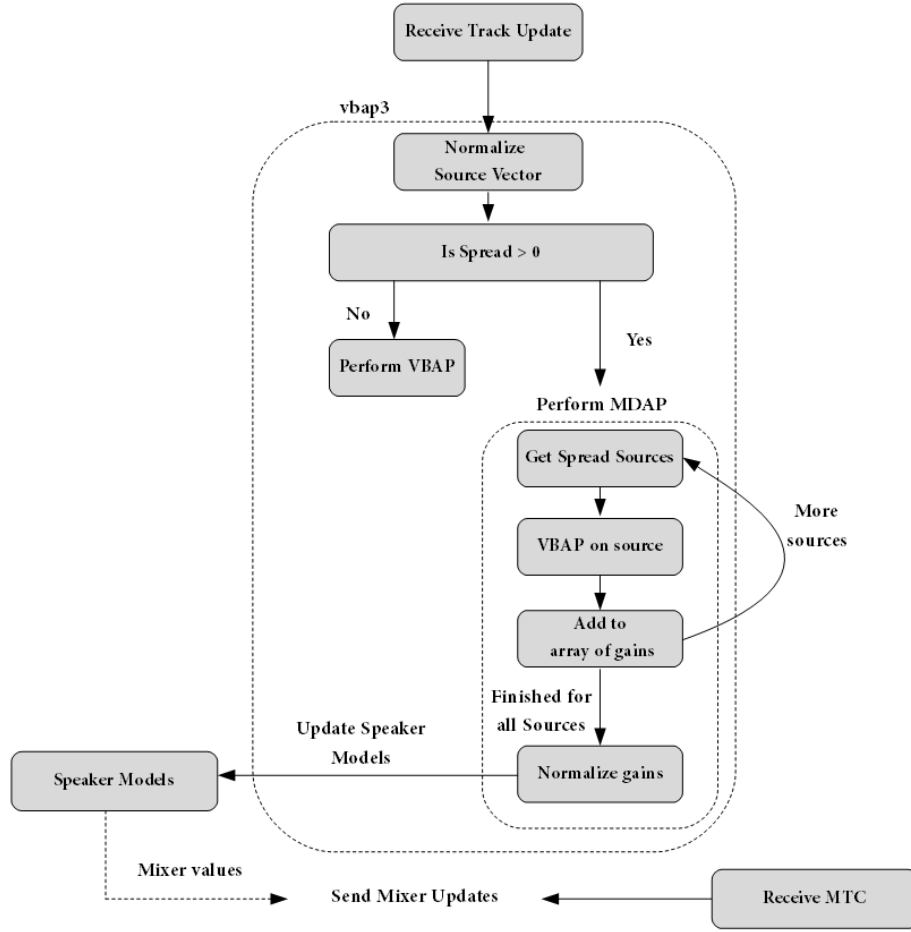


Fig. 52: Panning with MDAP in ImmerGo.

MDAP in ImmerGo creates 8 sources in 1 ring around the primary source. This process is shown in greater detail in Appendix B B2. As shown in Fig. 53, first a single vector is created perpendicular to primary source $\mathbf{u} = (x, y, z)$, and then rotated incrementally θ degrees around the primary direction vector, saving each new vector so as to create a set of secondary sources $S = \{\mathbf{s}_1, \mathbf{s}_2, \dots, \mathbf{s}_8\}$. The rotation is done using the cartesian rotation matrix shown in equation (5.11).

$$R(\theta) = \cos(\theta) I + \sin(\theta)[\mathbf{u}]_{\times} + (1 - \cos(\theta))(\mathbf{u} \otimes \mathbf{u}) \quad (5.11)$$

Where I is the identity matrix, $[\mathbf{u}]_{\times}$ is the cross product matrix of $\mathbf{u}$, and $\mathbf{u} \otimes \mathbf{u}$ is the tensor product. i.e

$$[\mathbf{u}]_{\times} = \begin{bmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{bmatrix} \quad \mathbf{u} \otimes \mathbf{u} = \begin{bmatrix} u_x^2 & u_x u_y & u_x u_z \\ u_x u_y & u_y^2 & u_y u_z \\ u_x u_z & u_y u_z & u_z^2 \end{bmatrix}$$

Once the set of secondary sources has been created, each vector is “squeezed” according to the spread angle β as shown in Fig. 54. The computation of the squeezing process is shown in Listing 25. Each secondary source vector is multiplied by the tangent of the spread angle β and then added to the primary source vector to position it correctly.

```

305  /* Squeeze the perpendicular ring to desired spread */
306  let spread_rad = (spread/2)*(Math.PI/180);
307  let ring_rad = (spread_rad/num_rings_3d);
308  let u_spread = math.zeros(num_src*num_rings_3d,3).toArray();
309  for(let nr = 0; nr < num_rings_3d; nr++){
310      for(let j = 0; j<num_src; j++){
311          u_spread[j] = math.add(u, math.multiply(spreadbase[j+nr], math.tan(ring_rad*(nr+1))));
312      }
313  }

```

Listing 25: Code to squeeze vectors

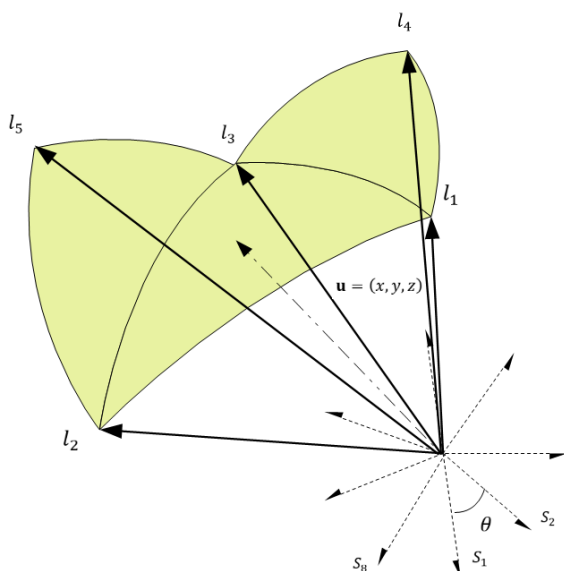


Fig. 53: Creating secondary sources

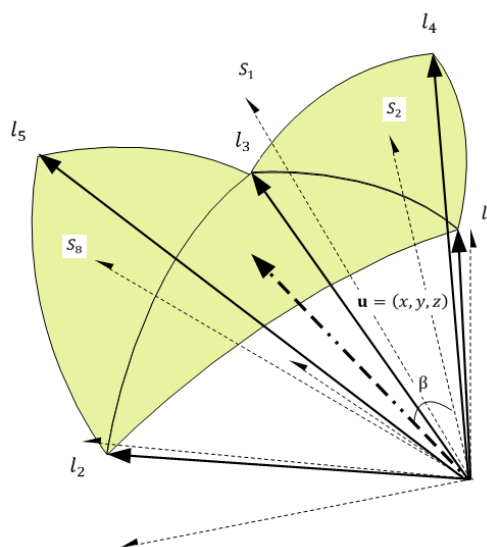


Fig. 54: Squeezing secondary sources

5.2.2 VBAP on the XMOS processor

VBAP required new variables to be added to the SAR and are shown in Listing 26. These are the static parameters that need to be set once for the VBAP calculation: a variable for the number of triangle faces in the convex hull, the vertices of the faces in the convex hull and the inverse loudspeaker matrices that correspond with those faces.

```

375  // In the SAR task ...
376
377  /* VBAP configuration */
378  char no_faces = 0; // Number of triangle faces
379  unsigned char triangle_faces[MAX_SIZE_CONFIG*3][3] = {{0}}; // Speaker IDs of triangle faces
380  double inv_LsMtx[MAX_SIZE_CONFIG*3][3][3] = {{0}}; // inverted loudspeaker matrices

```

Listing 26: VBAP configuration variables introduced to the SAR.

As shown in Table 12, ImmerGo filled these by sending AECP SET_MIXER commands with the descriptor index 0x0031. Once these were filled, the algorithm computed track sources that are sent with a descriptor index of 0x0030. Additional functions were added to the *avb_1722_1_control* internal interface so the Application task was able to make calls to SAR pertaining to these new functions.

Descriptor Index	Function
0x0030 : Perform VBAP	Perform VBAP for track source positions
0x0031 : Set VBAP Configuration	Set the VBAP inverted loudspeaker matrices and faces

Table 12: AECP SET_MIXER Descriptor Indices for the VBAP algorithm in the SAR.

```

2 // ... in avb_1722_1_control interface declaration
3 void set_VBAPConfig(unsigned char seq_id, unsigned char raw_config[]);
4 void perform_VBAP(unsigned char noSrcs, char src[]);

```

Listing 27: New functions added to the *avb_1722_1_control* internal interface.

5.2.2.1 Sending the VBAP configuration to the XMOS Processor

Before panning sources using VBAP, the SAR task is required to receive the VBAP configuration from ImmerGo. This is done similarly to sending the speaker configuration outlined in the previous chapter, except now floating point values are required to be received. This process is outlined in Fig. 55 below and described in greater detail in Appendix B B3. The descriptor index 0x0031 is used to address the correct function within the SAR. After experimentation, it was found that the maximum configuration size that could be sent at once to the XMOS was 468 bytes. Since the VBAP configuration is larger than the 468 bytes, the sending process needs to be split into multiple packets sent sequentially. A sequence ID is used to keep track of sequence order of the packet to ensure the packets are unpacked correctly on the XMOS side. Once the speaker configuration has been loaded and the VBAP configuration solved, the VBAP configuration is sent. First, the number of faces are sent with the sequence ID set to 0. This is then followed by the convex hull faces and the inverse matrices. If the number of bytes is greater than 468, the packet is split with appropriate sequence IDs.

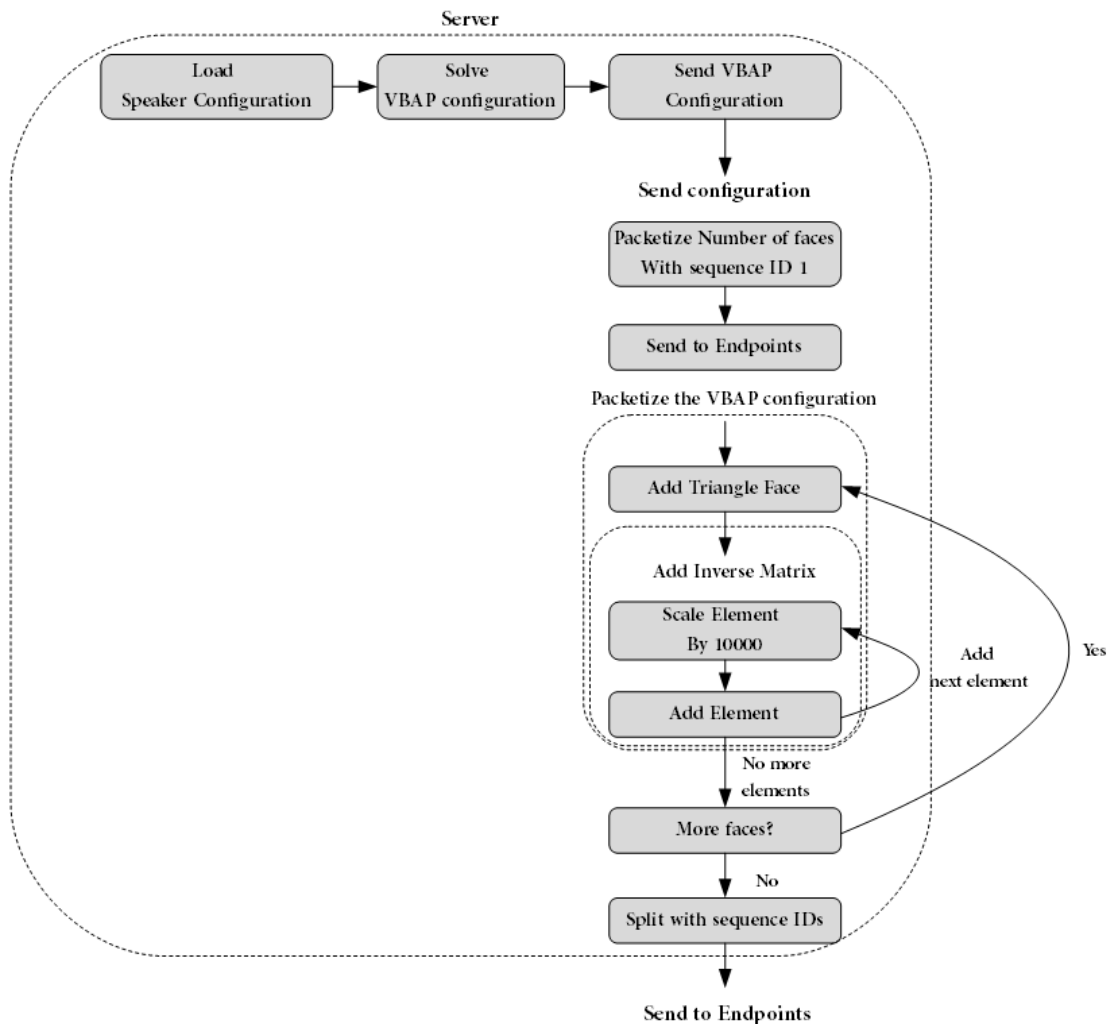


Fig. 55: Sending the VBAP configuration from the ImmerGo server.

The VBAP configuration is then received within the XMOS according to the procedure outlined in **Fig. 56** and shown in greater detail in Appendix B B4. When the AECP packet is received by the endpoint with the descriptor 0x0031 the packet is routed as follows.

1. The AECP packet headers are parsed within the 1722.1 task.
2. The descriptor index, along with the raw data, is passed from the 1722.1 task to the Application task using the *i_1722_1_entity* interface.
3. The Application task then routes the data to the “Set VBAP Configuration” function within the SAR according to the descriptor index 0x0031 using the *i_1722_1_control* interface.
4. If the sequence ID is 0, then the number of faces in the convex hull are saved.
5. If it is greater than 0, then the loudspeakers IDs associated with the face are saved followed by the inverted matrix associated with the face making sure to convert the inverted matrices back into a floating point by scaling down by 10000. This is then repeated for all the triangle faces in the convex hull.

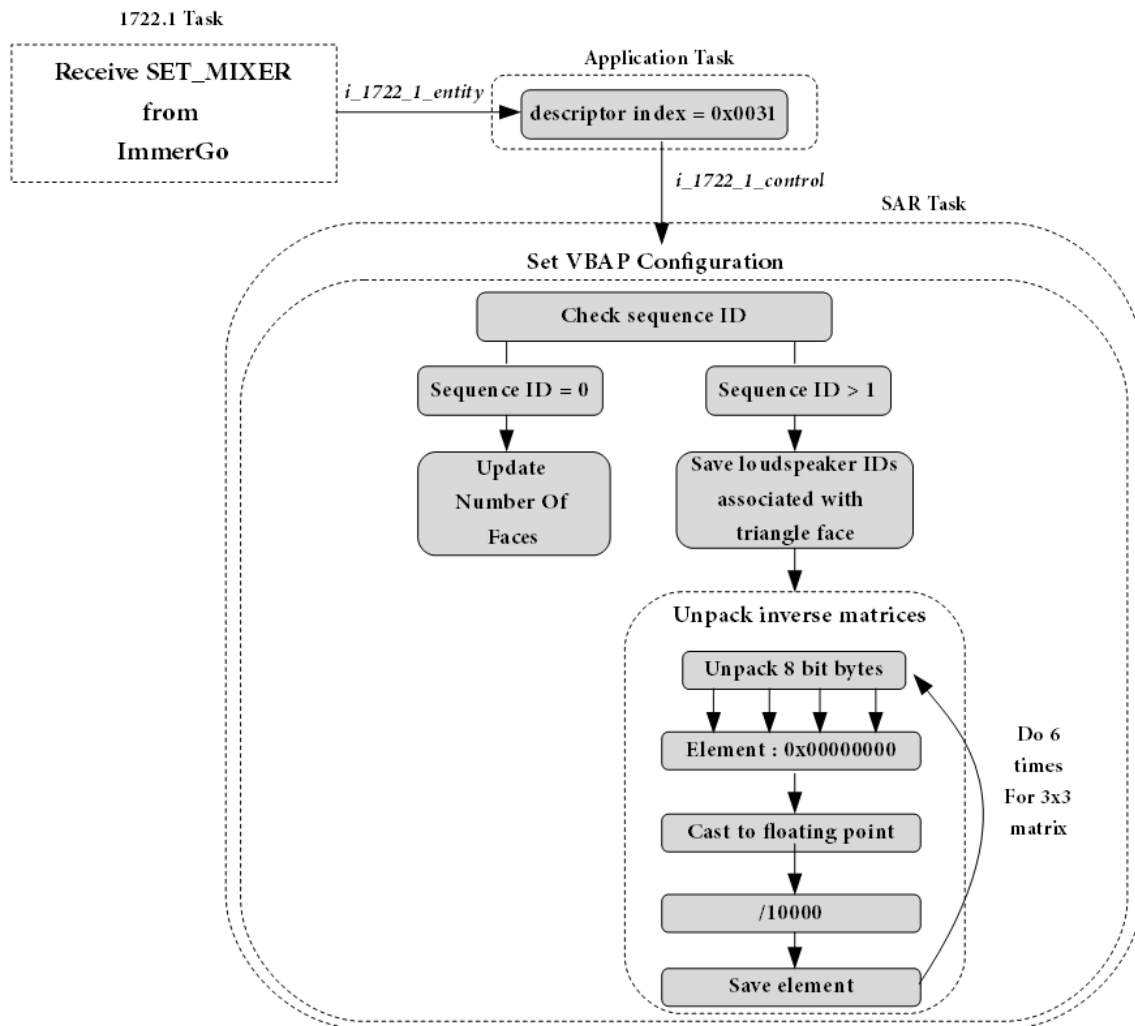


Fig. 56: Receiving the VBAP configuration within the SAR.

5.2.2.2 Performing the VBAP calculation within the XMOS

The XMOS processor performs the VBAP calculation using track source IDs and coordinates sent with the descriptor index 0x00300. As discussed previously in chapter 2, the track sources are sent according to the MTC received from the DAW. The coordinates are unpacked within the XMOS as discussed in chapter 3. This process is shown in greater detail in Appendix B B5.

Every endpoint performs the matrix calculation for each face until the correct face for the virtual source is found, as the 3 loudspeakers in the triangle $\mathbf{L}_{123} = [l_1 \ l_2 \ l_3]$ with their associated gains $\mathbf{g} = [g_1 \ g_2 \ g_3]$. Shown in Fig. 57, the gains are then assigned according to a loudspeaker selection process managed by a switch statement. For both the loudspeakers attached to the processor, the loudspeaker ID of the matrix calculation is checked against the ID of the loudspeaker attached to the processor. If they are the same, then the mixer value is set according to the corresponding gain value output from the matrix calculation, otherwise it's set to 0.

For the second step of distribution, the XMOS mixer matrix is required to be set according to the gain values outputted by the VBAP algorithm within the SAR. First, each gain value is scaled up from floating point to a signed char and added to the XMOS mixer values array in the SAR. A call to the Audio Buffer Manager is made from the SAR task using the *i_imr* interface which updates the XMOS mixer matrix according to these scaled values. This is not unlike the DBAP implementation discussed in the previous chapter.

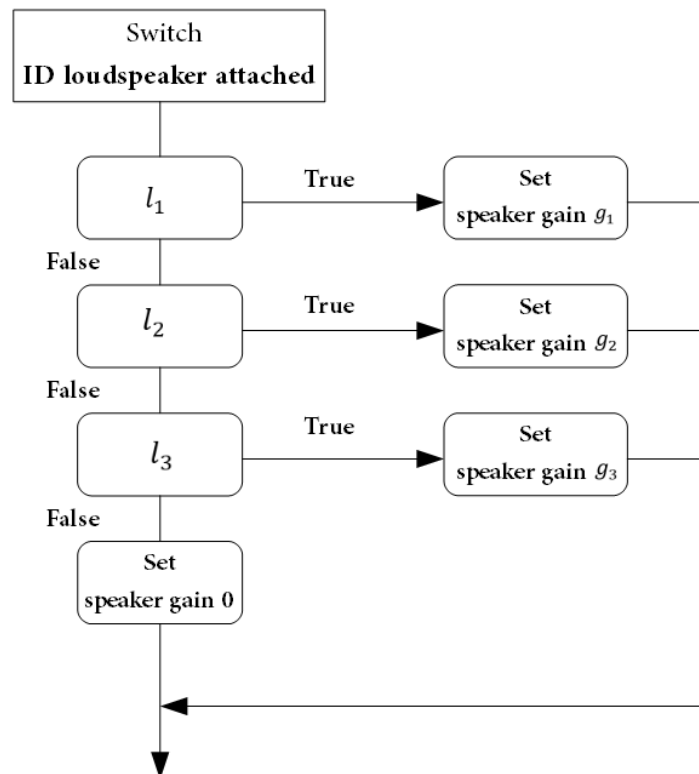


Fig. 57: Loudspeaker selection.

For the final level of distribution, the firmware with the SHARC mixer matrix is enabled and XMOS mixer matrix is removed. The VBAP calculation within the SAR is implemented the same, except now the output is used to update the SHARC mixer matrix. Similar to the method discussed for DBAP, once a track's mix values for each speaker are output from the algorithm, they are each converted into Q1.15 format. Since there is no delay processing required, the delay parameter is set to 0. These values are then combined into the 32 bit integer parameter value and sent to the SHARC mixer using the *DSP_AddrParaLoad* function.

MDAP requires that the secondary spread sources are first created, and then each of their gain values calculated by VBAP. As shown later in chapter 8, it was found that the time taken by the SAR to pan more than 8 sources simultaneously using VBAP exceeded the 8.3ms synchronization window imposed by the MTC quarter frame interval. This is not good for the case of MDAP, whereby a single source would require that multiple secondary sources are generated by creating multiple vectors and rotating them equally and squeezed to the desired spread angle. As well as this, once the sources are solved, the VBAP calculation is required to be done on each of those sources. For an MDAP solution within the XMOS processor, this amount of processing would take too much time and so an MDAP implementation within the XMOS processor is omitted.

One way to get around this processing limit is to create multiple processing tasks that manage the different aspects of the secondary source creation to try and increase efficiency. This method goes beyond the scope of this current project and as such an implementation has been omitted.

5.3 Summary

The theory behind VBAP has been discussed. The concept of stereophonic panning was extended to 2D VBAP which provided an understanding of the matrix calculations necessary to extend to 3D VBAP. Furthermore, this chapter covered an overview of the MDAP algorithm that allows for the spreading of sources using multiple secondary sources. This theory provided the information required to implement the algorithm at the three stages of distribution.

The VBAP algorithm is implemented within a library called "VBAP.js". This library provides all the functionality required to perform the VBAP algorithm. Processing power was saved by first determining the static parameters used by the VBAP matrix calculation known as the VBAP configuration. The configuration is specifically the convex hull of the array, its faces and their associated inverse loudspeaker matrices. When implementing VBAP calculations within the XMOS processor, the VBAP configuration was sent as a sequence of packets which were received by the SAR task. The panning process required a loudspeaker selection that matched the IDs of the speakers to that of the algorithm output. The mixing process for both iterations is managed the same as the DBAP algorithm in the previous chapters.

When spreading sources, the MDAP algorithm is used. This calculated values for multiple secondary sources situated in a ring around the primary position of the source being localized. Each of these sources, including the primary source, have their gains solved according to the VBAP algorithm and summed together. This was implemented within the VBAP library. Shown later in chapter 8, the VBAP algorithm becomes extremely processor intensive. As such, it was decided that since MDAP required even more processing time to allow for the generation of secondary sources which is then followed by the VBAP

calculation on each secondary source, it would not allow for an effective solution within the XMOS processor. For this reason, an MDAP solution within the XMOS processor has been omitted.

Chapter 6

Wave Field Synthesis

Wave field synthesis (WFS) is an object-based rendering algorithm that widens the listening area of spatial audio by synthesizing the physical wavefronts of objects in a scene. It is different from psycho-acoustical techniques such as VBAP that trick the mind into perceiving audio locality between sparsely spaced speakers (Ahrens, 2012). It employs a technique similar to the one used to project holographic images from light sources, and has thus been referred to as a ‘holophonic’ panning technique (Berkhout, 1988) (Berkhout, Vries, & Vogel, 1993). There are numerous commercial ²², as well as free ²³ WFS systems, and WFS has become a well-researched topic (Verheijen, 2010) (Firtha, 2018).

Its implementation is based on the Huygens’ principle (Ise, 1999), represented with the Kirchhoff-Helmholtz integral. For WFS, the Kirchhoff-Helmholtz is reduced to discrete loudspeakers using the Rayleigh integrals and the Green’s function that models how sound propagates through the air (Berkhout, 1988) (Berkhout, Vries, & Vogel, 1993) (Vries, 1996). Huygens’ principle states that a single wavefront may be regarded as a superposition of multiple smaller secondary wavefronts. Because of this, a virtual source’s wavefront may be synthesized by the contributions of multiple smaller source wavefronts generated by multiple loudspeakers. In practice, this results in using loudspeaker arrays that are arranged in a line in front of listeners, or a circle or rectangle around listeners. The location of the virtual source is then synthesized by emitting carefully delayed and attenuated copies of the virtual source signal across the loudspeakers which combine to generate the desired wavefront. The technique provides the ability to synthesize complete acoustic fields with no defined sweet spot, and gives a listener the opportunity to move around the acoustic field to pick up locational cues. Due to the requirement for a large number of loudspeakers, and the processing power associated with the loudspeakers, WFS has been minimally adopted. The systems that do exist often result in large complicated systems that rely on the synchronization of multiple processors. Theoretically, this makes WFS a strong candidate for the endpoint distributed solution.

6.1 Theory of Wave Field Synthesis

The goal of implementing the WFS calculation is to use the fundamental equations that represent Huygens’ principle, the Kirchhoff-Helmholtz integral, and derive a set of processing operations to be applied on a single audio track for a set of loudspeakers. These operations are determined according to the virtual source

²² - <http://www.iosono-sound.com/>
 - <https://www.barco.com/en/product/apx-audio-processor>
 - <https://www.astroaudio.eu/products/concept>

²³ - <https://github.com/GameOfLife/WFSCollider>
 - <https://github.com/mm Montag/wfs-designer>
 - <http://empac.rpi.edu/research/wave-field-synthesis>
 - <http://fouraudio.com/en/products/wfs.html>

and listening reference position with their associated 3D coordinates. The distance of the virtual source to each loudspeaker is then used to calculate for the mixer gain and delay parameters.

The Kirchhoff-Helmholtz integral provides a mathematical representation that models how points along a surface must behave to produce a wavefield within a measurable volume. It states that for reproducing a soundfield in a source-free listening area, it is sufficient to use a continuous distribution of monopole and dipole, secondary sound sources on a closed surface enclosing the listening area (Vetterli, 2009). Of course, in a real application of WFS, discrete loudspeakers do not allow for a continuous, infinite distribution of loudspeakers and so a number of approximations need to be made to reduce the integral from a general surface to discrete loudspeakers. This will be explained briefly in the following subsection.

The Kirchhoff-Helmholtz integral for a general closed surface is given below:

$$P(\mathbf{x}, \omega) = \oint_{\partial V} \left(G(\mathbf{x} - \mathbf{x}_0, \omega) \frac{\partial}{\partial \mathbf{n}} P(\mathbf{x}_0, \omega) - P(\mathbf{x}_0, \omega) \frac{\partial}{\partial \mathbf{n}} G(\mathbf{x} - \mathbf{x}_0, \omega) \right) \partial V \quad (6.1)$$

Where P is the pressure at the receiver position $\mathbf{x}$, ω is the angular frequency, or frequency spectrum, of the wave being transmitted, $\mathbf{x}_0$ is a position on the boundary surface, $\mathbf{n}$ is a normal vector for the surface at the point $\mathbf{x}_0$ (pointing inwards) and $\frac{\partial}{\partial \mathbf{n}} G(\mathbf{x}_0, \omega)$ is the directional gradient of the source in the direction of $\mathbf{n}$. $G(\mathbf{x} - \mathbf{x}_0, \omega)$ is a spatial temporal transfer function that models how sound travels through the air.

The geometry for equation (6.1) is shown in Fig. 58. Here, within a measurable volume, a sound source S exerts pressure variations, $P(\mathbf{x}, \omega)$, at a receiver position $\mathbf{x}$ that lies within an area ∂V . The receiver position may then be considered the position where a microphone R could be placed to receive the signal emitted by the source of the sound S . Now, the Huygens' principle tells us that we may calculate the pressure and velocity of the particles for every point $\mathbf{x}_0$ on the surface ∂V such that they can synthesize a wavefield exerted by S . In order to do so, one needs to use a spatial temporal function known as Green's function which describes the wavefront that would be emitted from $\mathbf{x}_s$ and all the points $\mathbf{x}_0$ on the surface.

Green's function defines the spherical wavefront of a monopole source placed at $\mathbf{x}_0$. A monopole, in reality, is an electrodynamic speaker (Verheijen, 2010). The Green's function is specifically used to calculate the disturbance of point $\mathbf{x}_0$ on a point $\mathbf{x}$ within air. For the general surface shown in Fig. 58, the 3D version of the Green's function may be used. This version of Green's function is given in equation (6.2) below:

$$G_{3D}(\mathbf{x} - \mathbf{x}_0, \omega) = \frac{1}{4\pi} \frac{e^{-j\frac{\omega}{c}|\mathbf{x}-\mathbf{x}_0|}}{|\mathbf{x} - \mathbf{x}_0|} \quad (6.2)$$

This function describes the sound pressure that a receiver at $\mathbf{x}$ would receive from $\mathbf{x}_0$. Specifically, this sound would be delayed by the time taken for the sound to travel from $\mathbf{x}_0$ to $\mathbf{x}$ according to the speed of sound, $\frac{|\mathbf{x}-\mathbf{x}_0|}{c}$ where c is the speed of sound, and attenuated proportionally to the distance of $\mathbf{x}_0$ to $\mathbf{x}$ according to the inverse square law. Depending on the dimensionality of the simulation environment, the Green's function must change.

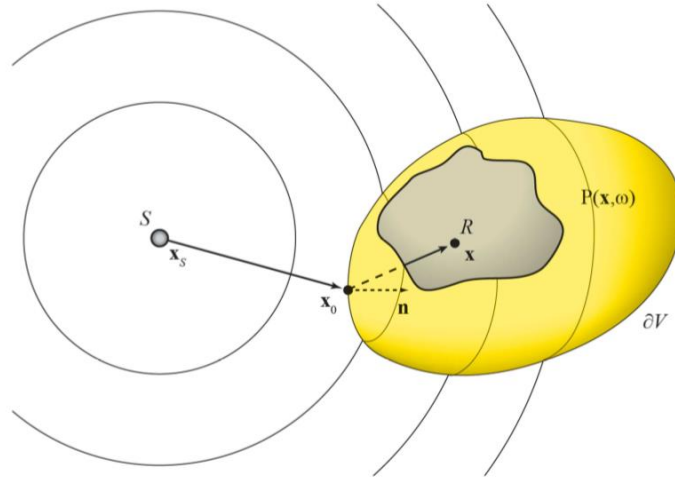


Fig. 58: Conditions of the Kirchhoff-Helmholtz integral. Diagram from (Montag, 2011).

The Kirchhoff-Helmholtz integral ensures that the volume outside of the surface is null. The only way to ensure this is to suppress the outgoing wave using a dipole source that emits a wave that is reverse to the one emitted within the surface. This is represented by the directional gradient of the source on the surface $\frac{\partial}{\partial \mathbf{n}} G$. In reality, a dipole source would be an electrostatic speaker (Verheijen, 2010).

In practical applications such as the one in this thesis, the outgoing wave does not need to be suppressed and the solution may be solved by omitting any dipole contributions. As such, the Kirchhoff-Helmholtz integral becomes the Rayleigh integral:

$$P(\mathbf{x}, \omega) = \oint_{\partial V} \left(G(\mathbf{x} - \mathbf{x}_0, \omega) \frac{\partial}{\partial \mathbf{n}} P(\mathbf{x}_0, \omega) \right) \partial V \quad (6.3)$$

6.1.1 Derivation of Loudspeaker Driving Signals

As described in (Berkhout, Vries, & Vogel, 1993), a series of steps need to be taken to approximate the Kirchhoff-Helmholtz to discrete loudspeakers. These are,

1. Approximate the general surface to a planar surface.
2. Approximate the planar surface to a horizontal line of continuous sources.
3. Finally, approximate the horizontal line to a line of discrete loudspeakers.

When applying the first two approximations, the Kirchhoff-Helmholtz integral degenerates into the Rayleigh I integral for monopole sources and the Rayleigh II integral for dipole sources (de Vries, 1996). Since we are concerned with monopole sources only, the Rayleigh I integral will be used. For a complete description of how the Rayleigh I integral is reduced to a line of discrete loudspeakers, the reader is directed to (Verheijen, 2010).

The Rayleigh I integral can be reduced from a plane to a line of continuous monopole sources by setting $z = 0$ (Verheijen, 2010). Now, the calculations are restricted to the horizontal plane but are still using 3D secondary sources represented by 3D Green's functions (Firth, 2018). This continuous line is then

approximated to discrete loudspeakers using the stationary phase approximation. To account for these approximations and ensure the amplitudes of the wavefronts are synthesized correctly, the Green's function receives a square root of the inverse distance as shown in equation (6.4).

$$G_{2.5D}(\mathbf{x} - \mathbf{x}_0, \omega) = \frac{1}{4\pi} \frac{e^{-j\frac{\omega}{c}|\mathbf{x}-\mathbf{x}_0|}}{\sqrt{|\mathbf{x} - \mathbf{x}_0|}} \quad (6.4)$$

This 2.5D derivation of the Green's function is explained in greater detail in (Verheijen, 2010, pp. 37-41) but for this thesis, it is sufficient for the reader to know this equation (6.4) is used to determine the speaker's driving function, or sometimes known as the synthesis operator.

Fig. 59 below shows the geometry of the source produced behind the loudspeaker array, that is received by a listener positioned at $\mathbf{r}$. Point sources in front of the array and plane waves will be given later in this chapter. Shown in equation (6.5), and described by the geometry shown in **Fig. 59**, the pressure $P(\mathbf{r}, \omega)$ at the receiver position would be the sum of the signals that would be emitted from each of the N loudspeakers transformed by the wavefront that would be emitted from the virtual source (de Vries, 1996). The driving function $Q(\mathbf{r}_n, \omega)$ is the process that would be applied to a signal ω at a loudspeaker $\mathbf{r}_n$ according to the 2.5D Green's function. $G(\theta_n, \omega) = 1$ for monopole sources, so vanishes in this equation. Δx is the spatial interval between each loudspeaker (de Vries, 1996).

$$P(\mathbf{r}, \omega) = \sum_{n=1}^N \left[Q(\mathbf{r}_n, \omega) G(\theta_n, \omega) \frac{e^{-jk|\mathbf{r}-\mathbf{r}_m|}}{\sqrt{|\mathbf{r} - \mathbf{r}_m|}} \right] \Delta x \quad (6.5)$$

The loudspeaker driving function $Q(\mathbf{r}_n, \omega)$ is derived from the stationary phase approximation method as shown in (de Vries, 1996). The stationary phase approximation physically means that the dominant contribution of the above discrete integral, is given by the loudspeaker positioned at the point of stationary phase, or on the same horizontal position as the virtual source. This is shown by the function given in equation (6.6) below. Here Q acts on the frequency spectrum ω of a source signal $S(\omega)$ for a speaker $\mathbf{r}_n$ according to a virtual source position $\mathbf{r}$.

$$Q(\mathbf{r}_n, \omega) = S(\omega) * \underbrace{\sqrt{\frac{jk}{2\pi}}}_{pre-eq} * \underbrace{\sqrt{\frac{|\mathbf{r} - \mathbf{r}_n|}{|\mathbf{r} - \mathbf{r}_n| + |\mathbf{r}_n - \mathbf{r}_m|}}}_{compensation\ factor} * \underbrace{\cos \theta_n \frac{e^{-jk|\mathbf{r}_n - \mathbf{r}_m|}}{\sqrt{|\mathbf{r}_n - \mathbf{r}_m|}}}_{delay\ and\ weighting} \quad (6.6)$$

where $\cos \theta_n$ is the directional weighting of the stationary phase approximation and is the angle of incidence to loudspeaker normal $\mathbf{n}$, $\sqrt{\frac{jk}{2\pi}}$ is a static pre-equalization filter that compensates for the discretization of the points, $\frac{1}{\sqrt{|\mathbf{r}_n - \mathbf{r}_m|}}$ is the weighting factor according to the distance of source to loudspeaker and $e^{-jk|\mathbf{r}_n - \mathbf{r}_m|}$ is the delay factor applied to the speaker according to the distance of source to loudspeaker. $\sqrt{\frac{|\mathbf{r} - \mathbf{r}_n|}{|\mathbf{r} - \mathbf{r}_n| + |\mathbf{r}_n - \mathbf{r}_m|}}$ is the compensation factor which ensures that the wave front amplitude is

synthesized correctly at the listener reference line which, in other words, ensures that the loudest volume of the synthesis is when a source is close to the listener reference line, and becomes softer as it moves further backward behind the array.

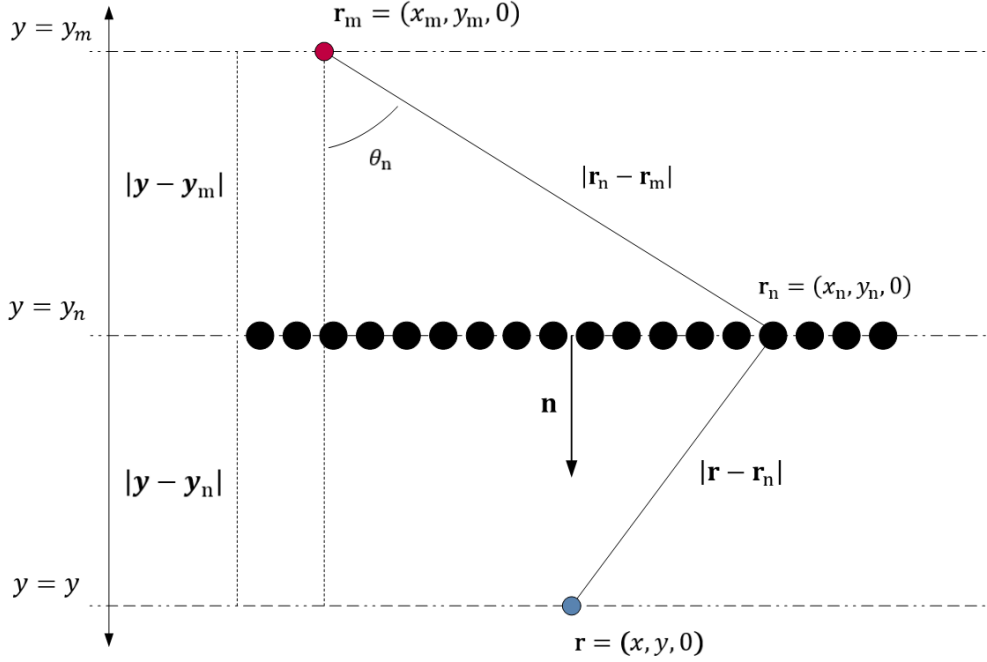


Fig. 59: Geometry for a point source behind the array.

Equation (6.6) also allows for the formulation of a general synthesis operator that can be applied to each loudspeaker position in the array. The general form of the synthesis operator is useful as the driving function is required to change depending on the type of source that is to be projected (Montag, 2011). The general form of the operator for each speaker $\mathbf{r}_n$ is given in the frequency domain by

$$D(\mathbf{x}_n, \omega) = S(\omega) \cdot H(\omega) \cdot w(\mathbf{x}_n) \cdot e^{-j\omega\tau_0(\mathbf{x}_n)} \quad (6.7)$$

where $S(\omega)$ is the spectrum of the source signal, $H(\omega)$ is a static pre-equalization filter, $w(\mathbf{r}_n)$ is the amplitude weighting coefficient and $e^{-j\omega\tau_0(\mathbf{r}_n)}$ is the delay factor according to the position of $\tau_0(\mathbf{r}_n)$

This can be translated into a time domain which is more useful when engineering the WFS system. Here the synthesis operator $d(\mathbf{r}_n, t)$ for a loudspeaker $\mathbf{r}_n$ acting on a source signal s at sample t

$$d(\mathbf{r}_n, t) = s(t) * h_{\text{pre}}(t) * w(\mathbf{r}_n) * \delta(t - \tau_0(\mathbf{r}_n)) \quad (6.8)$$

where δ is the delay factor that delays the signal according to $\tau_0(\mathbf{r}_n)$. τ_0 is a function dependent on the virtual source type. $w(\mathbf{r}_n)$ is the amplitude weighting function and $h_{\text{pre}}(t)$ is a static low pass filter that accounts for the discretization caused by the speakers.

6.1.2 Virtual Source Classification

WFS has three types of virtual sources that may be synthesized. These are seen in Fig. 60. As described in the introduction, they are all synthesized on continuous arrays (i.e. circular or rectangular) or discrete linear line arrays, however, the scope of the research in this thesis is for discrete linear line arrays, and so the driving functions for speakers in these arrays will be considered. The source types are:

- A non-focused spherical source that is behind the array.
- A focused spherical source that is positioned in front of the array.
- A plane wave source originates from an infinite distance behind the array.

All the notation for the remainder of this chapter will be derived from **Fig. 59** unless otherwise specified.

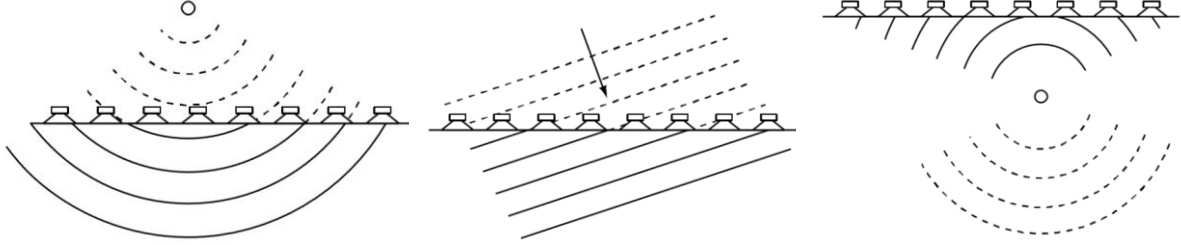


Fig. 60: Spherical non-focused source, plane wave source and focused source. Diagram from (Montag, 2011).

6.1.2.1 Plane Wave Source

Plane waves are naturally found as sounds that originate from far in the distance, for example, a thunderstorm in the distant horizon. To synthesize a plane wave that is travelling perpendicular to the array, no delays or gain level attenuation are required to be applied to the source signal at the speakers. To synthesize a plane wave coming from a particular direction, the delays are such that the wavefront will resemble straight lines that are perpendicular to the angle of that direction to the speaker array normal as shown in **Fig. 60**.

Defined in (Spors, Hahn, & Winter, 2017), the driving function for the plane wave is determined according to the angle of the normal of the plane wave to the normal of the speaker.

$$d(\mathbf{r}_{pw}, t) = s(t) * h_{pre}(t) * \underbrace{\sqrt{|\mathbf{r} - \mathbf{r}_n|} * \cos(\theta_n)}_{\text{weighting factor}} * \underbrace{\delta\left(t - \frac{\cos(\theta_n)}{c}\right)}_{\text{delay amount}} \quad (6.9)$$

Here the $h_{pre}(t)$ equalization filter amounts to the pre-equalization filter applied to all speakers according to the loudspeaker spacing, θ_n is the angle of incidence of the plane wave to the loudspeaker array normal, $\sqrt{|\mathbf{r} - \mathbf{r}_n|}$ is the compensation factor for the listener position and $\delta\left(t - \frac{\cos(\theta_n)}{c}\right)$ is the delay of the sample according to $\frac{\cos(\theta_n)}{c}$ where c is the speed of sound.

6.1.2.2 Non-focused Source

A non-focused source is a point source that originates from behind the array but still has a perceivable location which is the same as the example discussed in section 6.1.1. This is shown in **Fig. 61**.

The driving function for this source as defined in (Spors, Rabenstein, & Ahrens, 2008) is defined in (6.10).

$$d(\mathbf{r}_n, t) = s(t) * h_{pre}(t) * \underbrace{\sqrt{\frac{|\mathbf{r} - \mathbf{r}_n|}{|\mathbf{r} - \mathbf{r}_n| + |\mathbf{r}_n - \mathbf{r}_m|}} * \frac{\cos(\theta_n)}{\sqrt{|\mathbf{r}_n - \mathbf{r}_m|}}}_{\text{weighting factor}} * \underbrace{\delta\left(t - \frac{|\mathbf{r}_n - \mathbf{r}_m|}{c}\right)}_{\text{delay amount}} \quad (6.10)$$

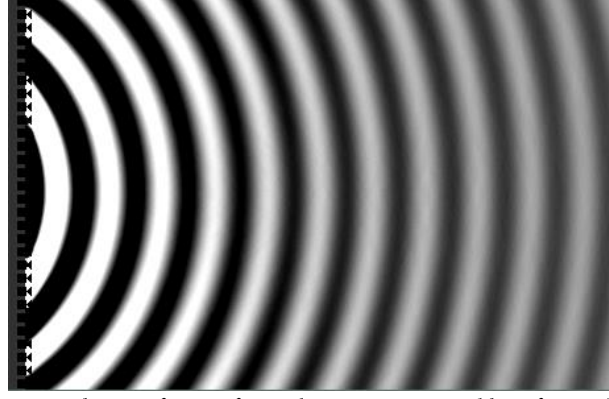


Fig. 61: Simulation of a non-focused source. Generated by wfs-visualizer.²⁴

The weighting factor consists of two components. First, the actual contribution of the loudspeaker's wavefield to generate the source, $\frac{\cos(\theta_n)}{\sqrt{|\mathbf{r}_n - \mathbf{r}_m|}}$. Here θ_n is the angle of incidence of the source to the loudspeaker normal and $\frac{1}{\sqrt{|\mathbf{r}_n - \mathbf{r}_m|}}$ is the inverse distance contribution of the loudspeaker to the virtual source position defined by Green's function. The second weighting component is listener position compensation factor $\sqrt{\frac{|\mathbf{r} - \mathbf{r}_n|}{|\mathbf{r} - \mathbf{r}_n| + |\mathbf{r}_n - \mathbf{r}_m|}}$ and is applied so the amplitude is correct at the listening position. $\delta\left(t - \frac{|\mathbf{r}_n - \mathbf{r}_m|}{c}\right)$ is the delay of the sample according to $\frac{|\mathbf{r}_n - \mathbf{r}_m|}{c}$, where c is the speed of sound.

As discussed in (Verheijen, 2010) and shown in (de Vries, 1996), if a linear line array is used, the compensation factor can be substituted for $\sqrt{\frac{|y - y_n|}{|y - y_m|}} = \sqrt{\frac{|y - y_n|}{|y - y_n| + |y_n - y_m|}}$. This makes the amplitude correction independent of listener position, whilst still correctly reconstructing the amplitude along the line of the listener position.

6.1.2.3 Focused Source

A focused virtual source is a point source that is projected between the speakers and the listener reference line as shown in Fig. 62. It is similar to a non-focused source except that the delay amount is inverted and the sign changed in the denominator of the distance compensation of the source to loudspeakers (Verheijen, 2010). Equation (6.11) shows the driving function for each loudspeaker.

$$d(\mathbf{r}_n, \omega) = s(t) * h_{pre}(t) * \underbrace{\sqrt{\frac{|\mathbf{r} - \mathbf{r}_n|}{|\mathbf{r} - \mathbf{r}_n| - |\mathbf{r}_n - \mathbf{r}_m|}} * \frac{\cos(\theta_n)}{\sqrt{|\mathbf{r}_n - \mathbf{r}_m|}}}_{\text{weighting factor}} * \underbrace{\delta\left(t + \frac{|\mathbf{r}_n - \mathbf{r}_m|}{c}\right)}_{\text{delay amount}} \quad (6.11)$$

Similar to non-focused sources, the weighting factor consists of two components, first the actual contribution of the loudspeaker's wavefield to generate the source $\frac{\cos(\theta_n)}{\sqrt{|\mathbf{r}_n - \mathbf{r}_m|}}$. The listener position

²⁴ <https://github.com/mm Montag/wfs-visualizer>

compensation factor $\sqrt{\frac{|\mathbf{r}-\mathbf{r}_n|}{|\mathbf{r}-\mathbf{r}_n|+|\mathbf{r}_n-\mathbf{r}_m|}}$ is applied so the wave field amplitude is loudest at the listening position. $\delta\left(t + \frac{|\mathbf{r}_n-\mathbf{r}_m|}{c}\right)$ is the inverted delay according to $\frac{|\mathbf{r}_n-\mathbf{r}_m|}{c}$ where c is the speed of sound.

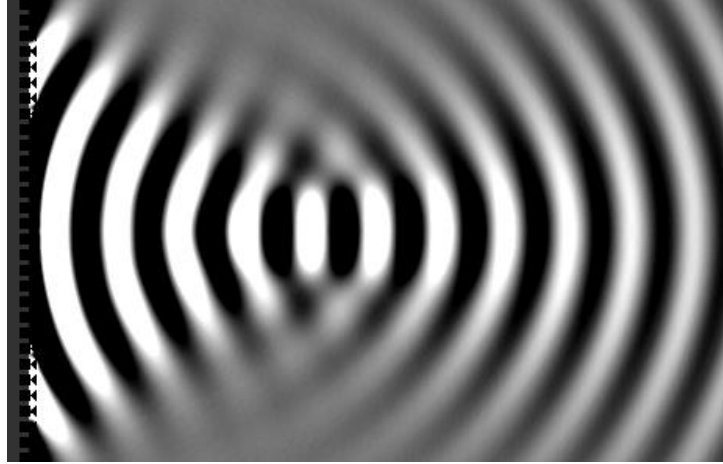


Fig. 62: Simulation of a focused source. Generated by wfs-visualizer.

6.1.2.4 Generalized WFS Focusing Operator

A method was introduced in (Verheijen, 2010) that calculates WFS for focused and non-focused sources on a linear array using a single function. This function is known as the generalized synthesis operator. It introduces a signed variable ζ which corresponds to the ratio between the y coordinates of the reference listening position and the virtual source such that $\zeta = \frac{y}{y_m}$. ζ is signed such that the $\text{sign}(\zeta)$ is positive for a focused source and negative for a non-focused source. This variable can be used to solve calculations for both equation (6.10) and (6.11) in a single equation given in (6.12).

$$d(\mathbf{r}_n, \omega) = s(t) * h_{pre}(t) * \underbrace{\sqrt{\frac{\zeta}{\zeta-1}} * \frac{\cos(\theta_n)}{\sqrt{|\mathbf{r}_n-\mathbf{r}_m|}}}_{\text{weighting factor}} * \underbrace{\delta\left(t + \text{sign}(\zeta) \frac{|\mathbf{r}_n-\mathbf{r}_m|}{c} * 48\right)}_{\text{delay amount}} \quad (6.12)$$

6.2 Known Problems with Wave Field Synthesis

The approximations of WFS to discrete loudspeakers cause certain artefacts to arise. In particular, these artefacts are spatial aliasing caused by discrete loudspeaker spacing and the truncation effect caused by truncation of the loudspeaker array to a finite distance. Fortunately, there are solutions to these problems, although the solutions do not completely solve these problems.

6.2.1.1 Spatial Aliasing

Spatial aliasing is caused by the fragmentation of the soundfield due to the reproduction surface being composed of discrete loudspeakers instead of continuous sources. The summed wave is made up of discrete wave components produced by the loudspeakers. As such, frequencies that have a wavelength shorter than the loudspeaker spacing become incoherent as they sum incorrectly. In reality, spatial aliasing is perceived as a position-dependent comb filtering that causes high frequency modulation and “flutter” echoes as the

listener moves around the listening area (Lopez, Bleda, Escolano, & Pueo, 2005) (Montag, 2011) however, some authors report that these artefacts are not very noticeable when listening (Verheijen, 2010). The spatial aliasing effect can be illustrated in Fig. 64 on the following page. It is traditionally given according to the Nyquist frequency of the spatial sampling process given by microphones (Corteel, Kuhn-Rahloff, & Pellegrini, 2008).

For a linear array, the aliasing frequency is dependent on the combination of angular direction of the plane wave to the listener $\theta_{max,\psi}$ and the maximum angular direction of the plane wave to the farthest loudspeaker in the array $\theta_{max,ls}$ (Corteel, Kuhn-Rahloff, & Pellegrini, 2008). This is shown in Fig. 63 below.

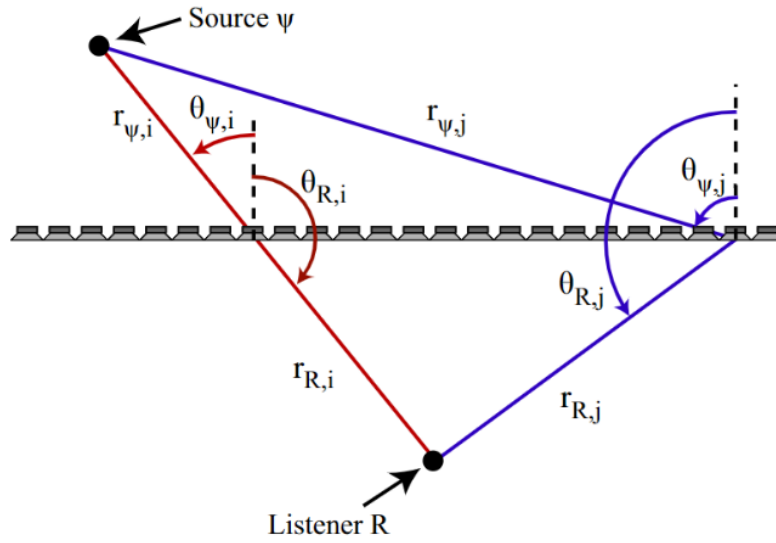


Fig. 63: Aliasing frequency. Taken from (Corteel, Kuhn-Rahloff, & Pellegrini, 2008)

This is expressed in equation (6.15).

$$f_{alias} = \frac{c}{2\Delta x (|\sin(\theta_{max,\psi})| + |\sin(\theta_{max,ls})|)} \quad (6.13)$$

It can be determined that the frequency for which the aliasing will occur is directly related to the loudspeaker spacing (Spors & Rabenstein, 2006). For a circular array, it can be determined that the frequency cut off for spatial aliasing of a linear array is given by:

$$f_{alias} = \frac{c}{2\Delta x} \quad (6.14)$$

where c is the speed of sound and Δx is the loudspeaker spacing.

It has been suggested by many authors that a frequency of 1500Hz is sufficient for accurate soundfield reproduction (Verheijen, 2010) (Vogel, 1993). According to (6.14), assuming $c = 343\text{m/s}$, a loudspeaker spacing of at most 11cms is required to prevent aliasing under 1500Hz.

There have been various solutions recommended to help overcome this problem. These solutions work in particular use cases and thus do not provide a complete solution. Some of the solutions that are relatively simple to implement are:

- To crossover the audio signal at the point of aliasing and let one or two speakers play the high frequency whilst the others have WFS rendering applied (Lopez, Bleda, Escolano, & Pueo, 2005) (Wittek, 2002).
- To use a spatial aliasing filter which reduces the bandwidth of each speaker's contribution (Theile, Wittek, & Reisinger, 2003) (Start, 1997).

These solutions are beyond the scope of this research and will not be included in the implementation.

6.2.1.2 Truncation Effect

The 2.5D Rayleigh I integral assumes that there is an infinite line of secondary sources. In practical implementations, this is not possible unless the loudspeaker array is distributed as a closed contour around listeners (Spors, Rabenstein, & Ahrens, 2008). As such this problem causes diffraction of the synthesized wavefield. As shown in Fig. 65 the solution to this problem requires tapering the gains of the speakers closer to the end of the array which prevents their individual wave from overpowering the synthesized wavefront. This does have the consequence of lessening the total listening area, however, with a large enough array, this effect is relatively minimal.

A variety of different tapering windows can be used such as a Hann window (Ahrens, 2012) (Spors, Wierstorf, Geier, & Ahrens, 2009).

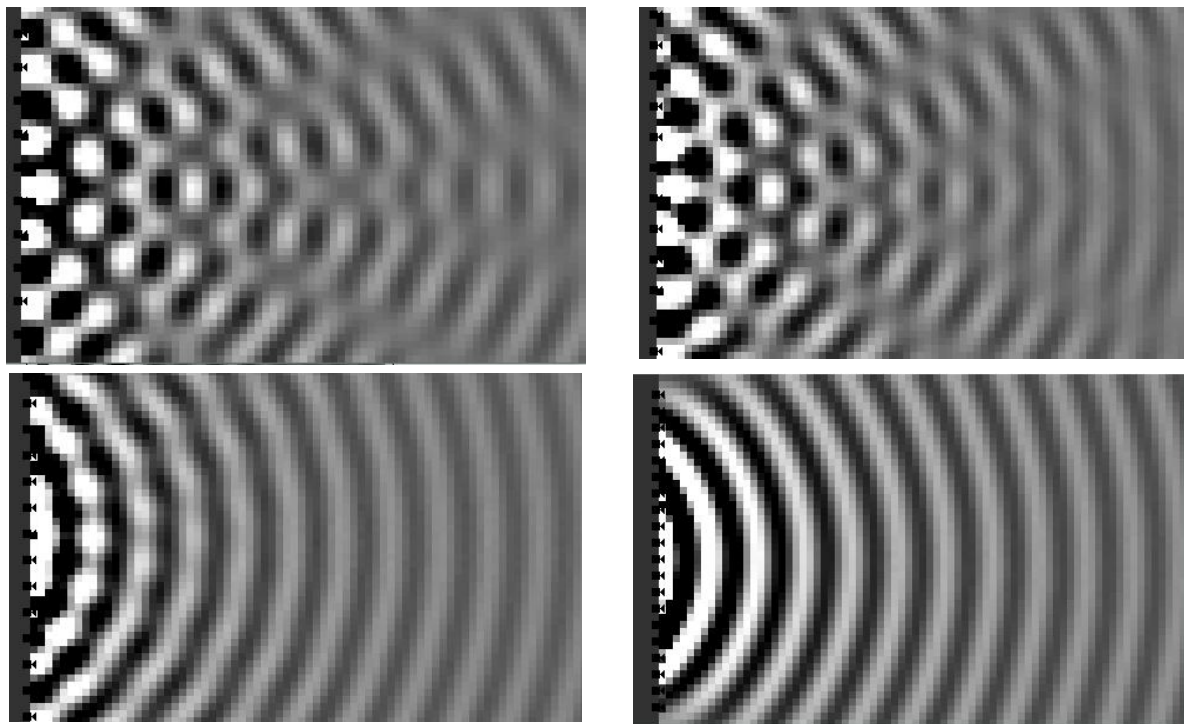


Fig. 64: Spatial aliasing that reduces as loudspeaker spacing gets narrower. Generated by wfs-visualizer.

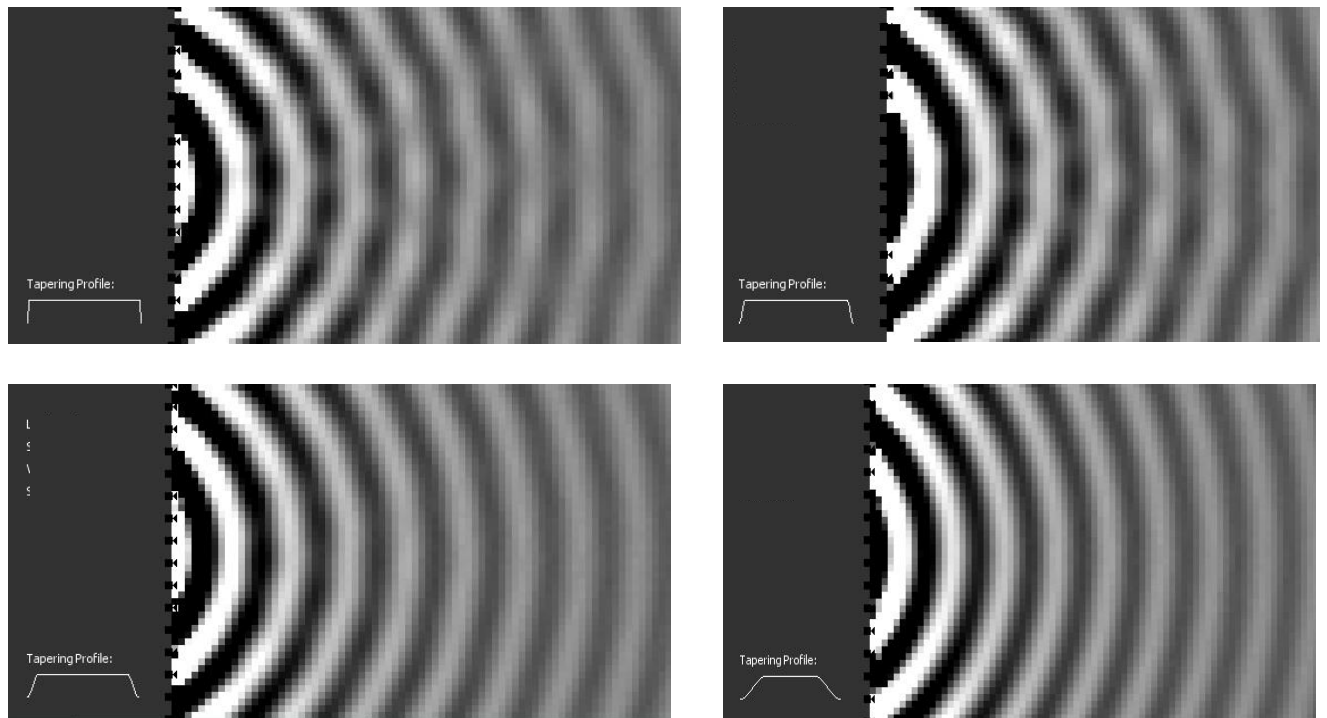


Fig. 65: Hann tapering window. Generated by wfs-visualizer.

6.3 WFS within ImmerGo

WFS was implemented within ImmerGo to synthesize static, focused and non-focused sources using a linear array of loudspeakers. In order to synthesize moving sources, a more complex processing implementation would need to be used. In particular, adaptive filtering and sub-sample timing accuracy would need to be implemented using convolutional filters which would require more memory and different processing blocks on the distributed processors (Roginska & Geluso, 2018). This is beyond the scope of the current research and is instead included in future work.

For the scope of this thesis, it was decided that it is sufficient to compare the processing power required by the algorithm to generate static sources using a linear speaker configuration. Spherical sources were implemented and tested which provide a good relative comparison as the calculations are more expensive than the ones used for plane wave simulation.

The WFS algorithm is different from the previous object-based algorithms, DBAP and VBAP, as it relies on delaying the virtual source signal to synthesize its wavefronts. As such, the mixer matrix within the XMOS processor was modified to incorporate delay processing, as well as attenuation processing, for each crosspoint. These processes are controlled with parameters sent from ImmerGo, for the first step of distribution, and the SAR for subsequent steps of distribution. The SAR task was modified to include functions that set the delay and attenuation of both the XMOS mixer and, for the final step of distribution, the SHARC mixer. The server was also modified to incorporate sliders that controlled filters within the SHARC.

As shown in **Fig. 66**, the ImmerGo UI was modified to represent the linear array of speakers as a solid line that runs across the centre of the room. This enables the virtual source to go both in front and behind the

speaker array to localize focused and non-focused sources. The listener reference position is represented as a small red dot in front of the array. All track sources are limited so they cannot leave the area of the loudspeaker, or go behind the listening reference position.

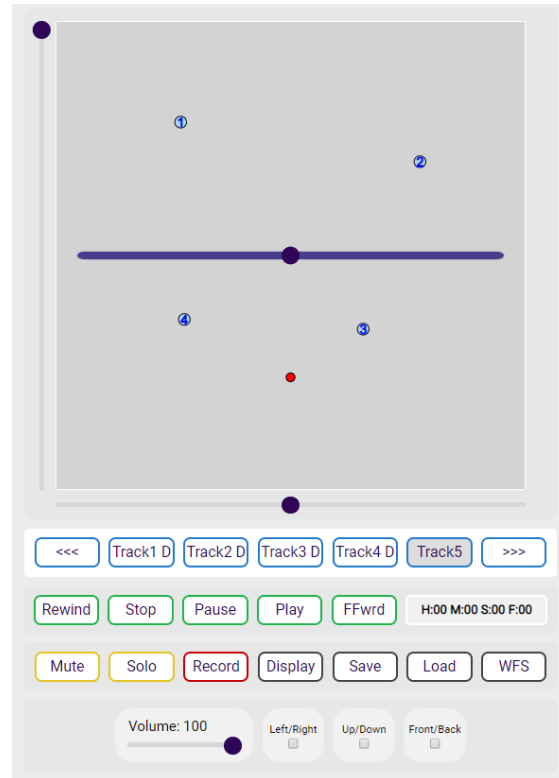


Fig. 66: ImmerGo WFS Client.

6.3.1 Modification to the XMOS Mixer matrix

The XMOS mixer has the delay-line implemented as a circular buffer of 1024 samples. This allows for a maximum delay of 21.3ms. As discussed later in the section, WFS requires inverted delays which means that a source may be delayed 10.7ms either side of the array. In other words, sources can move up to a limit of 3.7m on either side of the array assuming the speed of sound to be 343m/s. It has been shown in (Ahrens, Geier, & Spors, 2010) that the accuracy of a fractional delay line is unnecessary for WFS localization of sources that do not move. As such, an integer-based delay line is implemented as it is less processor intensive. So if a delay parameter for the read pointer lies between two samples, the earlier sample is chosen.

The implementation of the delay-attenuate mixer is shown in **Listing 28**. Circular delay buffers, with their associated write indices, are made for each matrix crosspoint. The *delay_proc* function within the mixer matrix takes the matrix crosspoint index and writes the incoming sample to the crosspoint's own circular buffer. The delayed sample is then given by reading from earlier in the buffer according to the delay parameter within the *XMOSDelayVals*. The output of the process is then attenuated according to the mixer levels held in the *XMOSMixerVals*.

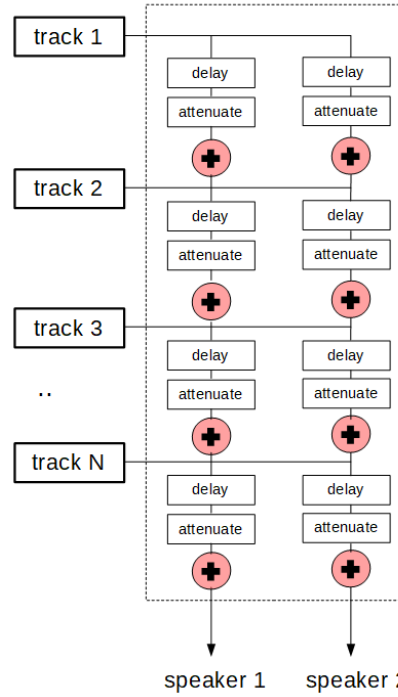


Fig. 67: Delay and attenuate mixer matrix.

```

112 #define MAX_DELAY_SAMPLES 1024
113 int32_t delay_bufs[MAX_SPEAKERS][MAX_CHANNELS][MAX_DELAY_SAMPLES];           // Buffers for delayed channels
114 int delay_buf_wr_idx[MAX_SPEAKERS][MAX_CHANNELS] = {{0}};                   // Sample write index for each buffer
...
130 unsigned int XMOSDelayVals[MAX_SPEAKERS][MAX_CHANNELS] = {{0}};             // Mixer delay parameters
...
268 /* Delay and mix samples */
269 #pragma loop unroll
270 for(int i=0; i<MAX_SPEAKERS; i++){
271     samples_out[i] = 0;
272     #pragma loop unroll
273     for(int j=0; j<MAX_CHANNELS; j++){
274         samples_out[i] += XMOSMixerVals[i][j]*delay_proc(i,j,sample_out_buf[j],XMOSDelayVals[i][j]);
275     }
276 }
277
278 c_audio <: (int32_t *unsafe)&samples_out;
279

```

Listing 28: XMOS delay summation mixer matrix in the Audio Buffer Manager task.

The use of the circular buffer is demonstrated in Fig. 68 and Fig. 69, for non-focused and focused sources respectively. The delay parameter δ_0 is offset by half the delay line, 512 samples, such that the starting delay, t_n , is always 512 samples before the write pointer. This ensures that when the delay parameter is positive, it is still able to read a sample correctly from the buffer. For a non-focused source, the sample is written into t_{n+5} and then the read pointer reads from t_{n-3} . When *delay_proc* is called again with the same delay argument, the sample write pointer increments to t_{n+6} , and as such the read pointer will also increment so the sample output is read from t_{n-2} .

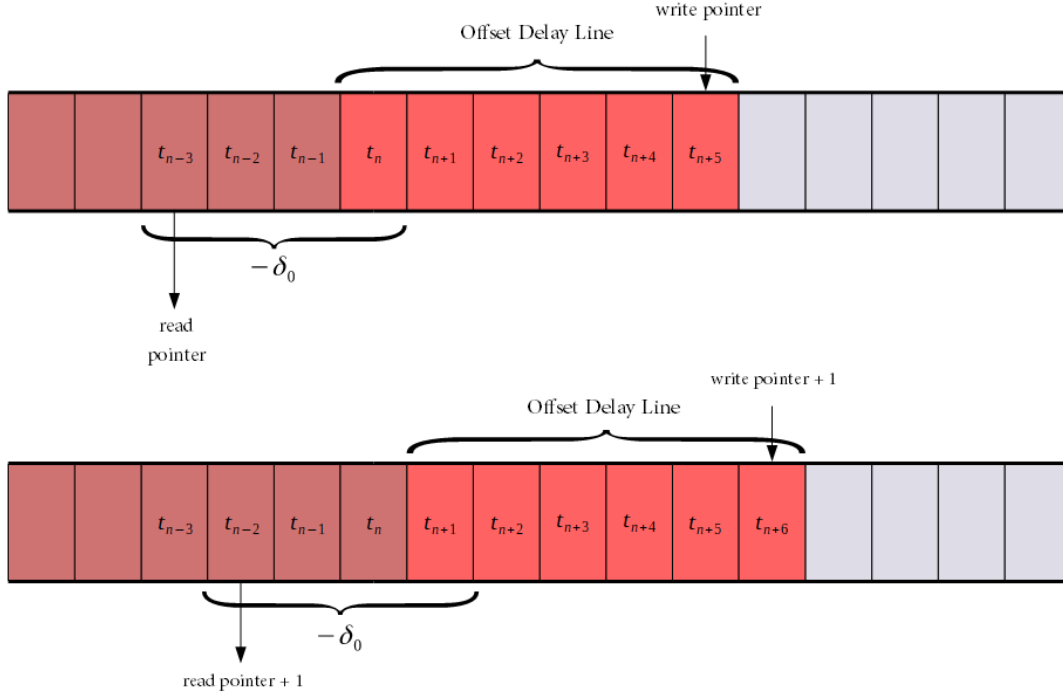


Fig. 68: Delay buffer for a non-focused source.

A focused source inverts the delay such that sources that are furthest from a speaker will be played from the speaker first. This results in a positive delay parameter, $+\delta_0$. As shown in Fig. 69, the delay line offset ensures that a larger sample delay, which results from a greater distance, is read closer to the write pointer, whilst a smaller delay in samples, from smaller distances, is read further from the write pointer. As such, when a focused source is delayed with the same parameter as the example above, the sample will be read from t_{n+3} which is closer to the write pointer.

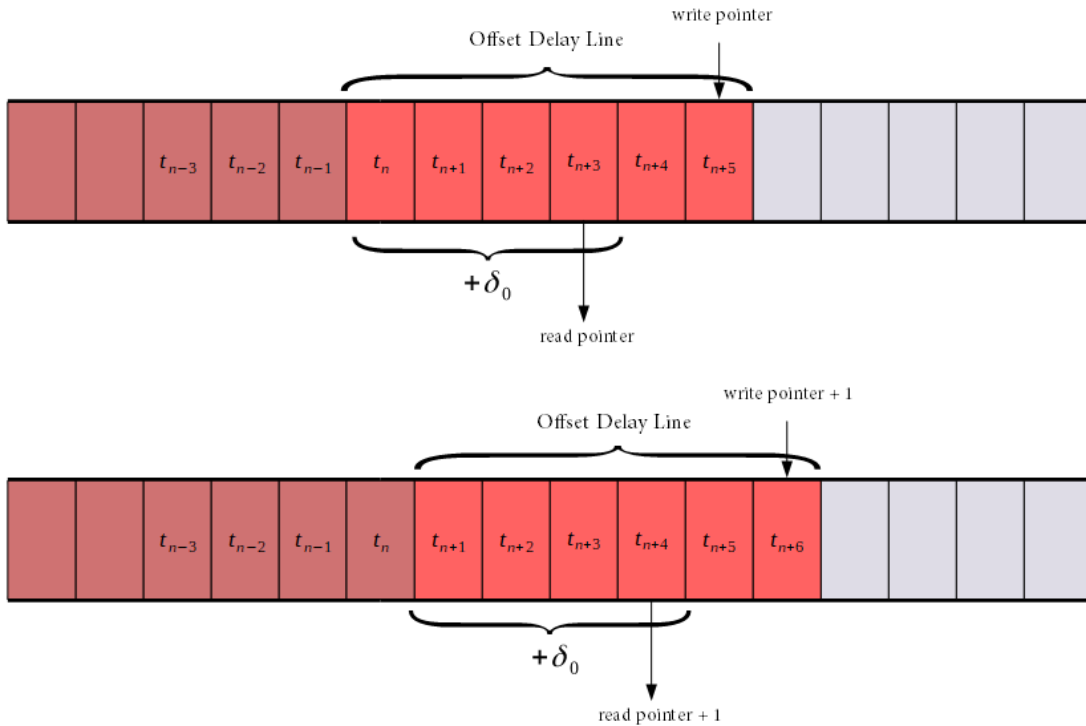


Fig. 69: Delay buffer for a focused source.

The delay summation mixer matrix was limited to 8 channels because of the memory constraints of the XMOS processor within the SPK-4P. It is controlled similarly to the XMOS attenuation mixer - when the SAR XMOS mixer parameters are changed, a call to the Audio Buffer Manager *i_imr* interface server is made using the *i_imr* client, holding the updated delay and attenuation values. These parameters are then copied into the parameter arrays held within the Audio Buffer Manager task as shown in **Listing 29**.

```

233     case i_imr.send_XMOSgain_delay_vals(unsigned y, unsigned x) :
234         unsafe{
235             char * unsafe prcv = (void*) x;
236             unsigned int * unsafe prcvDel = (void*) y;
237             memcpy(XMOSMixerVals,prcv,sizeof(XMOSMixerVals));
238             memcpy(XMOSDelayVals,prcvDel,sizeof(XMOSDelayVals));
239         }
240         break;

```

Listing 29: Receiving XMOS mixer gain and delay updates from SAR in the Audio Buffer Manager task.

6.3.2 Modifications to the SAR task

The SAR task incorporated new variables used for the WFS calculations. These are shown in **Listing 30**. The *XMOSDelayVals* holds the delay parameters that are sent to the XMOS mixer. The *delays* array holds the sample delay values used to set the SHARC mixer matrix. The *filter_coeff* array holds the floating point filter coefficient values for the SHARC mixer matrix. The WFS calculations required the listening reference position, a flag to note if the source is focused or non-focused and the tapering window array to taper the output of the speakers. The *dist_list_spks* holds the distances of the listener to each of the speakers in the configuration. The XMOS calculates these values prior to receiving track sources to save processing power.

```

367     /* Mixer parameters */
368     double delays[MAX_CHANNELS*MAX_SPEAKERS] = {0};           // Delays for each source and speaker
369     double filter_coeff[2] = {0};                             // Coefficients for SHARC filters
370
371     /* XMOS mixer parameters */
372     unsigned int XMOSDelayVals[MAX_CHANNELS*MAX_SPEAKERS] = {0}; // XMOS mixer delay parameters
373
374     /* WFS specific variables */
375     double listening_pos[3] = {0};                             // Listening position
376     double dist_list_spks[MAX_SIZE_CONFIG] = {0};             // Distance of listener to each speaker
377     unsigned char isfocused = 0;                                // Source in front or behind the array
378     double tapering_window[MAX_SPEAKERS] = {0};               // Tapering window for linear arrays

```

Listing 30: Variables added to the SAR task.

The functions that use these variables were added to the SAR task and are addressable from ImmerGo using AECp SET_MIXER commands with the appropriate indices shown in Table 13. Towards the first step of processing distribution, the SAR task incorporates functions that set both the XMOS mixer delay and gain values based on commands sent from ImmerGo. These are managed using either 0x0002 or 0x0003 descriptor indices.

For further distribution, the listener position is required to be sent to the SAR task prior to track sources using the 0x0011 descriptor index. Tapering weightings are able to be set using the 0x0041 descriptor. Descriptor index 0x0040 is then used to synthesize track source locations using WFS.

The final distribution step allows for the SHARC filters to be used which helps prevent tearing of the audio as the delay line is changed while audio is being played. Descriptor index 0x0007 sets these filter

coefficients. Functions to control the SHARC mixer matrix crosspoints have also been added, but are not specifically required for the experiments in this thesis. All of the above functionality is added to the *avb_1722_1_control* internal interface so that the Application task is able to route the data to the SAR correctly. The functions that do this are shown in **Listing 31**.

Descriptor Index	Function
0x0002 : Set XMOS Mixer Delay Values	Sets the XMOS mixer matrix delay values
0x0003 : Set XMOS Mixer Gain and Delay Values	Sets the XMOS mixer matrix gain and delay values
0x0005 : Set SHARC Mixer Delay Values	Sets the SHARC mixer matrix delay values
0x0006 : Set SHARC Mixer Gain and Delay Values	Sets the SHARC mixer matrix gain and delay values
0x0007 : Set SHARC Mixer Filter	Sets the SHARC mixer matrix filter values
0x0011 : Set Listener Position	Set the Spatial Audio Renderer Listening Position
0x0040 : Perform WFS	Performs WFS
0x0041 : Set WFS Tapering Window Weights	Sets the tapering weights for WFS array

Table 13: AECp SET_MIXER Descriptor Indices for the operations within the SAR.

```

3 // ... in avb_1722_1_control interface declaration
4 void set_listenerPos(unsigned char list_pos[]);
5 void perform_WFS(unsigned char noSrcs, char src[]);
6 void set_tapering_window(char window_Values[]);
7 void set_delayXMOSMixerVals( unsigned short num_values, char delay_Values[]);
8 void set_gainsdelaysXMOSMixerVals( unsigned short num_values, char gaindelay_Values[]);
9 void set_gainSHARCMixerVals( unsigned short num_values, char gain_Values[]);
10 void set_delaySHARCMixerVals( unsigned short num_values, char delay_Values[]);
11 void set_gainsdelaysSHARCMixerVals( unsigned short num_values, char gaindelay_Values[]);
12 void set_filterSHARCMixerVals( unsigned short num_values, char filter_Values[]);

```

Listing 31: Modifications to the *avb_1722_1_control* internal interface.

6.3.3 The WFS library and ImmerGo Modifications.

The WFS.js library is built to enable the server to synthesize source locations with WFS by calculating gain and delay values and relaying those to the mixer matrix within the endpoints. As shown in **Fig. 70**, the WFS library exposes functions and variables required to solve for the gains and delays. WFS.js exports 5 functions:

- *setListenerPos* sets the listening reference position within the WFS.js module.
- *getListenerPos* retrieves the listening position.
- *setSpeakerConfig* sends the speaker positions to the module that are used for WFS panning.
- *setTaperWidth* sets the tapering width of the tapering window calculation.
- *getTaperWidth* retrieves the tapering width of the function.
- *panWFS* calculates and returns the gains and delay values required by all the speakers for localizing a source with WFS.

The ImmerGo server has been modified to contain a listening reference position. The ImmerGo client was modified to hold the listening position retrieved from the server in a new variable, *ListeningPos*. This listening position is also sent to the endpoint using the *setListeningPos* function.

The filter coefficients are held within a new array, *filterCoeff*, which is set using sliders shown within the ImmerGo server menu. The coefficient values are sent to the endpoint using the *setFilterCoefficients* function.

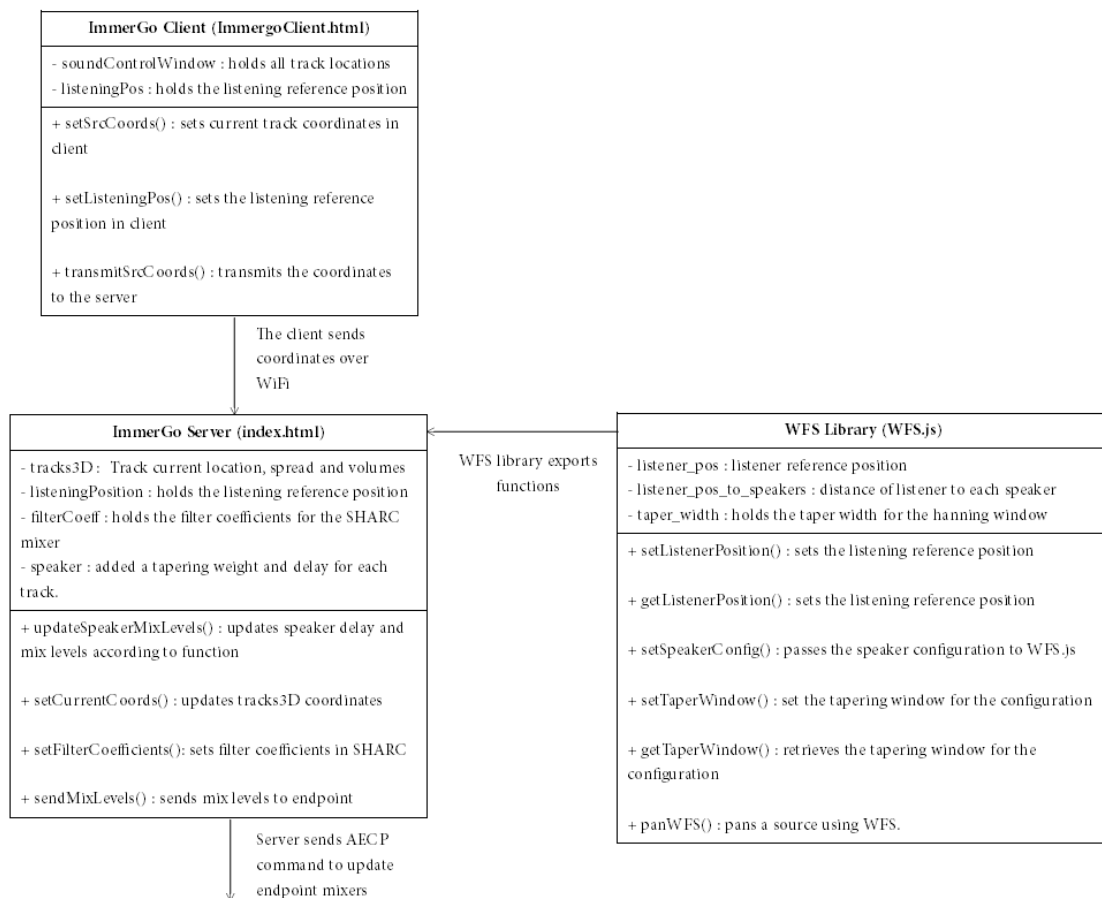


Fig. 70: The WFS.js module.

The ImmerGo loudspeaker model discussed in section 2.4.1 was updated to include two new variables pertaining to the WFS algorithm as highlighted in **Table 6** below. Each speaker has associated with it delay levels for each track being panned. The tapering weight variable is used to hold the tapering weight used for the output of the tapering window. When a loudspeaker is initialized, the delays are set to 0 and the tapering weights are set to 1.

Loudspeaker Model
- Position (x,y,z)
- Mix levels for each track
- Delay levels for each track
- Tapering Weight

Table 14: Loudspeaker model with modifications for WFS.

6.3.4 WFS Panning in ImmerGo

Before a source can be localized with WFS.js, a short pre-panning process is required to set both the listener reference position and speaker configuration. This process is shown in Fig. 71. After the speaker configuration has been loaded into the ImmerGo server memory, it is sent to the WFS.js module. This is then followed by setting the listener position within the module which defaults to 2m in front of the array.

As shown in Fig. 72, a client can only move a source within a limited area. This ensures that the source remains within a 3.6m area behind the speakers and to no go further forward than the listener reference position. When the server receives these coordinates, it makes a call to the WFS.js module with the track source position. The gain values for this position are then solved, followed by the delay values and returned to the server. They are then packetized and sent to the endpoints. These calculations are explained in depth in the following subsection. For more detail on how ImmerGo sends the gain and delay values, the reader is directed to Appendix C C1.

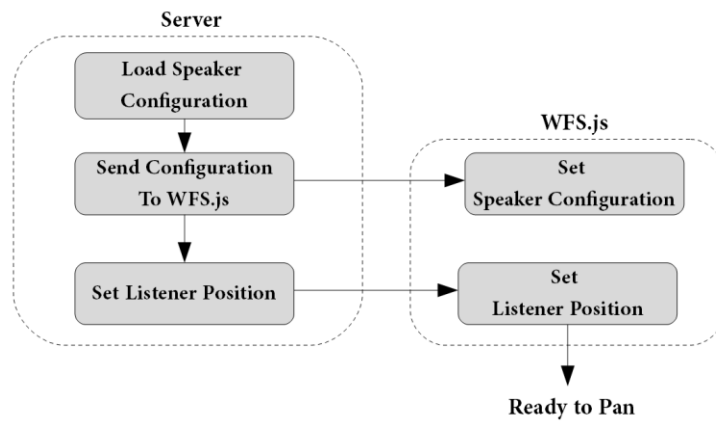


Fig. 71: Prepanning setup with WFS.js.

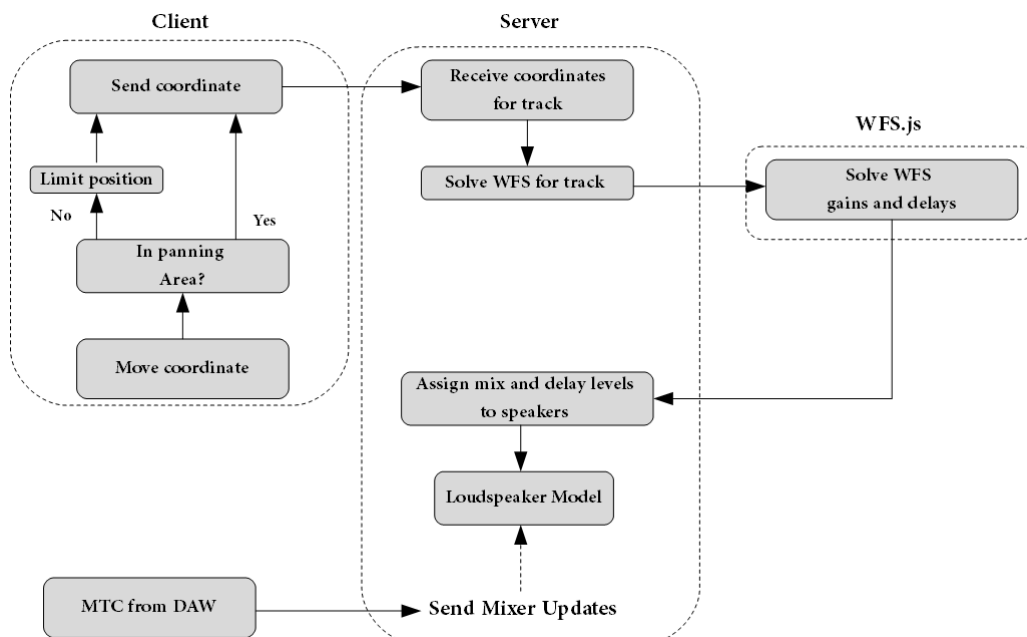


Fig. 72: Panning with the WFS library.

6.3.4.1 Calculating WFS for a Source

The WFS.js library solves for the speaker gains and delays according to the generalized WFS operator first discussed in section 6.1.2.4 and given in equation (6.12).

Fig. 73 shows the algorithm used to calculate the WFS gains and delays for a source coordinate, $s = (x, y, 0)$. For more detail on this calculation, the reader is referred to Appendix C C1. The y coordinate is checked if it is within 60cm of the linear array. If it is, a simple calculation determines whether the source and loudspeakers are further than the loudspeaker spacing, $\frac{\Delta x}{2}$, which ensures two loudspeakers will play the source at full volume while the others play nothing. 0 delay is applied to the source at this time.

If the source is further than 60cm from the array, the WFS calculations are performed as follows,

1. First the compensation ratio, $\zeta = \frac{y}{y_m}$ is solved.
2. This is used to solve for the compensation factor $\sqrt{\frac{\zeta}{\zeta-1}}$.
3. The distance of the source to the speaker, $|\mathbf{r}_n - \mathbf{r}_m|$, is then calculated.
4. This is used to determine the incidence gain, $\frac{y}{|\mathbf{r}_n - \mathbf{r}_m|} = \cos(\theta_n)$.
5. The square root of the inverse distance is then solved, $\frac{1}{\sqrt{|\mathbf{r}_n - \mathbf{r}_m|}}$.
6. These are then multiplied together to solve for the final gain weighting of a speaker.
7. The delay values are then calculated.

The delay calculation is output in milliseconds and needs to be converted into samples for a sample rate of 48kHz. The calculation for solving the delay in samples is shown in equation (6.15):

$$\tau(\mathbf{r}_n) = \text{sign}(s) \frac{|\mathbf{r}_n - \mathbf{r}_m|}{c} * 48 \quad (6.15)$$

Where $c = 34.3\text{cm/ms}$. This parameter is then offset to half the delay line size, 512 samples as shown in **Fig. 68** and **Fig. 69**.

After experimentation with this algorithm, it was found that the angle contribution component from equation (6.12) became 0 as the virtual source was moved closer to the loudspeaker array, this resulted in an unwanted dip in the virtual source volume as it crossed the array. To avoid this, when the virtual source reached a distance of 60cm from the array, the WFS calculations were disabled and the virtual source was simply played from the two closest loudspeakers.

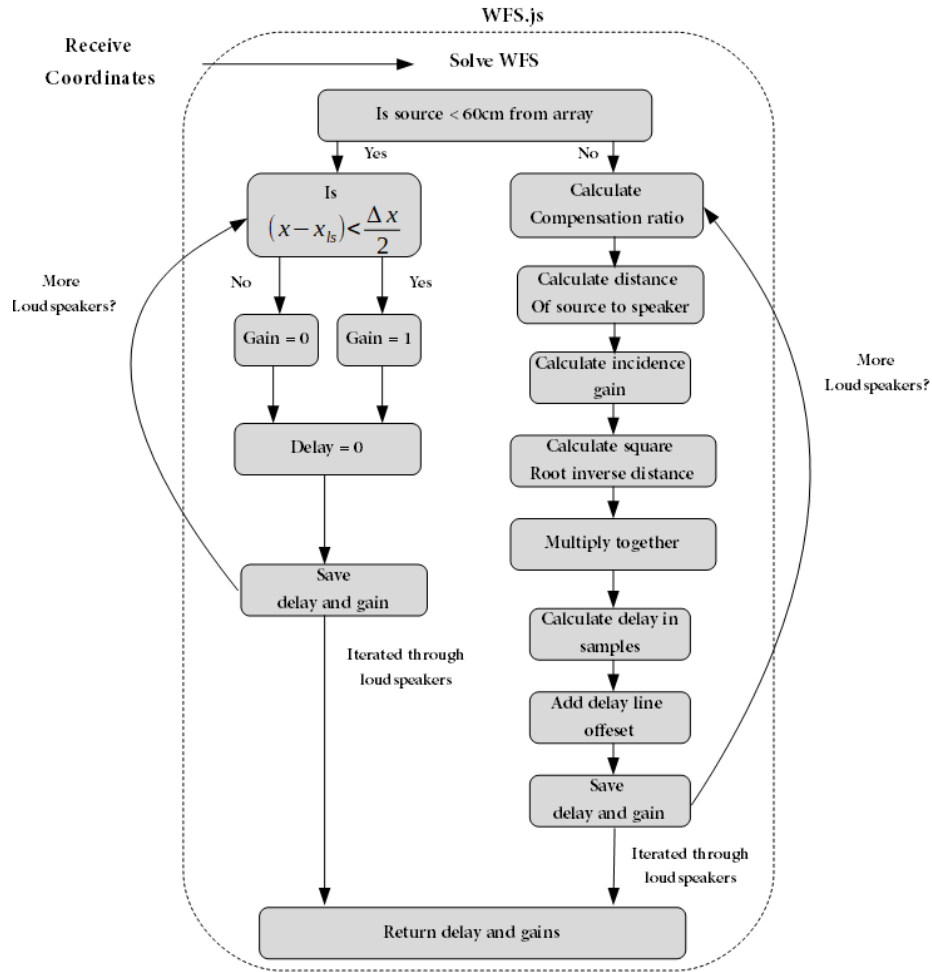


Fig. 73: Solving WFS gains and delays.

6.3.4.2 Calculating the Tapering Window

The tapering window is calculated according to a Hann window function²⁵ shown in equation (6.16). Assuming n is the ID of each N loudspeakers such that $0 < n < N$ then the taper weight $t(n)$ of that loudspeaker is:

$$t(n) = \frac{1}{2} \left[1 + \cos\left(\frac{\pi n}{a}\right) \right] \quad (6.16)$$

Where a is the width of the window such that a larger width increases the amount of attenuation of each speaker on either end of the array. This results in a tapering window that looks similar to those shown in Fig. 74 and Fig. 75 below.

²⁵ <http://mathworld.wolfram.com/HanningFunction.html>

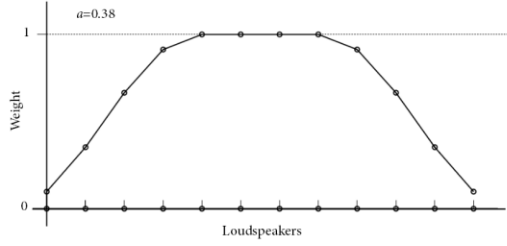


Fig. 74: Hann window with a width of 0.38.

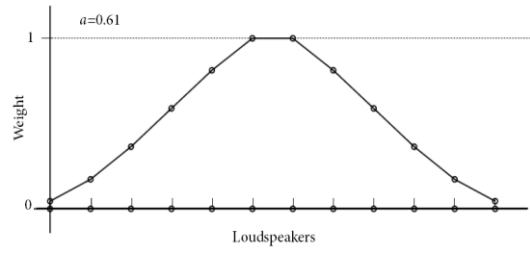


Fig. 75: Hann window with a width of 0.61.

The tapering width for the loudspeakers can be set using sliders found on the ImmerGo server main menu. As shown in Fig. 76, when the slider is set, the *setTaperWindow* function is called with a taper width value which creates a tapering window within the WFS.js module according to the number of speakers in the array. The server then makes a call to the *getTaperWindow* function which retrieves the tapering window from the WFS.js library. The loudspeaker model is updated with its new corresponding taper weight value. These are then sent to their corresponding endpoints by packing them into an AECP message with a descriptor value 0x0041. These are then unpacked within the SAR and used in all subsequent gain calculations such that the weight is combined within the gain calculation for each loudspeaker according to $w(\mathbf{r}_n) * t(n)$ where $t(n)$ is the tapering weight for the loudspeaker n within the array. This process is shown in greater detail in Appendix C C3.

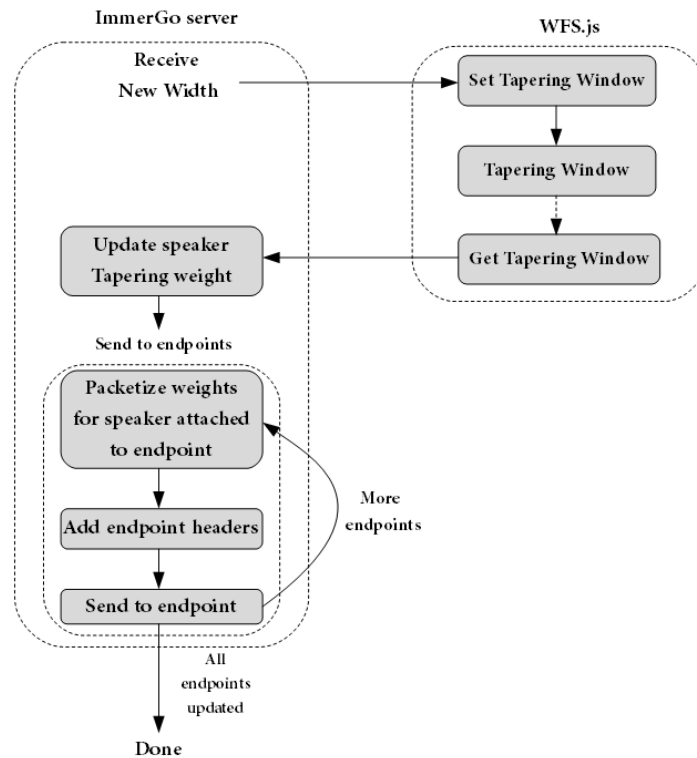


Fig. 76: Sending tapering window updates to endpoints.

6.3.4.3 WFS within the SAR

The SAR task was modified to incorporate the WFS algorithm. Before the SAR can perform WFS, it requires the listening reference position and the speaker configuration. This process is shown in Fig. 77. First, the speaker configuration, and IDs of the speakers attached to the endpoint, is set within the endpoints using the descriptor index 0x0010 like the DBAP implementation discussed in chapter 4. Once

the speaker configuration is set, the listening position coordinates are sent to each endpoint using the descriptor index 0x0011, and unpacked using the same process for track source coordinates that has been discussed in chapter 4. Once these are both received, the SAR is ready to begin panning using WFS.

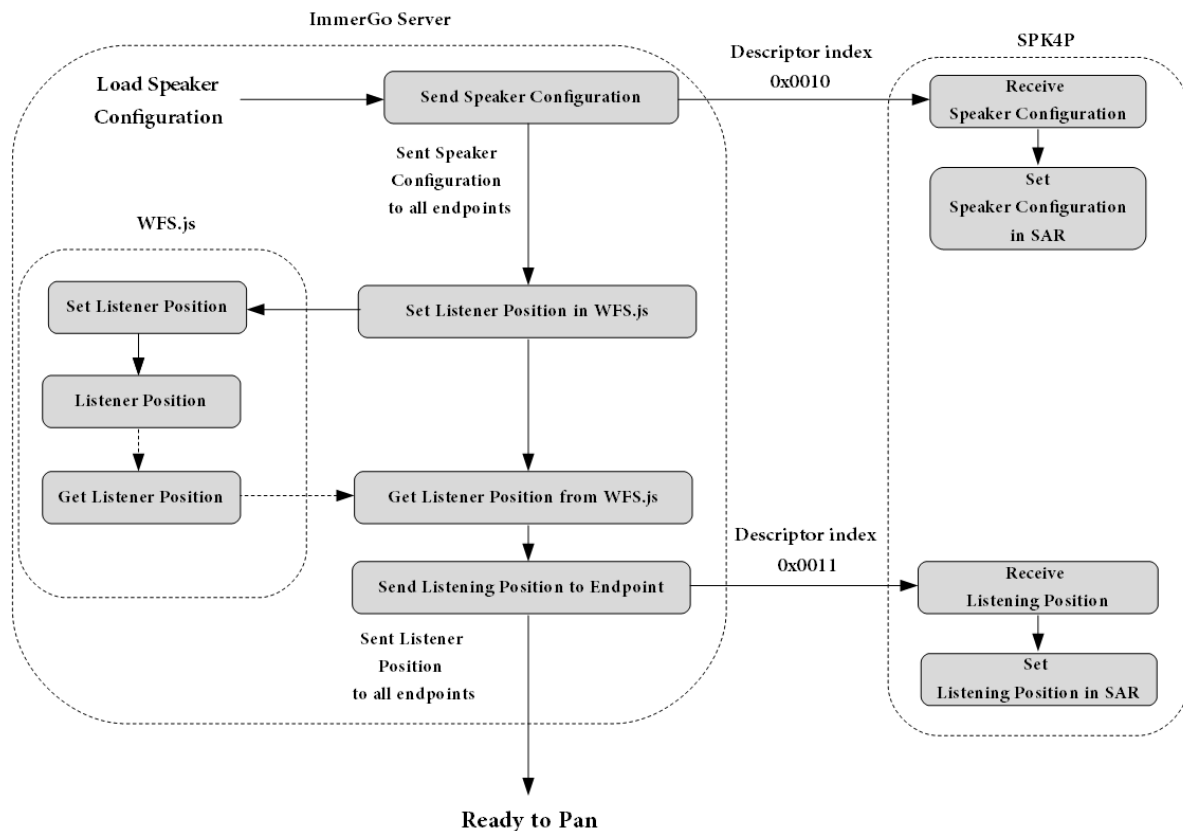


Fig. 77: Setting the listener position within the SAR.

When a source is moved by a user, or automation, the server receives the track ID and coordinates and then packs these into an AECP packet with the descriptor index 0x0040. This packet is then sent to each endpoint processor. As shown in Fig. 78, after the packet is received, it is routed to the SAR for the WFS calculation. The WFS calculations then are solved only for the speakers attached to the endpoints. The calculation is identical to the one used for each speaker when solving for WFS within WFS.js. The calculation outputs the gains and delays for the appropriate speaker which are saved to the delay and gains array within the SAR. After they are saved, a call to the *i_imr* mixer is made that sends pointers to the delay and gain values. These are then used to overwrite the *XMOSMixerLevels* and *XMOSDelayLevels* within the Audio Buffer Manager task as discussed in subsection 6.3.1 of this chapter.

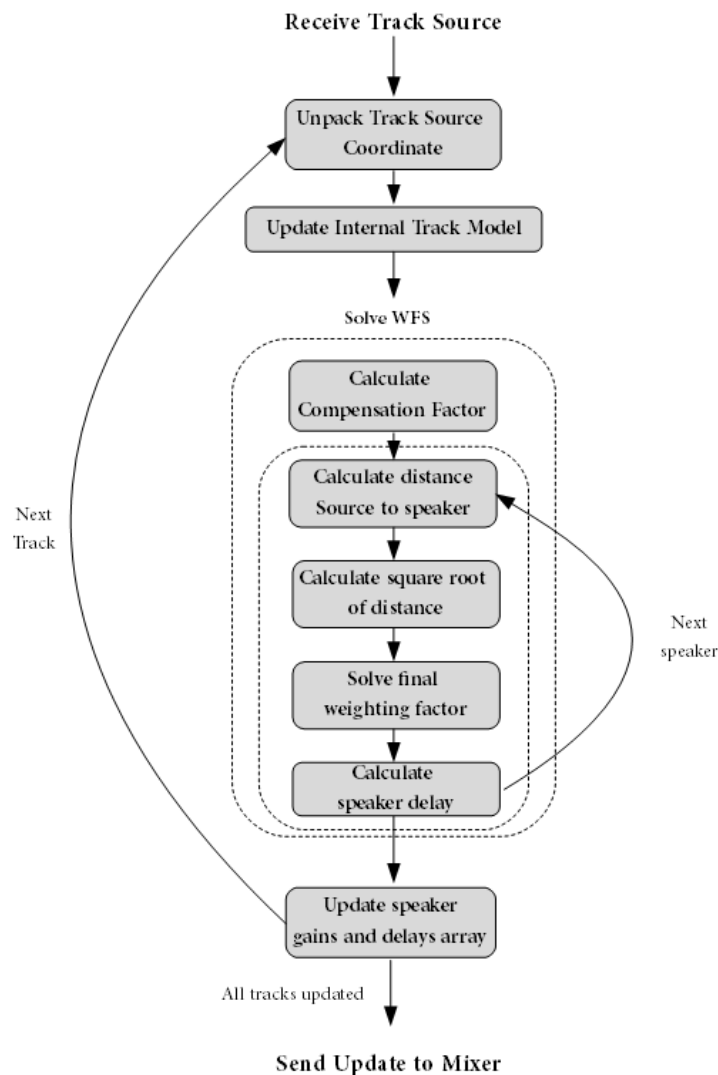


Fig. 78: Solving WFS within the SAR.

6.3.4.4 Mixing on the SHARC

The mixing process for the SHARC is shown in Fig. 79. The track sources are received, then for each track source, the WFS calculations are solved and the gains and delays array within the SAR is updated. As the array is updated, the new values are also sent to the SHARC mixer. Now, the gain values are converted to Q1.15 format and fill the first half of the 32 bit parameter. The delay values fill the second half. This is repeated for each speaker and the parameters are saved to the sending buffer along. The *DSP_AddrParaLoad* function as discussed in 4.3.1 is called with the parameter count, and the speakers corresponding to the track are updated. This is then repeated for any further tracks that were sent within the packet.

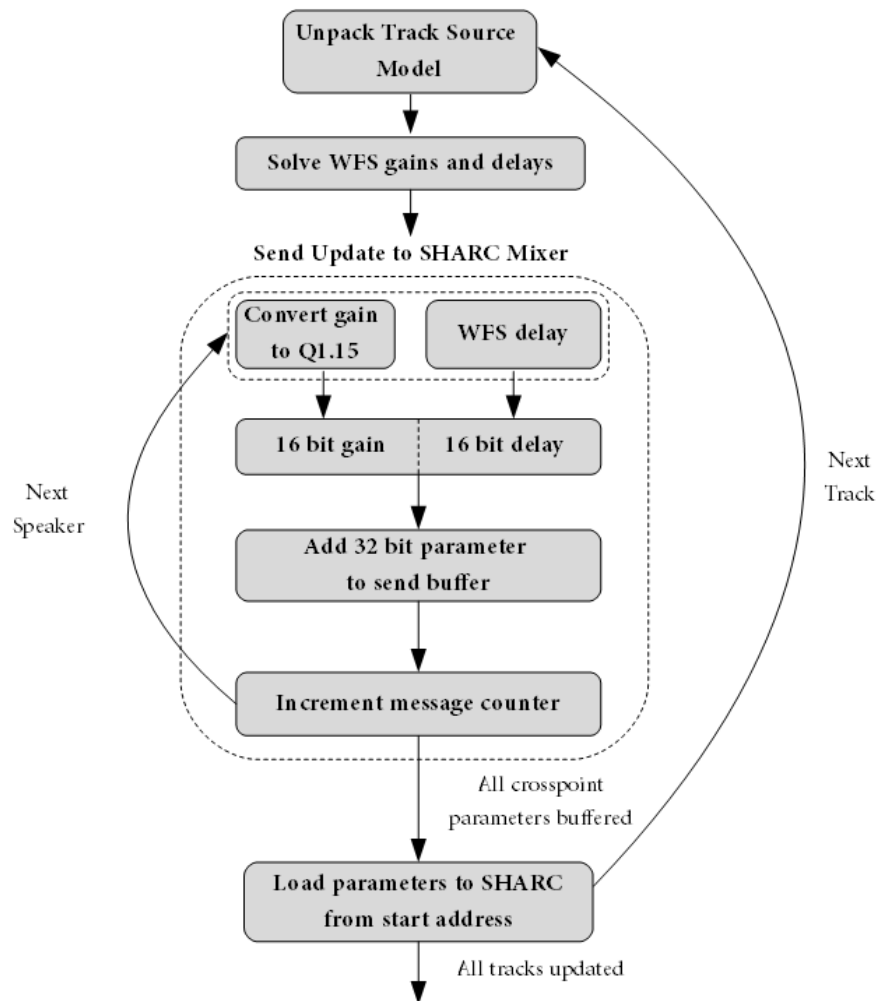


Fig. 79: Setting the SHARC mixer with WFS values.

6.3.4.4.1 Setting SHARC DSP Filters

The SHARC provides the attenuation and delay mixing capability with the inclusion of an amplitude smoothing filter and a one-pole low pass IIR filter. This IIR filter is used to prevent tearing of the audio as virtual sources are moved about the space and delays are updated often.

Incorporated within the ImmerGo server's interface are two sliders which are used to set the low pass filter and an amplitude smoothing filter coefficients. Once set, both the values are encapsulated within an AECp packet with the descriptor index 0x0007 and sent to each endpoint to update the mixer matrix.

The IIR and amplitude smoothing filter coefficients are addressed at the very end of the matrix crosspoint addresses. This means for the SPK-4P, they are addressed at 64 and 65. The SAR is able to send updates of the filter coefficients using the *load_filter2SHARC* shown in **Listing 32**. Here, a pointer to the float array holding both the coefficient parameters is given to the *DSP_AddrParaLoad* function with the starting address of the filters 64. The *DSP_AddrParaLoad* function is made with a *msg_count* of 2 which tells the SHARC to read in both functions.

```

418 void load_filter2SHARC(float* filter_coeff){
419     //Load filter coefficients as float into SHARC
420     unsigned int DSP_Address;
421     unsigned int msg_count;
422
423     DSP_Address = MAX_CHANNELS*MAX_SPEAKERS;           //start from MAX_CHANNELS*MAX_SPEAKERS
424
425     msg_count = 2;
426     DSP_AddrParaLoad(DSP_Address, (unsigned int*)filter_coeff, msg_count);
427 }

```

Listing 32: Function to load filter coefficients.

6.4 Summary

This chapter discussed the theory and implementation of the third object-based immersive audio rendering algorithm implemented within the ImmerGo system, WFS. There is a brief description of the Kirchhoff-Helmholtz integral and the Green's function which describe the general mathematical description of WFS. As described in this these, the Kirchhoff-Helmholtz integral needs to be approximated into monopole discrete sources, which are in reality loudspeakers. The Rayleigh I integral describes the approximated Kirchhoff-Helmholtz integral. Using the stationary-phase approximation, the Rayleigh I integral is further approximated into a discrete integral. This is used to determine the function that should be applied to the virtual source signal at each loudspeaker to synthesize its location. There is a discussion of types of sources, and the functions required to synthesize them.

This chapter covered an overview of how WFS was incorporated within ImmerGo. This was done similarly to VBAP, and a WFS.js library was created. ImmerGo is able to feed its track object locations into a function within the library which determine the attenuation and delay amounts required to synthesize focused and non-focused sources. In order to do this, the library also held variables for the listening reference position and, to prevent the truncation effect, the tapering window. The loudspeaker definition models were modified to contain variables that held the delay amounts required for each source being panned, and the tapering weight. In order to have the XMOS mixer matrix perform delay processing required by WFS, a delay-line has been implemented within the mixer matrix. This chapter discusses how this was done, with diagrams showing how the delay buffers are used for both the focused and non-focused sources. The delay-summation mixer matrix was limited to 8 tracks for each speaker, due to memory constraints within the XMOS processor.

For step 2 and 3 of distribution, the SAR was extended to incorporate WFS calculations that could be solved according to track objects received from the ImmerGo server. This chapter discusses how these two levels of distribution have been achieved. Similar to the previous algorithms, two versions of firmware were made, one that performed mixing in the XMOS and another that performed mixing in the SHARC. The SHARC DSP provided more capabilities than the XMOS mixer matrix since it was able to perform delay and attenuation processing on 32 tracks to each speaker, as well as a one-pole, low pass filter and amplitude smoothing that could be applied after the mixing summation.

Chapter 7

Ambisonics

All the rendering algorithms that have been implemented thus far require that the renderer solves for new mixing values according to updates of track object metadata. Without this metadata, the mixer updates for the rendering of object positions are unable to be calculated. Ambisonics is different from this because it uses an encoding process to encode the directions, and changes in directions, of track objects into a finite set of audio channels. To have accurate playback on a speaker system, the renderer then decodes these encoded audio channels to each speaker. The decoding process requires that the encoded audio channels are mixed according to values that are set once, and only once, provided the speaker layout does not change.

This puts Ambisonics into the category of scene-based audio (SBA) as it incorporates aspects of both object and channel-based audio. Objects are still used to define the location of audio tracks, however, these are encoded into a finite set of audio channels. The decoding process is essentially a mixing process that takes into account the direction of the speakers used for playback. These directions determine how the encoded audio channels are to be weighted when mixing them to each speaker. Since the speakers don't move, the mixing values output from the decoder are only solved once, which is very different from the real-time mixer updates required by the object-based techniques.

This chapter continues with an overview of how the Ambisonic encoding and decoding process is performed. This is followed by an implementation of Ambisonic decoding within the SAR.

7.1 Ambisonics Theory

Ambisonics was first introduced as a "with-height" audio recording technique in Gerzon's paper (Gerzon, 1973). It was presented as a method to both capture and reproduce a soundfield, also known as a sound scene, in vertical and horizontal directions. The idea is to use multiple mic capsules to capture sounds from all directions around a listener position which may then be transformed and played back on speakers. Since then, the study of Ambisonics has led to the development of microphone technology capable of unprecedented accuracy and realism (the "Soundfield" microphone), and a range of sound manipulation techniques for both individual audio tracks and complex sound scenes (Furness, 1990). The encoding and decoding process allows for the encoded format to be decoded on many different loudspeaker layouts, provided they meet basic requirements.

Ambisonics defines the encoding of every direction of a sound into one multichannel file format. This means that the Ambisonic-encoded signals do not feed any loudspeakers directly but rather carry an n -order representation of the entire soundfield (Furness, 1990) (Hollerweger, 2013) that is decoded to be heard on a loudspeaker configuration. This is analogous to the mid/side recording technique first introduced in Blumlein's 394 325 UK patent (U.K. Patent No. U.K. Patent 394 325, 1931).

Mid/side is best explained as a recording technique where a figure-of-eight microphone is used to capture left and right (horizontal) "side" information and one cardioid (or hyper-cardioid) microphone is used to capture "mid" information. The side microphone is positioned perpendicularly off-axis to the subject being recorded so as to pick up "side" information, whilst the mid cardioid microphone is positioned in-between the figure-of-eight pattern so as to capture "mid" information. This is best illustrated in Fig. 80.

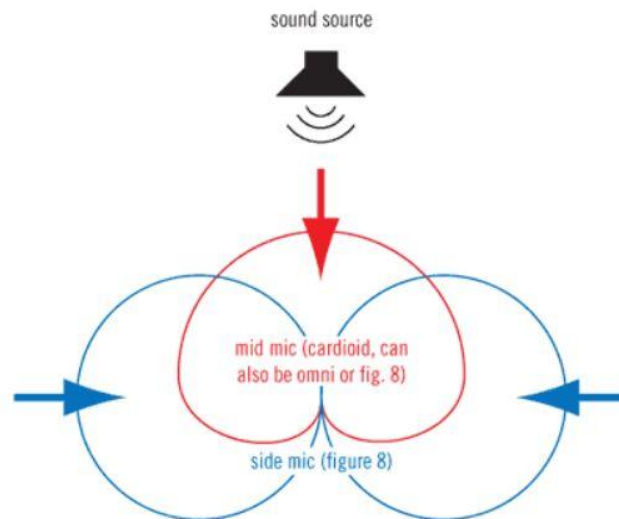


Fig. 80: Mid/Side recording technique.

This technique captures a horizontal sound scene which can then be reproduced by panning the left microphone capsule's information to the left, panning the right microphone's capsule to the right and inverting the phase. The mid capsule is generally a cardioid pattern which receives the overall volume of what is in front of the recording position. The side signal given by the figure-of-eight mic is then used to determine stereo directionality of the scene. Certain parameters like the spread of the scene can then be controlled by adjusting their level of left to right pan.

7.1.1 Ambisonics Overview

As stated in (Hollerweger, 2013), the physical formulation behind Ambisonics has been based on two assumptions.

1. A soundfield can be regarded as a superposition of plane waves.
2. A plane wave can be represented according to an infinite series of spherical harmonic functions.

The second condition holds true if a listener is situated at a point of origin, or "sweet spot" that the Ambisonic soundfield surrounds.

7.1.1.1 The Spherical Harmonic Functions

Ambisonics encodes the directionality of audio objects into components according to the spherical harmonic functions (Atkinson & Han, 2012). The spherical harmonics may be seen as a superset of the polar patterns shown in the mid/side technique. The reader is directed to the first section of (Monro, 2013)

should they want a clear explanation of how these spherical harmonics are formed. In essence, each component forms an orthogonal system that approximates the surface of a unit sphere, which can be used to capture information around the sphere (Atkinson & Han, 2012). The spherical harmonics are solved formally according to the Legendre polynomial and the Legendre polynomials are what derive the Ambisonic encoder components. A simple version of these functions is given in (Malham, 2003).

As shown in **Fig. 81** below, each spherical harmonic function has an area of capture, and polarity. These are seen as a hierarchy, where each function has an associated letter, and polarity, with red as positive, and blue as negative. In practice, the functions associated with the spherical harmonics are solved for particular azimuth and elevation directions, and then a single mono signal is copied into a new audio channel and multiplied by the appropriate output spherical harmonic function to generate a multichannel format holding the entire encoded Ambisonic soundfield.

As the order of spherical harmonics increase, so do the number of polar patterns. This means that the higher the order, the more resolution can be contained in Ambisonics soundfield. Starting at the zero-th order for an omnidirectional soundfield with no directional characteristic, the first order includes soundfield directions in the left, right, forward, backward, vertical directions and as well as the omnidirectional zero-th order, and this continues upwards through the spherical harmonics.

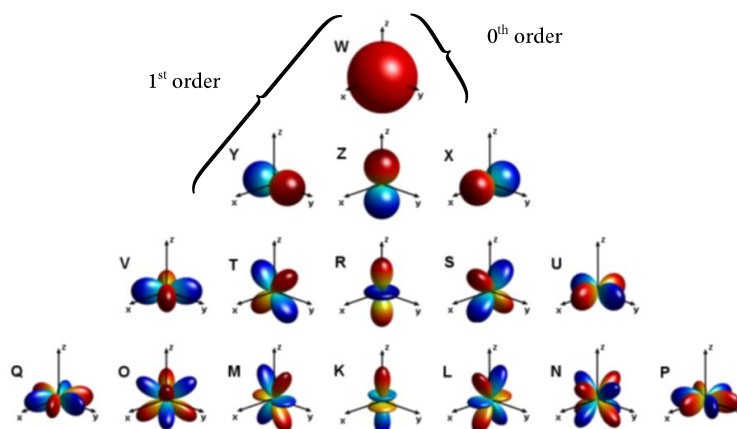


Fig. 81: Hierarchy of Spherical Harmonics and associated B-format ²⁶.

7.1.1.2 Ambisonics Playback

In order to have speakers play this back, the components need to be summed at each speaker in a manner that has the entire signal pressure and directional velocities proportional to the encoded components, not unlike the decoding of the mid/side technique. This process is known as Ambisonic decoding and is sometimes considered a “re-encoding” because the same functions used to encode the sound scene are used to decode it. The Ambisonic-encoded signal is able to be decoded to any loudspeaker array (Gerzon, 1973) but it is imperative that the decoder is made exclusively for the particular loudspeaker layout being decoded to. There are two important restrictions needed to decode an Ambisonic-encoded signal.

²⁶ <http://pcfarina.eng.unipr.it/SPS-conversion.htm>

1. The number of loudspeakers L must be more than or equal to the number of encoded components or Ambisonic channels N such that $L \geq N$ (Hollerweger, 2013).
2. The loudspeakers must have a regular layout that is based on a platonic solid as shown in Fig. 82 (Gerzon, 1973) (Hollerweger, 2013).

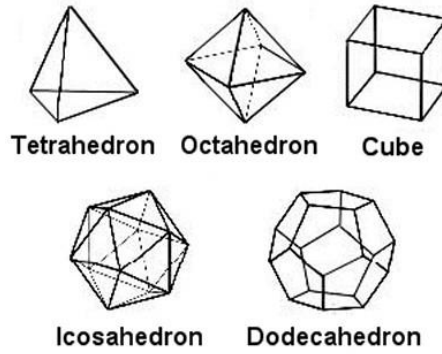


Fig. 82: Assuming the loudspeakers are at each vertex, any of these layouts could be decoded to.²⁷

For a N -order Ambisonic encoding, the number of Ambisonic components M for a three-dimensional loudspeaker setup is given by:

$$M = (N + 1)^2 \quad (7.1)$$

These loudspeaker layouts allow for decoder functions to be directly mapped without the need for further manipulation. For irregular loudspeaker setups, there are techniques which allow for an Ambisonic signal to be decoded. These involve the introduction of interaural time delay techniques, a combination of VBAP with Ambisonics to generate phantom vertices, and even a new formulation of soundfield components in terms of Fourier-Bessel coefficients (Bruno, Laborie, & Montoya, 2005) (Wiggins, 2007) (Trevino, Okamoto, Iwaya, & Suzuki, 2010) (Zotter & Frank, 2012). These techniques go beyond the scope of this current research and will be omitted.

For the purposes of this research, this thesis will not go above 4th-order Ambisonics because the channel counts that carry the soundfield become too high for the channel counts that can currently be received by the XMOS processor. There are, however, plugin-based approaches which have implemented up to 7th-order Ambisonics^{28 29}.

7.1.2 Encoding an Ambisonic Soundfield

For simplicity, this section will discuss how to encode an Ambisonic soundfield according to the simplest formulation of the Ambisonic functions. The method used for decoding in this research will then be given.

²⁷ http://www.maths.gla.ac.uk/~ajb/3H-WP/platonic_solids.gif

²⁸ http://research.spa.aalto.fi/projects/sparta_vsts/

²⁹ <https://plugins.iem.at/>

There are various Ambisonic formats that are currently used in both research and commercial applications and these will be discussed to inform the reader of the trends in current Ambisonic applications.

7.1.2.1 First Order

Ambisonic first order encoding components, also known as B-format, are shown below. Equation (7.2) shows how a mono signal would have its direction encoded using Ambisonics. Assuming we are encoding N mono signals for sample number t we have:

$$\begin{aligned} W(t) &= \frac{1}{N} \sum_{i=1}^N s_i(t) \left[\frac{1}{\sqrt{2}} \right] & X(t) &= \frac{1}{N} \sum_{i=1}^N s_i(t) [\cos \theta_i \cos \varphi_i] \\ Y(t) &= \frac{1}{N} \sum_{i=1}^N s_i(t) [\sin \theta_i \cos \varphi_i] & Z(t) &= \frac{1}{N} \sum_{i=1}^N s_i(t) [\sin \varphi_i] \end{aligned} \quad (7.2)$$

where s_i are mono signals that have a direction θ_i (horizontal azimuth) and φ_i (vertical elevation).

The equations themselves correspond to the first four 3D polar patterns described in Fig. 81. The W component is omnidirectional, the X and Y are horizontal planar and Z is the vertical axis. These equations are known as the SN3D ordering of the spherical harmonic equations, other orderings will be elaborated later in this chapter.

A 3D Ambisonic soundfield can be encoded using a minimum of 4 components that are based on the functions given in (7.2). These components are in fact 4 audio channels that have a weighted copy of the mono signal according to the spherical harmonics in the intended direction (θ, φ) (Hollerweger, 2013). The input and output of an Ambisonic encoded signal can be seen in Fig. 83 below,

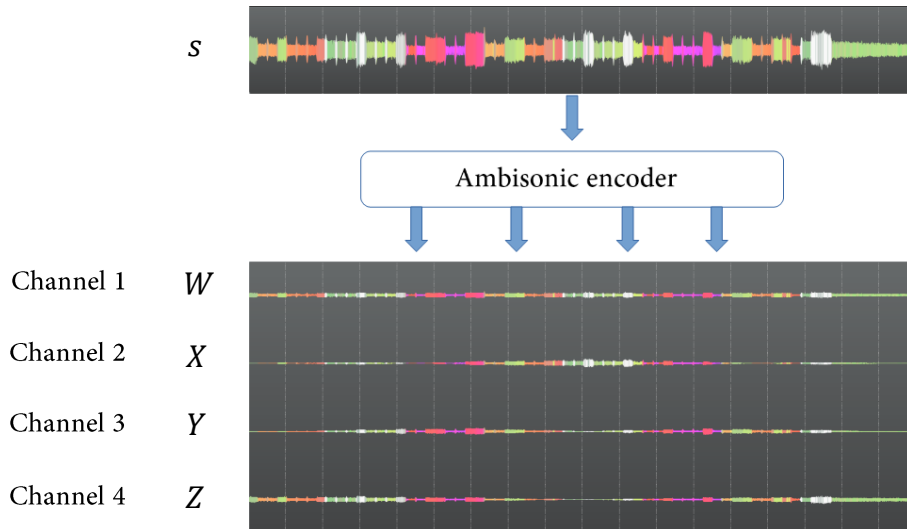


Fig. 83: Encoding process of a mono signal to first order Ambisonics.

7.1.2.2 Higher Orders

In order to increase the resolution, this encoding method can be extended to higher orders of the spherical harmonics by weighting the mono channel using the appropriate spherical harmonic function and thereby creating additional encoded channels. **Table 15** below shows functions which are used for the SN3D (Semi-Normalized 3 dimensional) spherical harmonic functions and FuMa channel ordering (Malham, 2003). As described in (Malham, 2003) this representation is used in combination with a weighting factor, known as the FuMa (Furse Malham) weight, which ensures equal power or volume intensity as more channel components are introduced. These functions are what represent 3D polar patterns as shown in **Fig. 81**.

Order	Channel	SN3D definition	FuMa Weight
0	W	1	$1/\sqrt{2}$
1	X	$\cos \theta \cos \varphi$	1
	Y	$\sin \theta \cos \varphi$	1
	Z	$\sin \varphi$	1
2	R	$(3\sin^2 \varphi - 1)/2$	1
	S	$\sqrt{3}/2 \cos \theta \sin(2\varphi)$	$2/\sqrt{3}$
	T	$\sqrt{3}/2 \sin \theta \sin(2\varphi)$	$2/\sqrt{3}$
	U	$\sqrt{3}/2 \cos(2\theta) \cos^2 \varphi$	$2/\sqrt{3}$
	V	$\sqrt{3}/2 \sin(2\theta) \cos^2 \varphi$	$2/\sqrt{3}$
3	K	$\sin \varphi (5 \sin^2 \varphi - 3)/2$	1
	L	$\sqrt{3/8} \cos \theta \cos \varphi (5 \sin^2 \varphi - 1)$	$\sqrt{45/3}$
	M	$\sqrt{3/8} \sin \theta \cos \varphi (5 \sin^2 \varphi - 1)$	$\sqrt{45/3}$
	N	$\sqrt{15}/2 \cos(2\theta) \sin \varphi \cos^2 \varphi$	$3/\sqrt{5}$
	O	$\sqrt{15}/2 \sin(2\theta) \sin \varphi \cos^2 \varphi$	$3/\sqrt{5}$
	P	$\sqrt{5/8} \cos(3\theta) \cos^3 \varphi$	$\sqrt{8/5}$
	Q	$\sqrt{5/8} \sin(3\theta) \cos^3 \varphi$	$\sqrt{8/5}$

Table 15: SN3D Ambisonic encoding functions up to third order with FuMa weightings.

7.1.3 Decoding an Ambisonic Soundfield

In order to reproduce an Ambisonic soundfield, it needs to go through a decoding process for loudspeakers placed around a listener. There are a variety of different decoding methods that have been developed since Gerzon's paper. These methods can be considered either physical or psychoacoustic (Arteaga, 2018). There are also extensions to these methods that allow for the reproduction of the soundfield on irregular loudspeaker arrays (Zotter & Frank, 2012). The research in this thesis will cover the basic 3D decoding method known as mode-matching.

Mode-matching decoding relies on the assumption that all speakers are positioned on the vertices of a regular platonic shape and that each speaker is equidistant from the listener. The decoder intends to

reproduce the pressure of the W component and the velocity of the directional components proportional to the encoded signal (Artega, 2018) so there are no distortions. As shown in **Fig. 84**, each speaker has its own unique decoder, which is determined according to its direction. This decoder is fed each component of the Ambisonic signal. It weights and sums the components, creating a signal which is fed to the loudspeaker.

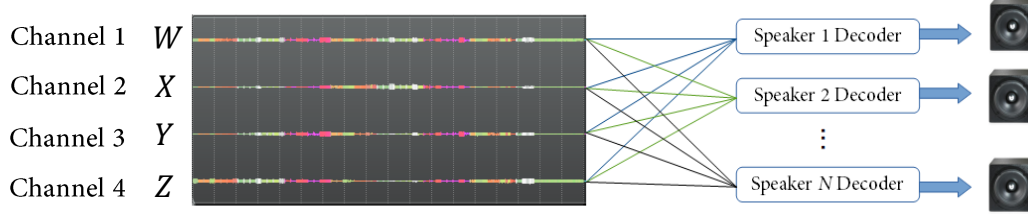


Fig. 84: Decoding a B-format signal to multiple speakers.

To help explain this process more clearly, an example of a mode-matching first order decoding for SN3D soundfield with FuMa weights is discussed below. Given a set of N loudspeakers that are equidistant from a listener at the origin, the decoder signals for the j th loudspeaker is given as:

$$s_j(t) = w_j W(t) + x_j X(t) + y_j Y(t) + z_j Z(t) \quad (7.3)$$

where the coefficients w_j , x_j , y_j and z_j are dependent on the loudspeaker direction and $W(t)$, $X(t)$, $Y(t)$ and $Z(t)$ are the Ambisonic encoded signal values at sample t .

Assuming the j th loudspeaker has a direction of (θ_j, φ_j) the decoder's weighting function for each component prior to summation will be:

$$\begin{aligned} w_j &= \sqrt{2} \\ x_j &= \cos \theta_j \cos \varphi_j \\ y_j &= \sin \theta_j \cos \varphi_j \\ z_j &= \sin \varphi_j \end{aligned} \quad (7.4)$$

Referring back to **Fig. 84**, each speaker's decoder will sum the encoded signal according to (7.3) as shown in **Fig. 85**.

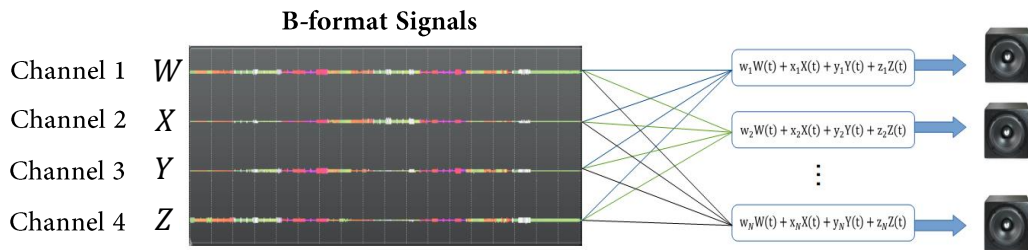


Fig. 85: Decoder functions within each speaker's Ambisonic decoder.

The Ambisonic decoder reproduces the pressure of the W component and the velocity of the directional components proportional to the encoded signal (Artega, 2018). This means that the B-format components, $\mathbf{B}$, may be written as a set of linear equations that is produced by each speaker's signal $s_j(t)$ according to the speaker directions (θ_j, φ_j) . This is shown below in (7.5) where $\mathbf{C}$ is an $M \times N$ matrix with M rows as

the weighting component functions and with N columns for each loudspeaker. Then $\mathbf{p}$ is a matrix for the loudspeaker signals.

$$\mathbf{B} = \begin{bmatrix} W(t) \\ X(t) \\ Y(t) \\ Z(t) \end{bmatrix} = \begin{bmatrix} \sqrt{2} & \sqrt{2} & \dots & \sqrt{2} \\ \cos \theta_1 \cos \varphi_1 & \cos \theta_2 \cos \varphi_2 & \dots & \cos \theta_N \cos \varphi_N \\ \sin \theta_1 \cos \varphi_1 & \sin \theta_2 \cos \varphi_2 & \dots & \sin \theta_N \cos \varphi_N \\ \sin \varphi_1 & \sin \varphi_2 & \dots & \sin \varphi_N \end{bmatrix} \begin{bmatrix} s_1(t) \\ s_2(t) \\ \dots \\ s_N(t) \end{bmatrix} = \mathbf{C} * \mathbf{p} \quad (7.5)$$

Now if $\mathbf{C}$ is a square matrix (i.e $N = 4$) this equation can be solved using the inverse of $\mathbf{C}$ according to,

$$\mathbf{p} = \mathbf{C}^{-1} * \mathbf{B} \quad (7.6)$$

Generally, more than 4 loudspeakers are used in practical applications, so this equation must be solved using the pseudoinverse method, where the pseudo-inverse of $\mathbf{C}$ is $\mathbf{D}_{pinv}$ and $\mathbf{C}^T$ is the transpose of $\mathbf{C}$.

$$\mathbf{D}_{pinv} = \mathbf{C}^T (\mathbf{C} \mathbf{C}^T)^{-1} \quad (7.7)$$

This $\mathbf{D}_{pinv}$ matrix may then be used in place of $\mathbf{C}^{-1}$.

$$\mathbf{p} = \mathbf{D}_{pinv} \begin{bmatrix} W(t) \\ X(t) \\ Y(t) \\ Z(t) \end{bmatrix} = \mathbf{D}_{pinv} * \mathbf{B} \quad (7.8)$$

$\mathbf{D}_{pinv}$ is often called the decoder or ‘re-encoder’ matrix for the loudspeaker configuration (Hollerweger, 2013) and its values are used to mix the Ambisonic encoded channels. As has been mentioned, Ambisonics imposes particular loudspeaker layouts that can be used with this method. A common layout for first order Ambisonics is a cube or an octohedron, but there are also others³⁰.

7.1.4 Ambisonic Formats

The format used to encode the Ambisonic soundfield’s components has changed over the years since it was first introduced by Gerzon. This has made interoperability between various Ambisonics systems interesting and a variety of tools have been built to both encode and convert between different formats. The popular Ambisonics formats that are used in different Ambisonics systems are FuMa format (SN3D with FuMa weights), the AmbiX format and the Two Big Ears (TBE) format for facebook360’s Spatial Audio Workstation (Nachbar, Zotter, Deleflie, & Sontacchi, 2011) (Malham, 2003) (Farina, Ambisonics to TBE conversion, 2017). Each of these formats is based on spherical harmonics, but each has their own channel orderings and encoder function weightings. This is important to know when decoding the encoded soundfield, otherwise the decoder would render the directions and volumes of the audio objects incorrectly.

³⁰ Ambisonic decoder database - <https://www.york.ac.uk/sadie-project/ambidec.html>

7.1.4.1 FuMa format

The FuMa method has been used in the examples above describing the encoding and decoding process. The encoder outputs channel orderings are given by letters in the English alphabet and ordered into channel numbers as shown in **Fig. 86** below. Each number relates to the audio channel within a multichannel wave.

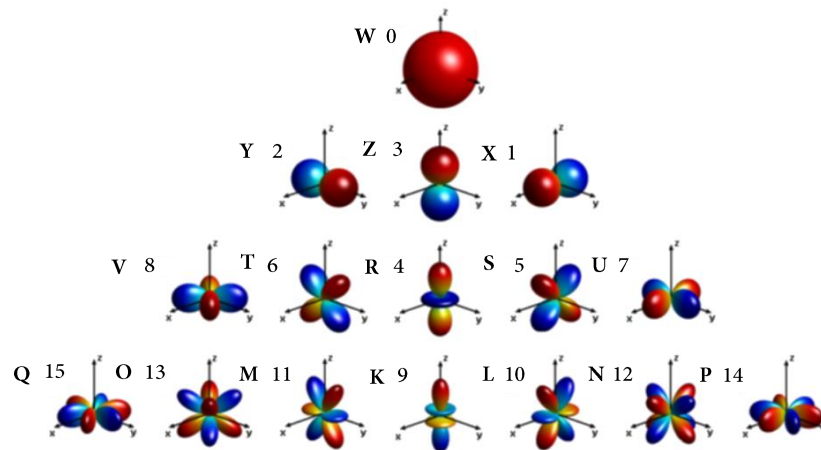


Fig. 86: FuMa channel ordering ³¹.

7.1.4.2 AmbiX format

A recognizable problem with the FuMa format is the implicit channel limit and since there are only 26 letters in the English language, a maximum of 26 components may be encoded. This causes an issue when encoding Ambisonics into orders greater than 4th order. A solution to this problem is given in (Nachbar, Zotter, Deleflie, & Sontacchi, 2011) and shown in **Fig. 87**. Here, a new Ambisonics format was introduced with Ambisonic components represented as numbers instead of letters known as the AmbiX format. This format uses the Ambisonic Channel Number (ACN) channel ordering which solves the problem of the FuMa letter ordering. Typically, AmbiX encodes using the SN3D normalization, however N3D (3D Normalized) is also sometimes used ³².

As indicated in (Nachbar, Zotter, Deleflie, & Sontacchi, 2011), it would seem AmbiX will become the most common standard for Ambisonic formats. A table of encoder functions up to 5th order can be found in (Farina, HOA Explicit Formulas, 2017) and are the formulas that have been used in this project.

³¹ <http://pcfarina.eng.unipr.it/SPS-conversion.htm>

³² https://ambisonics.iem.at/xchange/fileformat/ambi_x_format_appendix.pdf/view

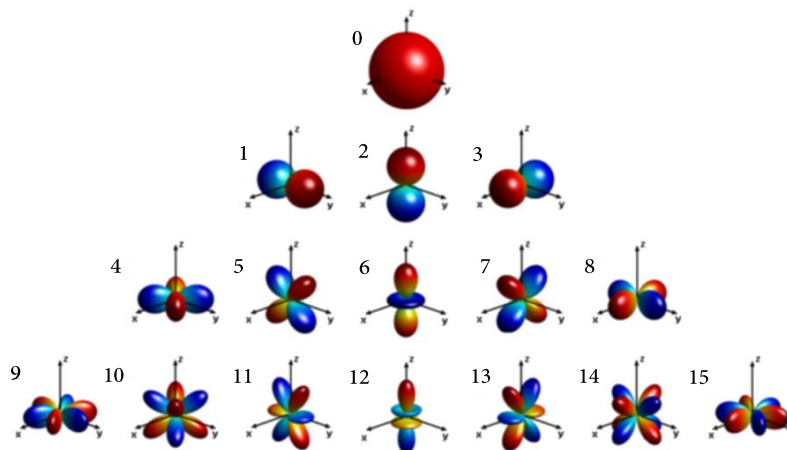


Fig. 87: ACN channel ordering³³.

7.1.4.3 Two Big Ears format

The Two Big Ears (TBE) Ambisonic format is used in the facebook360 Spatial Audio Workstation and is a non-standard format design for 360 video and audio distribution over the web. The standard employs a 2nd order Ambisonic format with one of the channels removed which leaves 8 channels. As well as this, it also has a polarity shift that complies with its own unique azimuth system. The formulas used for encoding the TBE format are found in (Farina, Ambisonics to TBE conversion, 2017) which have specific weights that are used. It has also been shown that an encoded TBE format is able to be transcoded into AmbiX format, which can then be decoded like a normal AmbiX format.

7.2 Ambisonics in ImmerGo

Ambisonics decoding is different from the object-based algorithms as it does not need to calculate and change the mixer matrix values in real-time. Instead, Ambisonics decoding requires each speaker's mixer matrix to be set once according to its location. As such, a different approach is taken when implementing the algorithm, since the decoders may be solved and set once prior to streaming audio. The host is used to calculate the spherical harmonic functions because it is less complicated to implement within the NWJS app than in the XMOS processor. The output from the calculations are then used to update the endpoint mixer matrices accordingly. The mixer matrix can then receive an Ambisonics encoded signal and decode it according to the decoder weights.

Ambisonic decoders for FuMa and AmbiX using SN3D normalization have been implemented. They are able to decode up to 4th order Ambisonics. The limitation being that the XMOS processor may currently only receive one stream with 32 channels. This can be extended provided the endpoints are able to pick up two streams of 32 audio channels, which would then allow for 7th order Ambisonics, and even higher with further AVB streams. Since there are already multiple plugins that perform Ambisonics encoding, ImmerGo was modified to send MIDI messages to the DAW containing the azimuth and elevation of a track source which is used to control the encoder plugin on a track.

³³ <http://pcfarina.eng.unipr.it/SPS-conversion.htm>

The following subsections will discuss the incorporation of an Ambisonic library that generates a decoder for each speaker. It is then shown how the endpoint mixer matrices for each speaker were set according to the values within the decoders.

7.2.1 Ambisonics Library and ImmerGo Modifications

The Ambisonics library exports one function to the ImmerGo server. This function is called the *getAmbisonicsDecoder*. It receives the speaker configuration and then determines an Ambisonic decoder for each speaker according to its direction. The loudspeaker definition model within the ImmerGo server was modified to have an Ambisonic decoder array that holds the summation weights for the speaker's decoder as shown in **Table 6** below.

Loudspeaker Model	
-	Position (x,y,z)
-	Mix levels for each track
-	Ambisonic decoder

Table 16: Ambisonic decoder in the loudspeaker model.

The *getAmbisonicsDecoder* uses an NPM library to help compute the Ambisonics decoders. The library is called the spherical transform library³⁴ and computes the decoders according to the pseudo-inverse matrix method as described earlier in equation (7.7).

Fig. 88 shows how the library is used within the server and described in further detail in Appendix D D1. After the speaker configuration has been loaded, the *getAmbisonicsdecoder* function is called with the speaker configuration and Ambisonics format that is to be decoded to. Then,

1. The total number of speakers within the configuration is used to determine the order of the Ambisonics decoder that should be made for the speaker configuration.
2. The speaker directions are converted from cartesian coordinates to spherical coordinates to be used by the spherical harmonic functions.
3. The decoder values are calculated using the specified format and returned to the server.
4. The server updates each loudspeaker model with its corresponding decoder and sends mixer updates to the corresponding endpoint.

³⁴ <https://www.npmjs.com/package/spherical-harmonic-transform>

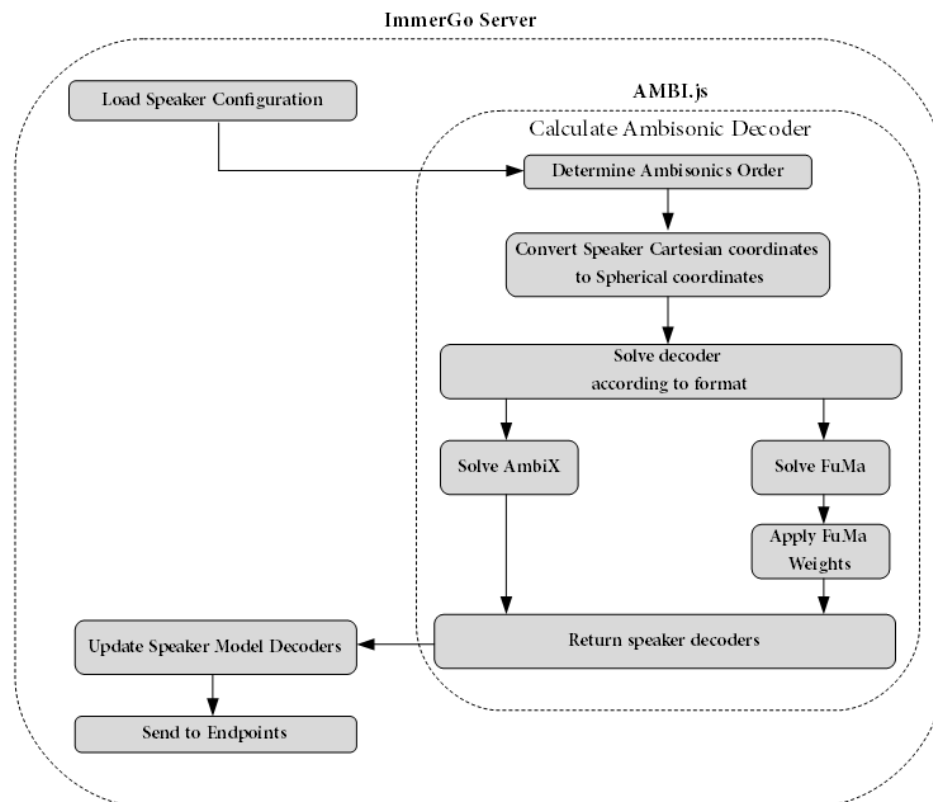


Fig. 88: Solving the Ambisonics decoder for each speaker.

Since the output of the mixer matrices is floating point, a scaling factor is used to scale the values by 10 000 prior to packetization and so each speaker's decoder components are sent as 2 byte values. This process can be seen in Fig. 89 below.

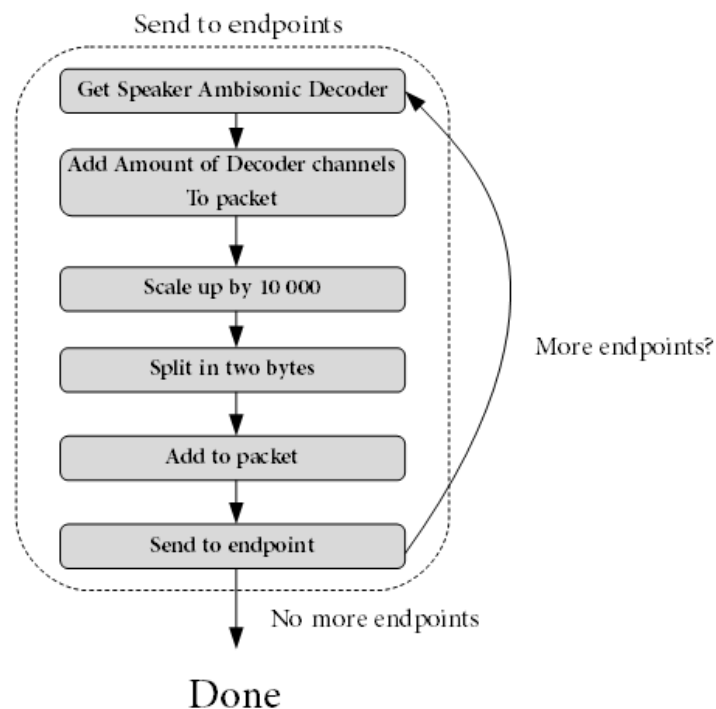


Fig. 89: Sending Ambisonic decoders to endpoints. From the ImmerGo server.

7.2.2 Ambisonics Decoding within the SAR

The SAR was modified to include another function shown in **Table 12**. This function is addressable by ImmerGo using the 0x0050 descriptor index within the AECp SET_MIXER command. When the ImmerGo server sends an AECp message with this descriptor index, the Application task passes the raw packet data to the SAR which then parses the Ambisonic decoder values and sets the mixer matrix according to these values.

Descriptor Index	Function
0x0050 : Set Ambisonic Decoder	Sets the mixer matrix values with Ambisonic decoder

Table 17: AECp SET_MIXER Descriptor Indices for Ambisonics decoding within the SAR.

The packet that is received by the SAR is shown in **Table 18** below. The payload has the first byte consisting of the number of weighting components per speaker decoder. Then the following bytes consist of the decoder for speaker 1 with each weight split into an upper and lower byte, followed by the decoder for speaker 2 also with each weight split into two bytes.

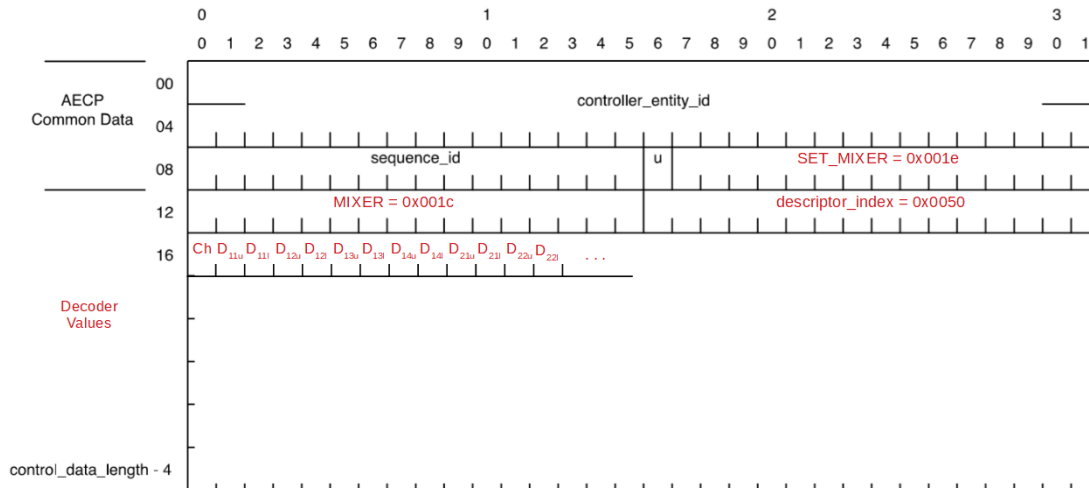


Table 18: AECp Set Ambisonic Decoder packet template.

Fig. 90 shows how this packet is parsed into the SAR. Each component is unpacked and scaled back down into floating point. This then updates the gains array for every $N \times 2$ matrix crosspoint which is dependent on the order. Once all the components have been unpacked, an update to either the XMOS or SHARC mixer matrix is made.

7.2.2.1 Setting the Mixer Matrix

Assuming a FuMa representation, where K is the total number of speakers - the decoder values for an N order Ambisonics field for speaker i , is given by $D_i = \{D_{iw} D_{ix} D_{iy} D_{iz} \dots D_{iN}\}$ such that $0 < i \leq K$. Each decoder value is determined by the Ambisonic library. These values are used to set the mixer matrix as illustrated in **Fig. 91** below. Here, column 1 is the decoder for speaker 1, with each track input a component of the Ambisonic signal. Column 2 is the decoder for speaker 2. This is then extended to the remaining speakers with their own decoder values.

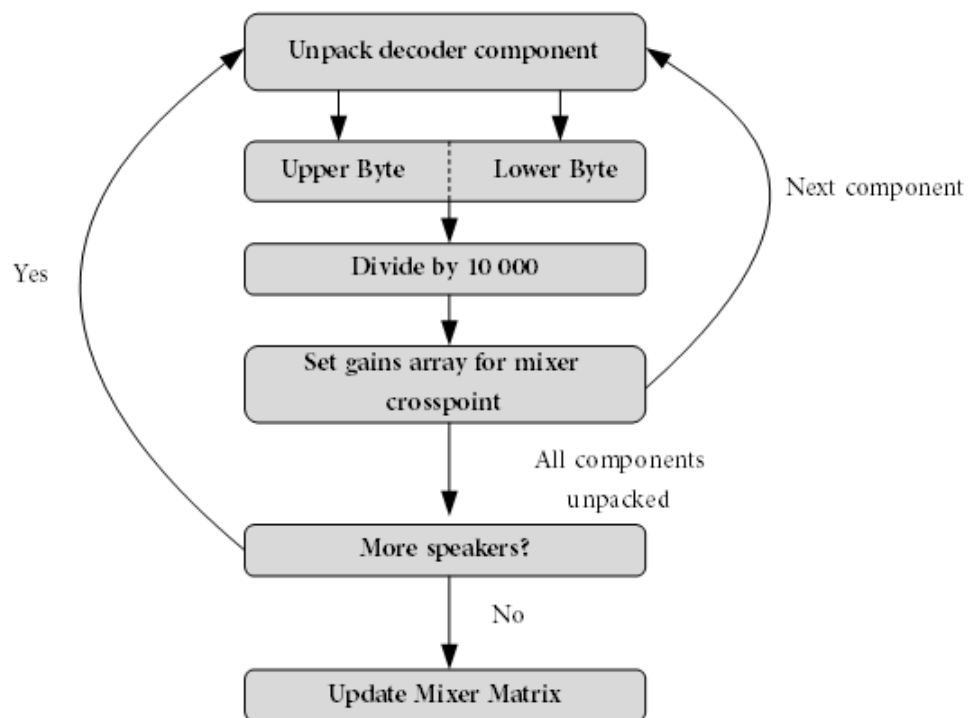


Fig. 90: Parsing the decoder values into the SAR.

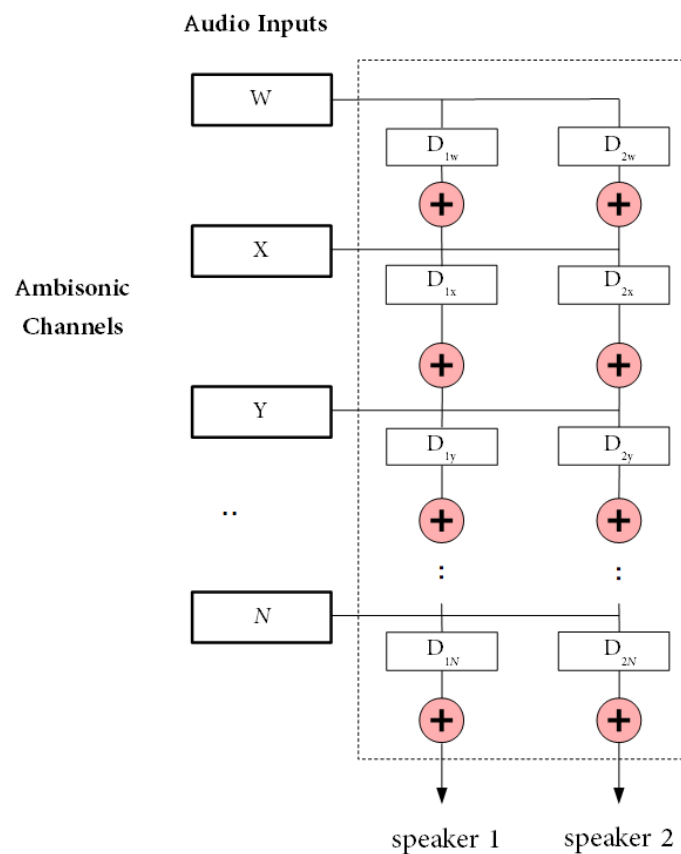


Fig. 91: Setting the endpoint mixer matrix according to decoder mixer values.

After the decoder values are received from ImmerGo, the SAR converts the floating point values into 8 bit signed chars within the XMOS. This ensures that the summation process is able to weight components negatively prior to the summation. The XMOS mixer matrix has its parameters set using the *i_imr* interface which connects the SAR task to the Audio Buffer Manager, not unlike how it is done within the DBAP and VBAP implementation.

7.2.2.1.1 Setting the SHARC Mixer Matrix

The Ambisonics mixing can also be performed using the SHARC mixer matrix. The mixer matrix is updated with the decoder values using the *DSP_AddrParaLoad* function, not unlike DBAP and VBAP. Of course, the gain values need to be converted into Q1.15 and concatenated into a 32 bit integer with the least significant bits set to 0 since no delay processing is required. The parameter buffer is then sent across to the SHARC with a starting address of 0 and the total message count. The SHARC sequentially updates each mixer crosspoint until all the entire mixer matrix has been set.

The SHARC is able to mix 32 tracks using the Q1.15 representation, and so the mixing happens at a higher resolution than that of the XMOS mixer.

7.3 Summary

This chapter has covered the spatial audio rendering algorithm known as Ambisonics. As shown in this chapter, Ambisonics is different from the other object-based rendering techniques such as DBAP, VBAP and WFS because it has an encoding and decoding stage. The encoding stage encodes the directionality of multiple objects into a known finite number of audio channels. The decoding stage mixes all of these audio channels together according to the direction of the speaker it is decoding to. The key difference with this decoding approach is that the mixer matrices used to mix the audio channels need only be set once.

The functions used to perform both the encoding and decoding stage are known as the spherical harmonic functions. These functions are used to encode the directionality of an audio object into separate components at different directions in a sphere. These components are essentially audio channels and are known as the encoded Ambisonic soundfield. As shown in this chapter, the number of channels that comprise the Ambisonic sound field are dependent on the order of the spherical harmonics used. The Ambisonic soundfield can be ordered in different ways, depending on the format it was encoded with, which is important when moving to the decoding stage. The FuMa, AmbiX and TBE formats were discussed to provide further information on how ImmerGo was required to decode these different formats.

This chapter covered an implementation of Ambisonics within ImmerGo. It has been implemented as a library that exposes a function to ImmerGo. The function determines the decoder values for speakers according to their directions within the speaker layout. To save these decoders to particular loudspeakers, the loudspeaker definition model was modified to contain a further variable to hold its decoder. The Ambisonics library is able to determine the decoder values required to decode AmbiX and FuMa formats. The SAR was modified to receive the Ambisonic decoder for the speakers it is connected to. When the decoder values are received, the SAR is able to set the XMOS mixer matrix according to these values. The SHARC mixer matrix is also able to be set according to the decoder values but required different firmware.

Chapter 8

Testing and Results

A series of tests were designed to evaluate how the successive levels of distribution compared with one another. The tests were designed to provide the following information,

1. Subjective listening evaluation to ensure the algorithms operated correctly.
2. Host processor and XMOS processor timing tests to ensure algorithm calculations were able to be completed within the 8.3ms MTC quarter frame interval.
3. Host processor load tests that compare step 1 of the endpoint distribution, where the rendering calculations take place in the server, against step 2 and 3 where the rendering calculations take place in the endpoints.

The hardware setup for these tests consisted of the host computer, which was a Mac Mini with an Intel 2.6Ghz i5-4780U processor³⁵. This processor contains 2 processing cores with hyperthreading³⁶. The Mac Mini had the MacOSx High Sierra operating system and ran the ImmerGo NWJS application.

For the purposes of proving the hypothesis, only relative computation figures needed to be obtained to determine a comparison of each distributed approach.

8.1 Listening Tests

Listening tests were conducted to ensure the algorithms were implemented correctly. The tests were held double-blind such that the tester and subjects did not know which source location was being rendered. The sound of a male voice saying “this is an immersive audio test” was looped for each test. Each location in the test was distributed uniformly around the speaker configuration. A random number generator was then used to determine the playback positions of the sources to the listener, without any repeated locations. The order of the positions was then recorded and logged to a file. For each position, the listener was instructed to point to the direction of the sound source and say in words if it was above or below them. Once identified, the tester then marked the direction that corresponded most closely with a known panned position. The perceived directions were then cross-checked with the output from the random number generator.

A total number of 10 people were tested for each algorithm, with each test having 10 different locations to identify. WFS and Ambisonics required different loudspeaker arrangements to that of DBAP and VBAP, and as such slightly different test environments were constructed.

³⁵ https://everymac.com/systems/apple/mac_mini/specs/mac-mini-core-i5-2.8-late-2014-specs.html

³⁶ <https://ark.intel.com/products/83507/Intel-Core-i5-4308U-Processor-3M-Cache-up-to-3-30-GHz->

The test setup comprised the host computer connected to a Netgear AVB 24 port PoE switch³⁷. Also connected to the Netgear switch were 7 SPK-4Ps loaded with firmware that incorporated the SAR and SHARC mixing, which allowed for each algorithm to be performed on the multiple channels. Each SPK-4P was connected to its own SPK-4 via the passive output. The Reaper DAW provided both the audio and MTC to the ImmerGo server within the host.

8.1.1 DBAP and VBAP Listening Tests

The DBAP and VBAP listening tests were performed within a 14 speaker, hemispherical layout as illustrated in Fig. 92 below.

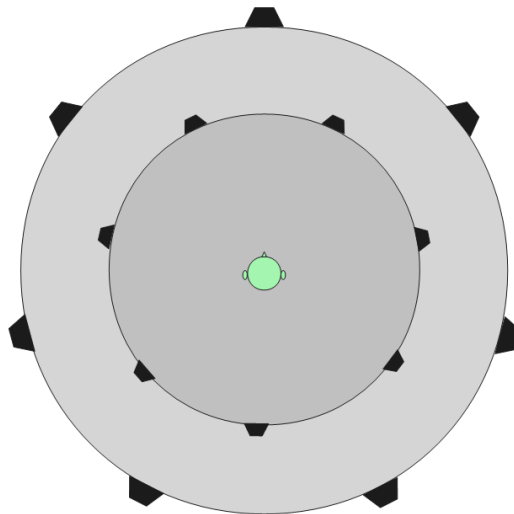


Fig. 92: Speaker Layout for DBAP and VBAP.

As shown in Fig. 93, a camera was situated above the listening position to provide a better vision of the directions that a subject pointed to. It also allowed for each test to be recorded for later review.



Fig. 93: Camera above the hemispherical array.

³⁷ <https://www.netgear.com/landings/poe/>

In order to dampen room reflections, the hemisphere was covered entirely with duvets as shown in Fig. 94 below.



Fig. 94: Duvets covering the hemispherical array.

The RT60 measurement of the room (i.e the time it takes for room reflections to decay by 60dB) was measured with the miniDSP UMIK-1 measurement microphone³⁸ that was situated at the listener position. An impulse response was emitted as a sine sweep of 100Hz-10kHz over 5 seconds from the first speaker in the layout. The microphone then captured the impulse response for the room.

The Room EQ Wizard (REW) provides the ability to determine the RT60 of a space using a recorded impulse response. The recording from the UMIK-1 was analyzed with the microphone's calibration file to reduce imperfections that occur from the microphone. Once processed, REW output the RT60 of frequency bands as a graph as shown in Fig. 95 below.

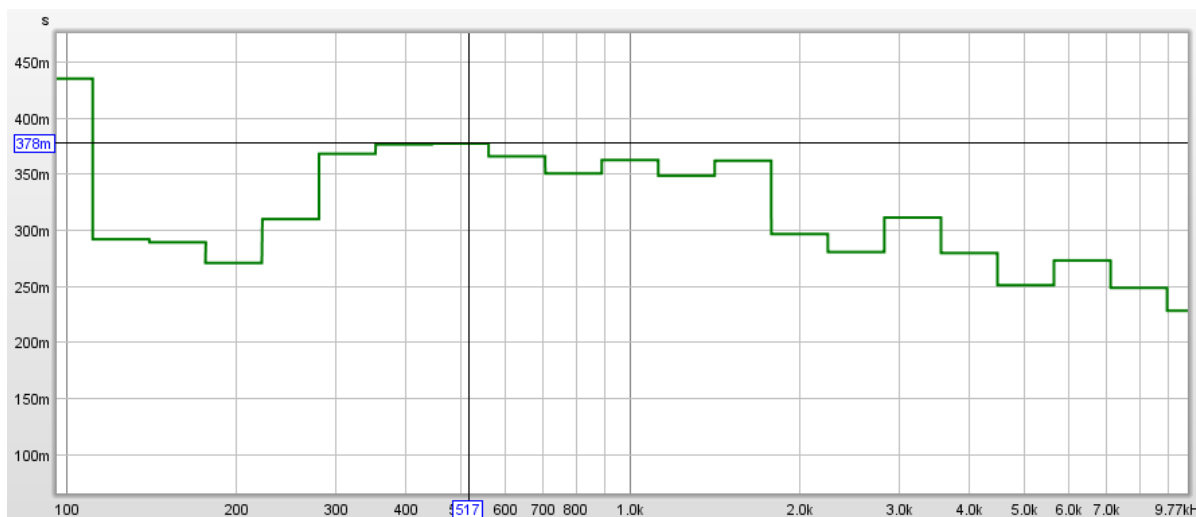


Fig. 95: The RT60 analysis given by REW.

³⁸ <https://www.minidsp.com/products/acoustic-measurement/umik-1>

The frequencies lower than 100Hz are omitted as these were removed in the playback of the voice. The graph shows each frequency on the x-axis, and the time taken to decay by 60dB in the y-axis. Each straight line is a width of a third of an octave band. As can be seen, the RT60 did not go higher than 377ms except for the extremely low frequency of 100Hz which was indicated as an unreliable reading by REW. The RT60 then decreased abruptly in the 2kHz octave and tapered down to 230ms close to 10kHz. This reading is satisfactory for a clear voice to be heard in the test.

8.1.1.1 DBAP Listening Test

Fig. 96 below shows the layout of the DBAP testing environment as seen on the ImmerGo client UI. The first 4 test positions were situated at the bottom of the hemisphere with no elevation. Test position 4 to 8 were positioned at 45-degree angles of elevation. Test position 9 is located at the origin underneath test position 10 which was situated at the top of the array where a “voice-of-god” speaker would typically be located.

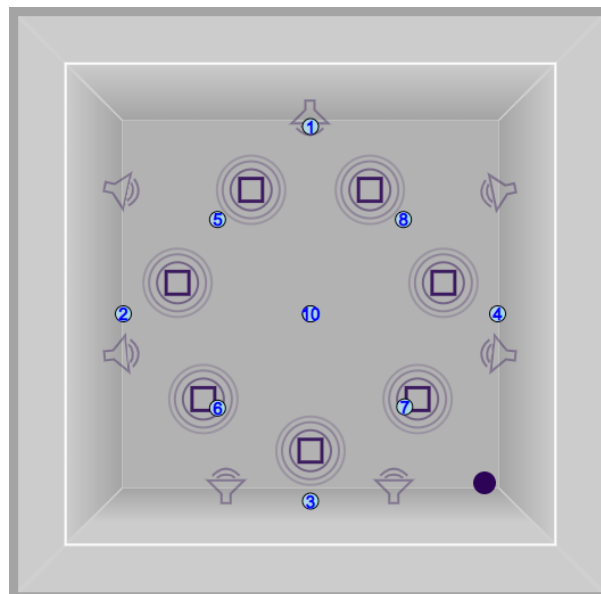


Fig. 96: DBAP listening test source locations.

The test showed a total of 66% of the total guesses were correct for DBAP. The graph shown in **Fig. 97** below illustrates the results of the test. The x-axis has the positions that were generated, and the y-axis is the positions that were guessed. The numbers at each point depict the number of subjects that guessed the position.

As can be seen, by this graph, DBAP performed satisfactorily. A position that was the hardest for listeners was position 9 centred at their head. Many listeners confused position 9 for position 10 which is likely due to the fact that they moved their head slightly off centre. Position 5 was often mistaken for positions closer to 10 and 9, which may be due to the fact that they were moved off centre. However, with all the incorrect guesses, the general location of the source was located correctly. This means that positions that were panned above, were generally located above, and positions panned, left, right, front and behind were generally located in their respective areas, even though the accuracy was not very high.

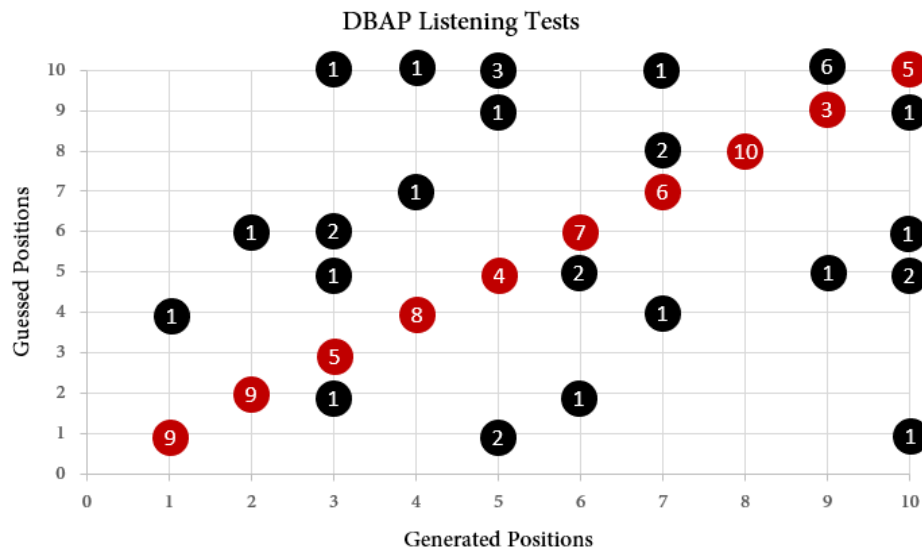


Fig. 97: Distribution of incorrect and correct responses for DBAP listening tests.

8.1.1.2 VBAP Listening Test

The VBAP listening test source positions are shown in Fig. 98 below. Here we have the same positions as DBAP, except position 9 has been moved to the front of the speaker array, about a 45 degree angle elevated from position 1.

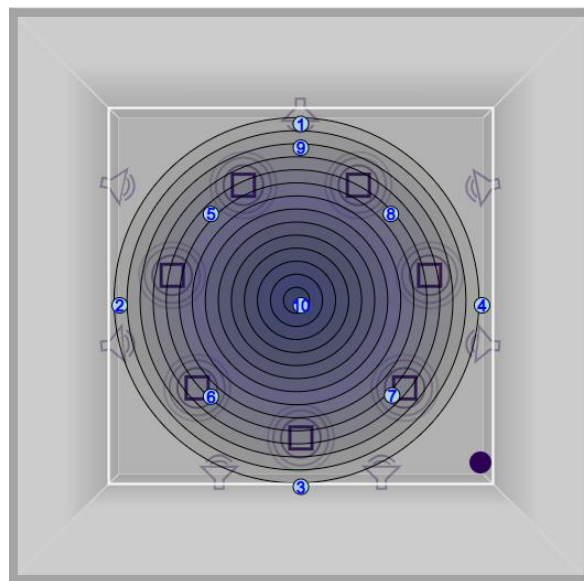


Fig. 98: VBAP listening test source locations.

The total percentage of correct guesses for VBAP was 74%. The graph in Fig. 99 below shows the results in a similar format to that of DBAP. The results show that VBAP performed accurately with most of the generated positions identified correctly. Position 10 had the most positions incorrectly identified with listeners generally locating the source close to either three of the speakers in the active speaker triplet. At least 50% of the positions panned to right above the listener were identified correctly which is positive considering there was no “voice of god” speaker in this array.

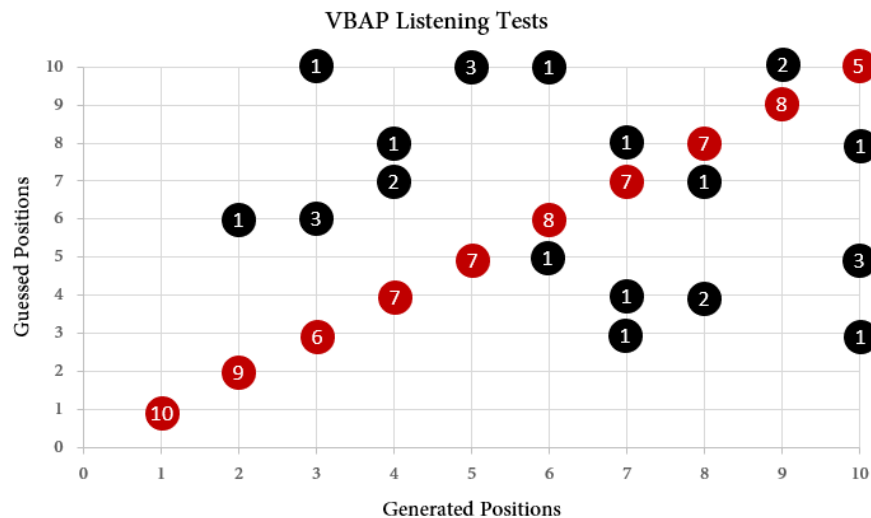


Fig. 99: Distribution of incorrect and correct responses for VBAP listening tests.

8.1.2 WFS Listening Test

The WFS listening test required a different speaker setup. An array of speakers situated on a straight line was set as shown in **Fig. 100**. The listener was positioned 2 metres away from the array and was given the option to walk parallel to the array along the reference position line if it would help to localize the sound. In order to provide an orientation of the generated sounds prior to testing, the listener was played each of the test positions and given a brief description of how the algorithm works when projecting focused and unfocused sources.



Fig. 100: WFS speaker array.

As shown in **Fig. 101**, the test sources were laid out both in front and behind the array. 5 sources were placed in front of the array, with position 9 being closest to the listener position at 2m. 5 sources were placed behind the array such that they should be roughly 1m behind the array, with position 10 being synthesized 2m behind the array.

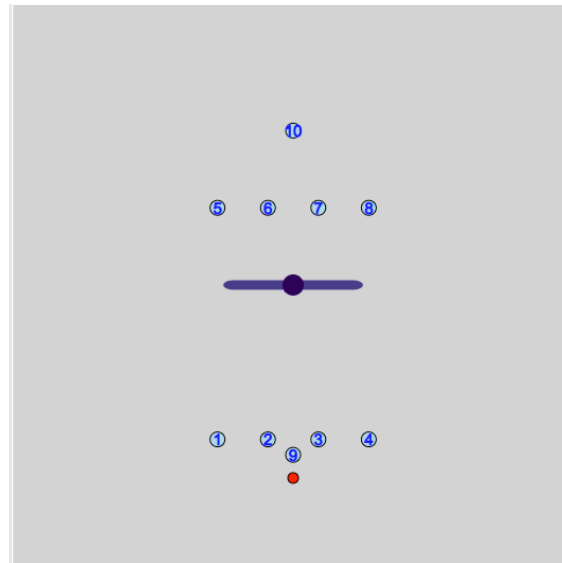


Fig. 101: WFS listening test source locations.

A total of 71% of guesses were perceived to be located at the place they were synthesized. Fig. 102 is a graph showing the distribution of correct to incorrect guesses. As can be seen in this graph, the subjects guessed correctly for the majority of the positions shown as the diagonal red circles. Position 4 had the least correct when a listener often mistook the source position to be slightly to the left at position 3 or further to the left at position 9. The most correct guesses were directly behind the array at position 10.

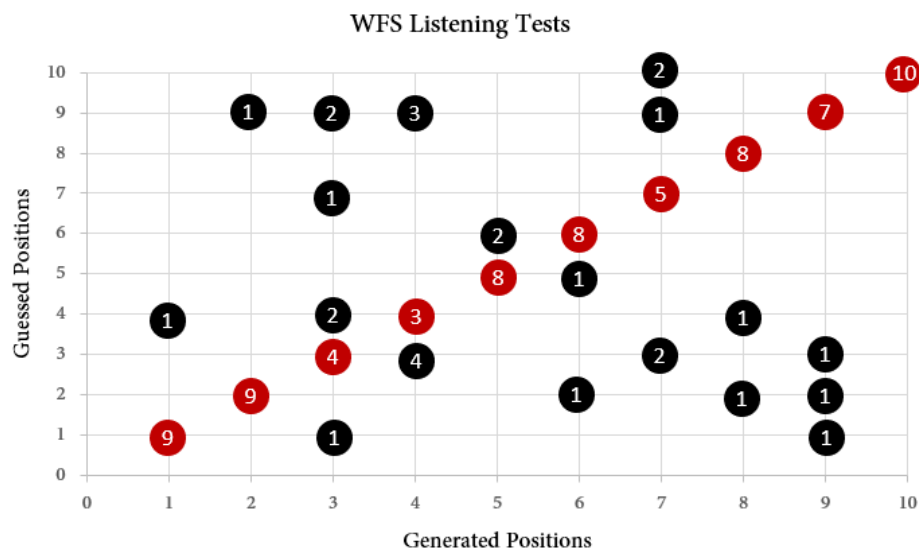


Fig. 102: Distribution of incorrect and correct responses for WFS listening tests.

8.1.3 Ambisonics Listening Tests

The Ambisonics test required that the speakers be positioned in a cube formation. The voice was encoded into a 1st order Ambisonics format using the Reaper ATK plugins³⁹ which encode a mono signal into FuMa B-format. The decoder was set up as a 1st order B-format decoder.

The testing positions for Ambisonics are shown in **Fig. 103** below. Here positions 1 through 5 are located at points between the speakers, and positions 6 through 9 are positioned in a square formation elevated 45 degrees from the origin position. Position 10 is situated directly above the origin position.

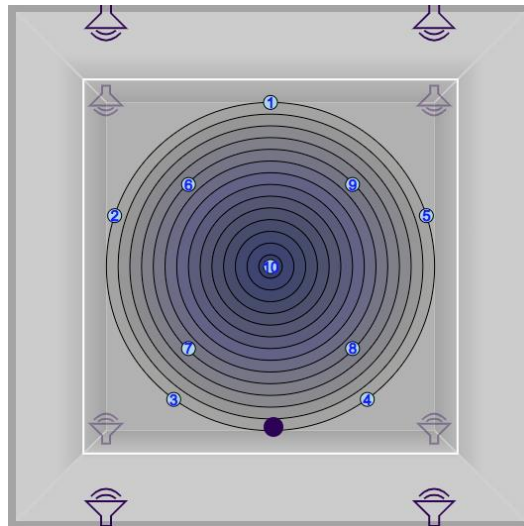


Fig. 103: Ambisonics listening test source locations.

A total of 55% of the guesses for Ambisonics were correct. The graph in **Fig. 104** below shows the distribution of the correct to incorrect guesses. As can be seen from this graph, the majority of the guesses were in similar areas. For example, when position 5 was played back, it was sometimes confused for a source higher, closer to position 9 and position 8 which were on the same sides of the listener.

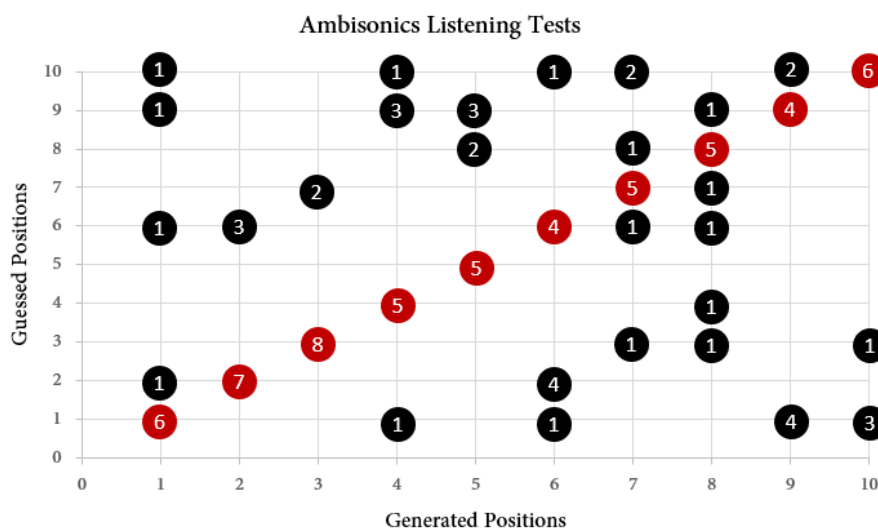


Fig. 104: Distribution of incorrect and correct responses for Ambisonics listening tests.

³⁹ <http://www.ambisonictoolkit.net/documentation/reaper/>

8.2 Algorithm Time Tests

When panning track sources with DBAP, VBAP and WFS, new mixer values need to be calculated within a time interval of 8.3ms determined by the intervals between each MTC quarter frame. As the number of speakers and track sources panned simultaneously increases, so does the amount of time taken to calculate the new mixer values.

In order to gain insight into exactly how much time these algorithms take to perform their calculations, each of the algorithms were measured. This was done for each step of distribution.

For the first step of distribution, the NWJS application performed the algorithm calculations and the XMOS processors performed the mixing. For the subsequent steps of distribution, the XMOS processor performed the calculations and the mixing was handled internally either within the XMOS or the SHARC DSP mixers. This means that the NWJS application and the XMOS processors needed to be timed.

8.2.1 NWJS Time Profiling

The timing measurements for the NWJS application were done using the Web “Performance” API. The Performance API is specifically designed to profile web applications⁴⁰. In particular, the high-resolution timer⁴¹ is used to take a measurement at the start and end of a particular section of the application code.

Each algorithm was tested with a different number of speakers for a varying number of sources. The speakers were generated randomly such that they fit the geometry required by each of the algorithms. An increasing number of 14, 25, 50, 75 and 100 speakers were generated for each algorithm as follows,

1. DBAP was generated as a random number of speakers spanning an area of 10x10m.
2. VBAP was generated on a sphere with a radius of 10m such that each speaker was placed on the perimeter of the sphere with a unique horizontal and vertical angle.
3. WFS was generated as a line of speakers.

The algorithm then panned an increasing number of sources that correlate with the number of tracks that may be sent in a single AVB stream – 1, 8, 16, 24 and 32 track sources – to these speaker configurations. In order to avoid granularity problems caused by the non-deterministic nature of the MacOSx scheduler, each algorithm was run 1,6 million times, and then the total time of this was divided by 1,6 million to provide the measure of time to process the algorithm.

8.2.1.1 Time Taken to Perform DBAP within the NWJS Application

Fig. 105 below shows the timing measurements of the DBAP algorithm. As can be seen from this graph, NWJS was able to perform the DBAP algorithm calculations well within the 8.3ms window. For 100

⁴⁰ <https://developer.mozilla.org/en-US/docs/Web/API/Performance>

⁴¹ <https://developer.mozilla.org/en-US/docs/Web/API/Performance/now>

speakers, the worst case would be 32 track sources panned simultaneously which only took 0.28ms to be calculated. The best case, being 1 source, was well below 0.02ms.

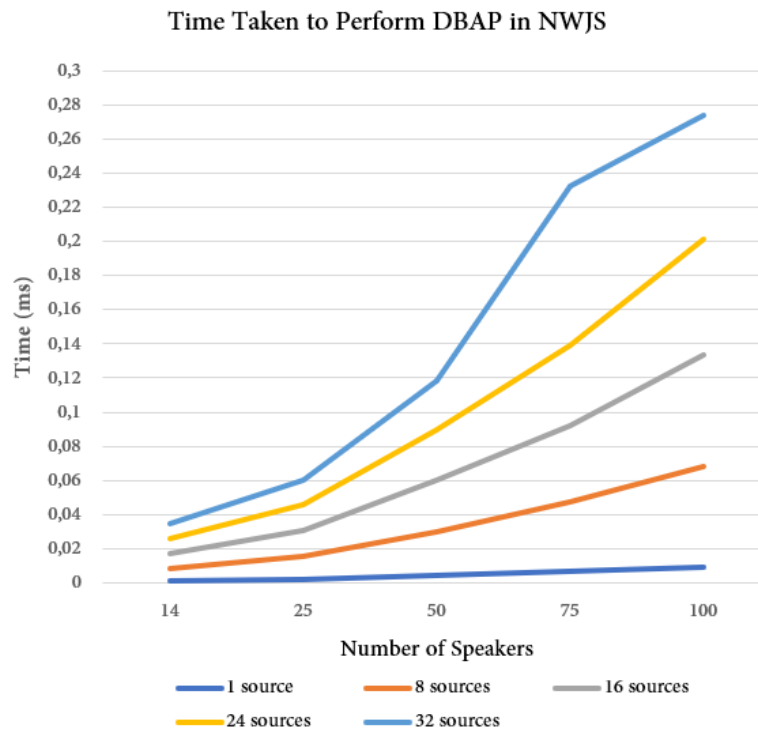


Fig. 105: Time taken to perform DBAP within the NWJS application.

8.2.1.2 Time Taken to Perform VBAP within the NWJS Application

The results showing the time taken to perform VBAP by the host processor are shown in Fig. 106. As can be seen from this graph, VBAP took significantly more time than DBAP to complete. For 100 speakers, the worst case of 32 tracks reached 4.5ms. The best case panning only 1 source is just below 0.25ms. This is due to the fact that the VBAP algorithm as implemented in this thesis, is required to iterate through every face in the convex hull, and perform the matrix calculation for that face until it finds the correct face that is used to localize the source. Regardless, these results show that the host is able to perform the VBAP calculations within the 8.3ms window.

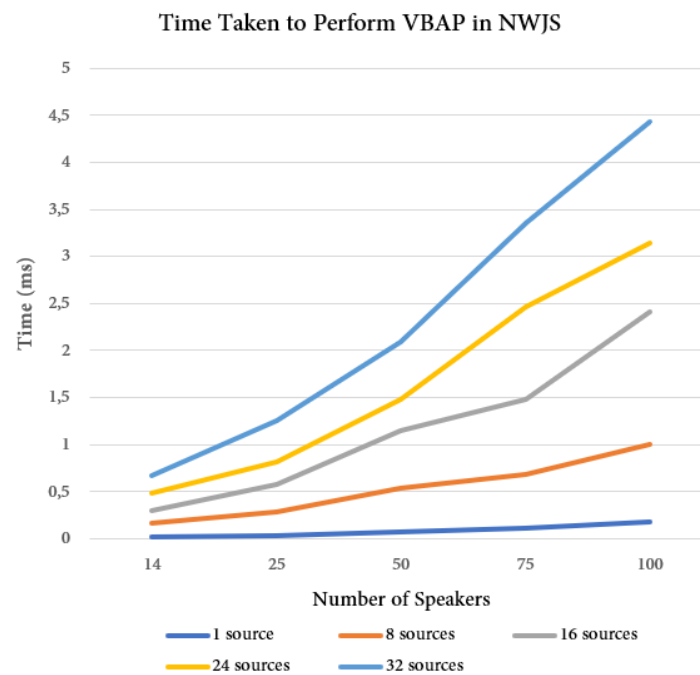


Fig. 106: Time taken to perform VBAP within the NWJS application.

8.2.1.3 Time Taken to Perform WFS within the NWJS Application

The graph showing the time taken to perform the WFS calculations is given in Fig. 107 below. The graph shows that it lies between DBAP and VBAP in terms of the amount of time taken for the host to complete its processing. For 100 speakers, panning 32 track sources results in roughly 1.6ms of time. When panning a single source to 100 speakers, the time taken is well below 0.25ms. As can be seen from the graph, the amount of time taken seems to increase with a large gradient from 25 speakers and up. Still, these results show that the host processor should be able to complete the WFS calculations within the 8.3ms window.

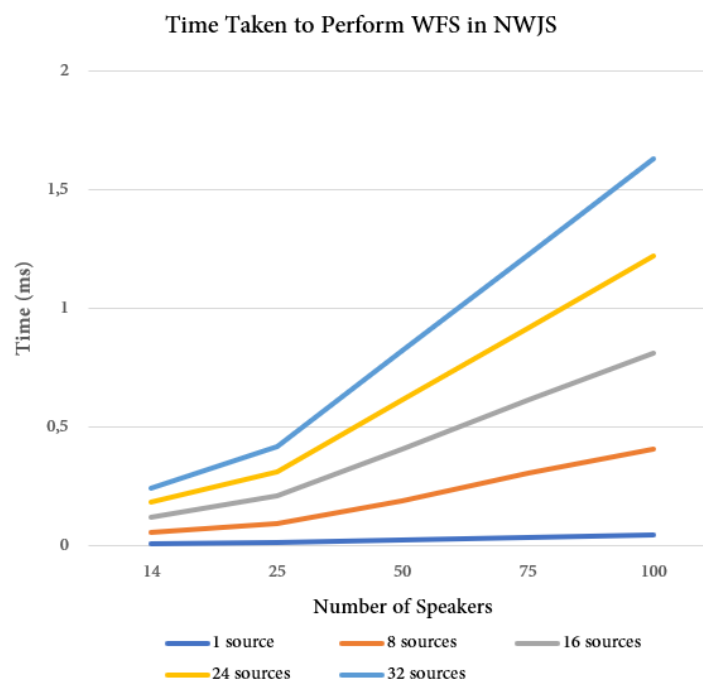


Fig. 107: Time taken to perform WFS within the NWJS application.

8.2.2 XMOS Time Profiling

The xTIMEComposer Time Analysis (XTA) tool was used to perform profiling tests on the XMOS processor to determine how each algorithm performed within the endpoint processors and what the limits of the processor are. Each stage of distribution needs its own particular processes. Profiling tests for each stage of distribution of development were done to provide a better understanding of how each stage performed.

The tests were performed using a differing number of speakers and sources as follows,

1. 1, 8, 16, 24 and 32 sources.
2. 10 – 100 speakers with 10 speaker intervals.

8.2.2.1 XTA Results for Step 1 of Distribution

The first step of the endpoint distribution was to perform the rendering calculation within the host and then send the output of the calculation to the endpoints. The host would calculate the XMOS mixer values for the speakers according to a spatial audio algorithm. Then, each endpoint would receive an AECP packet holding the mixer matrix parameters, which were subsequently unpacked. These mixer values were then used to set the XMOS processor's mixer. Regardless of the number of speakers, each endpoint would have the mixer matrix set after each renderer calculation and the time taken to set the mixer would always remain the same, regardless of the number of speakers for a given number of track sources.

The timing results for the host calculations are given in the subsections above. Once the mixer values were sent to the endpoints, two processes were performed within the XMOS processors,

1. Unpacking the mixer values.
2. Setting the XMOS mixer to mix the audio.

As shown in **Fig. 108**, the unpacking and mixing of audio to the two loudspeakers attached to the processor took a total of 7.76µs for 32 track sources.

```

~ ★ function test_step1 : 7.76 us / 7.76 us
  > ≡ block : 20.0 ns (x 1) / 20.0 ns (x 1)
  > ★ function unpackMixVals : 4.71 us / 4.71 us
  > ≡ block : 10.0 ns (x 1) / 10.0 ns (x 1)
  > ★ function audio_buffering_simulation : 3.01 us / 3.01 us
  > ≡ block : 10.0 ns (x 1) / 10.0 ns (x 1)

```

Fig. 108: XTA showing time taken to perform attenuation based mixing within XMOS according to mixer parameters received from the host.

When performing Ambisonics, the above method is used to set the mixer matrix to the decoder values. Once set, no further updates need to be made to the speaker endpoints unless the speaker configuration changes. As such, in the subsequent sections, Ambisonics will be left out of the timing analysis.

The WFS implementation was different, as it required an XMOS based mixer to be implemented with delay buffers. The memory of the XMOS processor is constrained to 16 delay buffers of 1024 samples. This means that a total of 8 track sources may be implemented, with each speaker having a delay buffer for a

track. As shown in Fig. 109, the total amount of time taken to unpack and perform the attenuation-delay based mixing is 11.07µs/0.01107ms for 8 track sources.

```

▼ ★ function test_step1wDelay : 11.07 us / 10.75 us
  > ≡ block : 20.0 ns (x 1) / 20.0 ns (x 1)
  > ★ function unpackMixDelayVals : 3.11 us / 3.11 us
  > ≡ block : 10.0 ns (x 1) / 10.0 ns (x 1)
  > ★ function audio_buffering_simulation_wDelay : 7.91 us / 7.59 us
  > ≡ block : 20.0 ns (x 1) / 20.0 ns (x 1)

```

Fig. 109: XTA showing time taken to perform attenuation-delay based mixing within XMOS according to mixer parameters received from the host.

8.2.2.2 XTA Results for Step 2 and 3 of Distribution

As has been shown in the above subsection, the XMOS processor is able to manage the mixing of the audio within a short amount of time. For subsequent stages of distribution, steps 2 and 3, the spatial audio rendering calculations are performed within the XMOS processor. These calculations require significantly more time to complete as they are more complex than simply setting the mixer parameters.

Step 2 of distribution required that the XMOS processor calculate mixer values according to track object updates received from the host and then set the internal mixer matrix whilst having the SHARC processor bypassed. Each processor was required to perform 3 main steps,

1. Unpacking the track object model.
2. Solving mixer values according to an algorithm.
3. Mixing within the XMOS processor.WW

Step 3 of distribution is similar to step 2, except the audio mixing is done in the SHARC DSP chip instead of the XMOS. This requires the following,

1. Unpacking the track object model.
2. Solving mixer values according to an algorithm.
3. Loading the mixer values to the SHARC mixer.

It is shown in the following sections that these two different processes require similar amounts of time due to the fact that the time taken to perform the mixing of the audio is negligible when compared to the time taken to perform the calculations.

8.2.2.2.1 XTA Timings for Endpoint Distribution of DBAP

The graph in Fig. 110 shows the output given by the XTA for the various test cases for DBAP at step 2 of distribution. The y-axis shows the amount of time in seconds and the x-axis shows the number of speakers. Each line in the graph represents the time taken for a particular number of sources being panned simultaneously.

As seen in this graph, the amount of time taken by the number of sources increases linearly with each interval of speakers. These results show that for the worst case of 100 speakers, only the calculations for a single source being panned were able to be completed within the 8.3ms window. 8 sources would be able to be panned simultaneously to any speaker array under 15 speakers. Once the number of sources increased

above 8 sources, the XMOS processor was not able to complete the calculations in time. The worst case of 32 track sources being panned simultaneously to 100 speakers resulted in a time of 172ms, way beyond the 8.3ms window. Of particular mention is the amount of time it took to perform the square root calculation. This was the most time-consuming operation which took $13.87\mu\text{s}$ and was very large in comparison to other operations.

DBAP Times in XMOS for Step 2 of Distribution

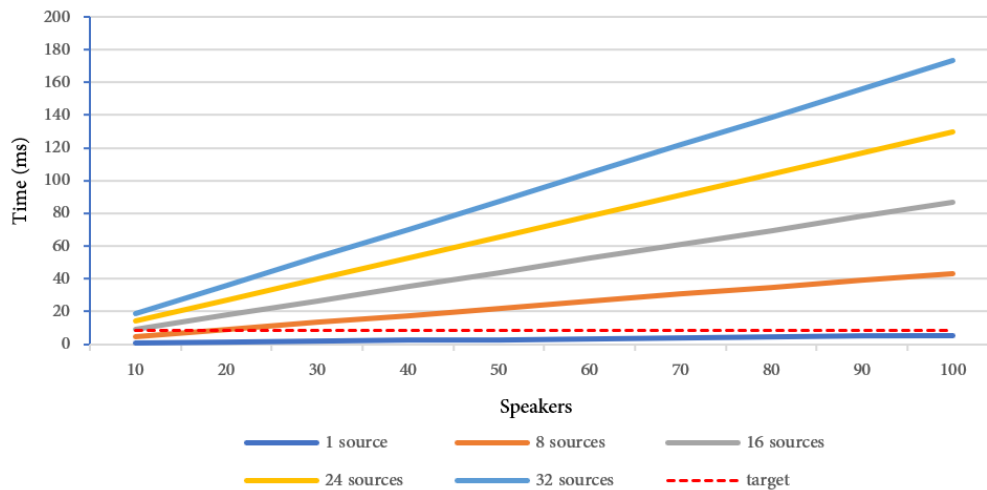


Fig. 110: XTA time analysis of the DBAP algorithm with the second step of distribution.

Fig. 111 shows the results of the DBAP implementation with further distribution of mixing to the SHARC. As seen in this graph, the amount of time taken to perform the DBAP calculations and loading the output into the SHARC is very similar to step 2. The way in which the XMOS loads the mixing parameters onto the SHARC takes a similar amount of time to the process of setting the XMOS mixer and mixing the audio samples.

DBAP Times in XMOS for Step 3 of Distribution

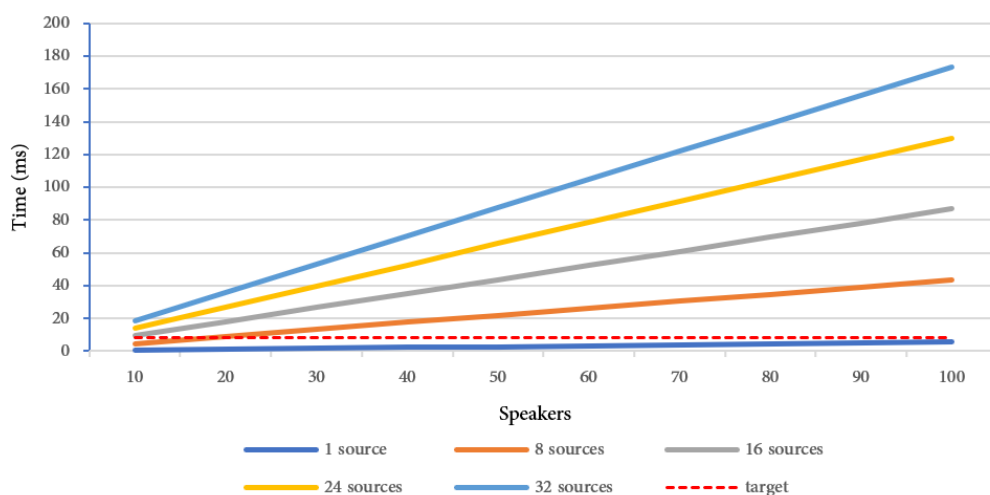


Fig. 111: DBAP times for step 3 of distribution.

8.2.2.2.2 XTA Timings for Endpoint Distribution of VBAP

As described earlier in section 8.2.1.2, the time taken to perform VBAP is dependent on the number of faces in the convex hull. A series of random speaker arrays were generated and the number of faces in the hull corresponding to the speaker size was used for the number of loops in the calculation.

The graph showing the amount of time taken by the XMOS processor to calculate VBAP is shown in Fig. 108. This graph shows that the XMOS processor was only able to perform the panning calculations for 1 source in time, which was limited to 80 speakers. 8 sources were able to be panned simultaneously to a maximum of 10 speakers. For all other numbers of sources and speakers, the calculations were unable to complete within the 8.3ms window. Furthermore, the amount of time taken drastically increases above the 8.3ms time to 320ms for the worst case of 32 sources panned to 100 speakers simultaneously.

VBAP Times in XMOS for Step 2 of Distribution

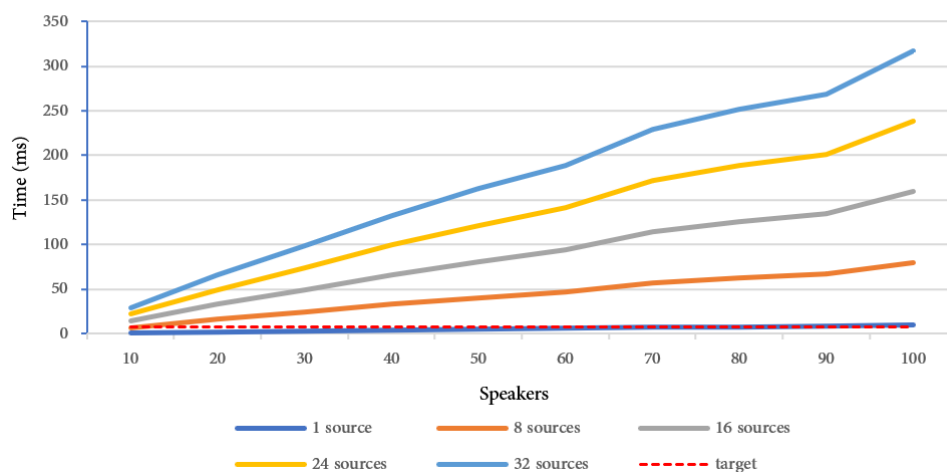


Fig. 112: XTA time analysis of the VBAP algorithm with the second step of distribution.

As shown by the graph in Fig. 113, the amount of time taken does not change significantly in comparison to step 2. The calculations required by the VBAP implementation are too time consuming to provide an effective solution when implemented on the XMOS processor.

VBAP Times in XMOS for Step 3 of Distribution

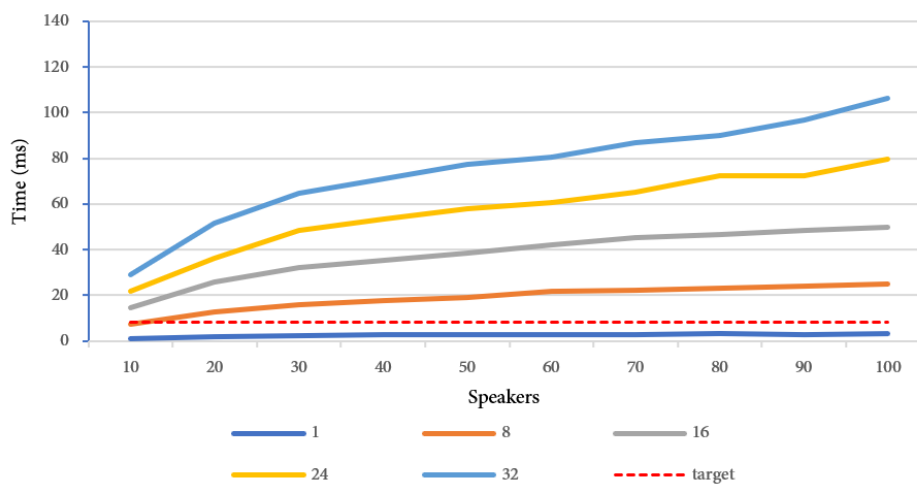


Fig. 113: XMOS timing for VBAP at varying sources and speaker sizes.

8.2.2.2.3 XTA Timings for Endpoint Distribution of WFS

The WFS calculations were required to happen on a per speaker basis. Since each endpoint was attached to at most 2 speakers, each endpoint was only required to do WFS calculations for at most 2 speakers. **Table 19** shows the amount of time taken by the XMOS processor to perform the WFS calculations. The worst case being 32 track sources panned simultaneously to 2 speakers. These results show that each endpoint was able to complete the WFS panning calculations for the speakers attached to the endpoint within 6.02355ms which is well within the 8.3ms window. This means that theoretically, real-time calculations of the WFS algorithm fit well within the endpoint distributed implementation.

As mentioned previously in section 8.2.2.1, the only constraint of step 2 of the endpoint distribution was that WFS was limited to 16 delay buffers of 1024 samples. This resulted in the XMOS only being able to delay a maximum of 8 tracks for each speaker.

Sources	Speakers	
	1	2
1	0,10459	0,19648
8	0,77743	1,51227

Table 19: WFS times for XMOS calculations and XMOS mixing.

This constraint is avoided when using the SHARC DSP mixer matrix. The SHARC mixer matrix allows for 64 delay buffers of 1024 samples to be implemented which means that 32 track sources may be mixed to the 2 speakers attached to the endpoint. **Table 20** shows the results of the WFS profiling tests. Since each SPK-4P had attached 2 speakers, the processor only needed to calculate the algorithm for the speakers that were attached to it. The results show that the XMOS WFS calculation for 32 tracks to both its attached speakers was well within the 8.3ms time limit.

Sources	Speakers	
	1	2
1	0,09683	0,18872
8	0,76967	1,50451
16	1,53863	3,00827
24	2,30759	4,51203
32	3,07655	6,01579

Table 20: WFS times for XMOS calculations and mixing within the SHARC.

8.3 Host Processor Load Tests

DBAP, VBAP and WFS required that calculations for the mixing values be solved whenever a source is moved. This requires a percentage of the CPU to perform the algorithm's calculations.

In order to determine the CPU load exerted by ImmerGo in conjunction with DAW software on the host machine, a series of panning tests were done for each algorithm. These tests consisted of panning 1, 8, 16,

24 and 32 sources to varying speaker array sizes of 14, 50, 75 and 100. The speakers were generated according to the geometry required by each algorithm similar to what was done in section 8.2.1.

To simulate a typical spatial audio project, a Reaper DAW was set up alongside the ImmerGo application. The Reaper project contained the same number of tracks required by each test, with each track having equalizer, compressor and reverb plugins in their fx chain.

Each panning test spanned 20 seconds. An automation track was created for a single source and then duplicated for further sources to ensure that ImmerGo was calculating the mixer values for multiple sources for every MTC timestamp.

The ImmerGo host application's CPU load was profiled using the "top" bash tool⁴². "Top" outputs the CPU load exerted by applications every second. A simple script was set up to log the CPU load of the ImmerGo and Reaper applications to a file. The average of these values was then calculated to determine the average amount of CPU load exerted by a particular algorithm being run. In order to gain an understanding of the load exerted by both ImmerGo and Reaper, their amount was combined.

As mentioned at the beginning of this chapter, the Intel i5 processor includes 2 cores with hyperthreading. Hyperthreading is designed to allow for multiple applications running simultaneously to run in parallel on the same core. A core with hyperthreading means that the operating system addresses 2 virtual logic cores. By doing this, it means that should an application saturate the core with operations, there is still a core available for another application's thread to be processed. It does not mean that the application saturating the core will also use the other virtual core (Marr, et al., 2002).

This can confuse the top tool measurements, as having 4 logical cores should mean that the processor should theoretically be able to have a maximum of 400% CPU available to any application, however, this is not the case as the virtual cores are saved for other applications. In certain cases, within these tests, "top" showed that the total processor usage was well over 200%. When this happened, the application slowed down which indicated that the CPU was delayed due to the operations. It was decided that for the load tests, any application running above 200% would be considered as saturating both cores. The amount of percentage above that would be the percentage of the processor that was needed to wait to be processed.

8.3.1 Host CPU Load Results for DBAP

Fig. 114 shows the results of the CPU load exerted by the host when using DBAP. Here, the dotted lines show the processor usage for a different number of sources panned with the DBAP calculations distributed to the endpoints. The solid lines show the CPU load exerted on the processor when the DBAP calculations are performed within ImmerGo.

For the worst case of 32 track sources being panned to 100 speakers simultaneously, 39% of the host CPU was used. As shown in Fig. 115, this was increased by 28% when included with the Reaper project. Interestingly, when 1 source was being panned to 14 speakers, the CPU load was similar to the calculations happening within the server. This means that the overhead of packetizing the data prior to sending to the

⁴² <https://ss64.com/bash/top.html>

endpoints was similar to actually performing the DBAP algorithm and packetizing the mixer values. However, as the number of sources increases, the CPU load escalated.

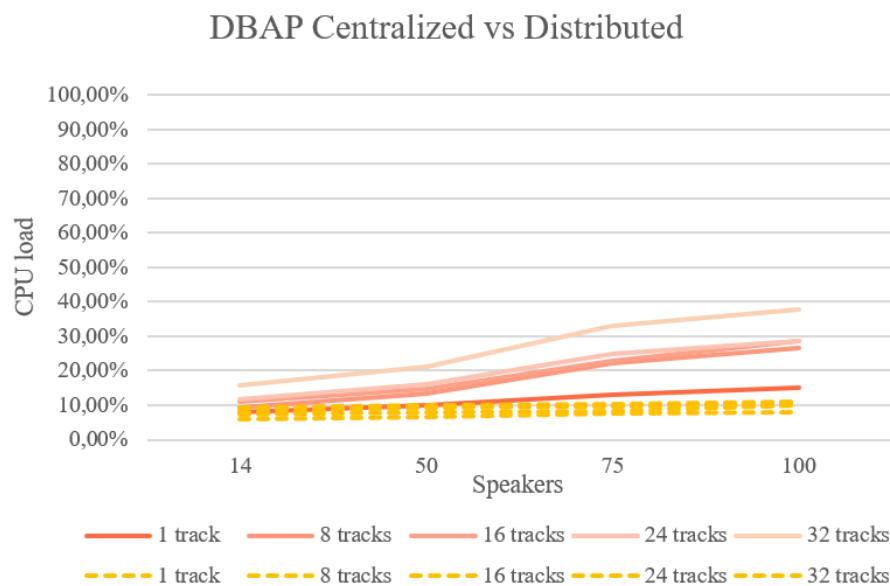


Fig. 114: Host CPU loads for DBAP.

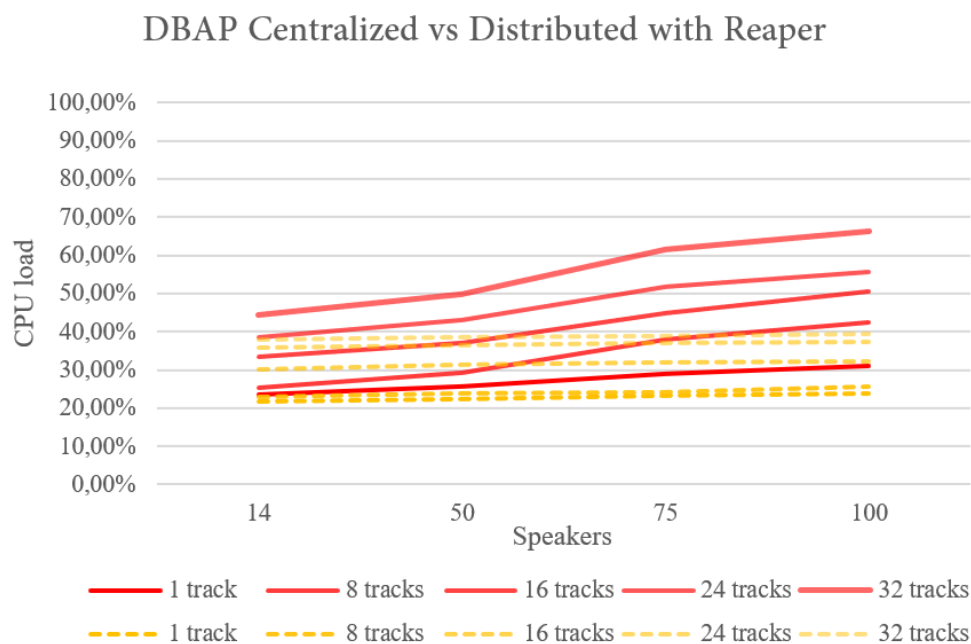


Fig. 115: Host CPU loads for DBAP with Reaper.

8.3.2 Host CPU Load Results for VBAP

The percentage of CPU being utilized when performing VBAP standalone is shown in Fig. 116 below. This graph shows that the processor benefitted greatly when using the distributed approach, with the distributed approach remaining relatively constant for the increasing number of sources and speakers. It is also shown by this graph the host processor became extremely busy when performing the VBAP calculations. As shown in these results, when panning 24 or 32 tracks to 14 speakers and above, the processor load is 200% and so is fully exerted.

The high CPU utilization was noted when doing the test as the ImmerGo application slowed down significantly when panning more than a single source. This is understandable when considering the amount of time taken by the processor to complete the VBAP calculations, as shown in section 8.2.1.2. Fig. 117 shows that when combined with the Reaper DAW, it is reasonable to conclude that the VBAP algorithm is too processor intensive for the current implementation to be an effective solution with the calculations happening in real-time. Here we see the processor usage jump to 300%.

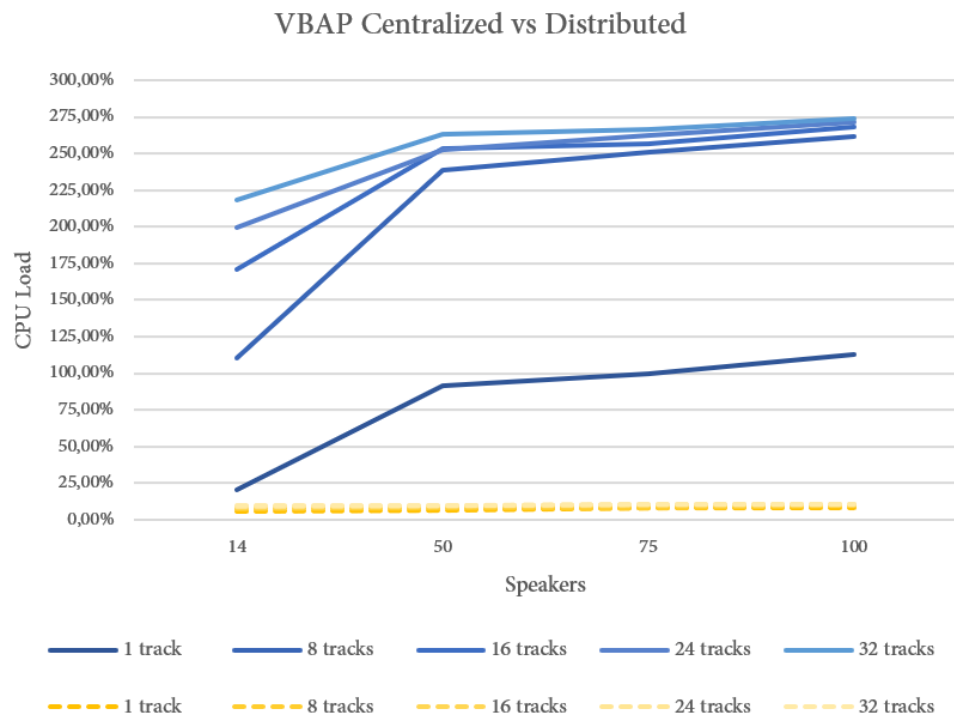


Fig. 116: Host CPU loads for VBAP.

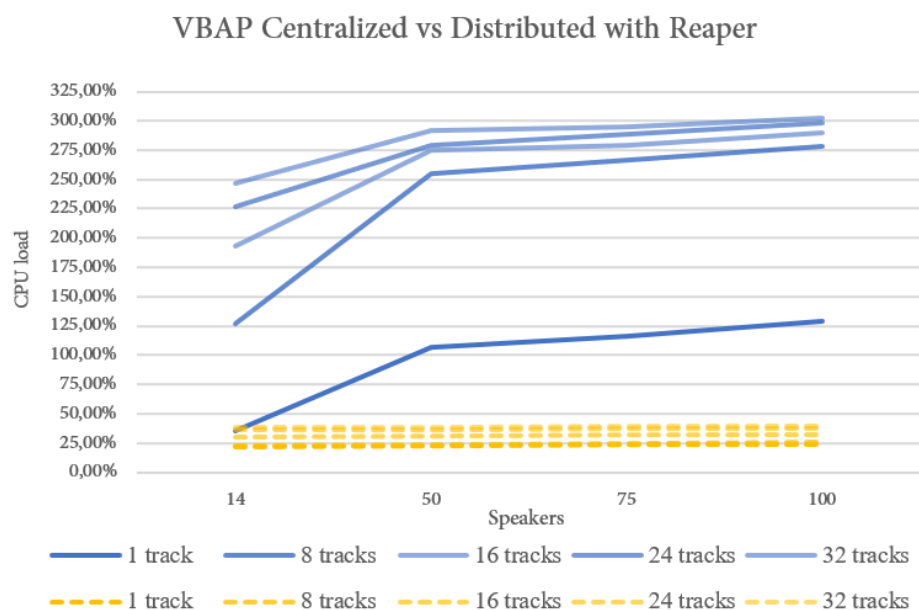


Fig. 117: Host CPU loads for VBAP with Reaper.

8.3.3 Host CPU Loads for WFS

The CPU load exerted by the WFS algorithm is shown in **Fig. 118**. WFS lies between DBAP and VBAP with how much load it exerts on the CPU. As shown in **Fig. 119**, when combined with Reaper, the processor usage increases to 110%. As with the previous algorithms, distributing the calculations to the endpoint frees up the host CPU significantly.

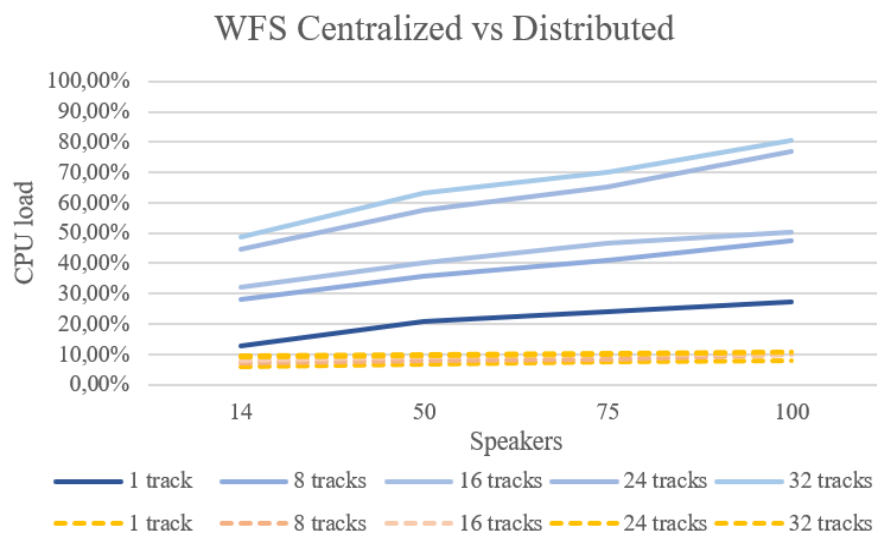


Fig. 118: Host CPU loads for ImmerGo WFS.

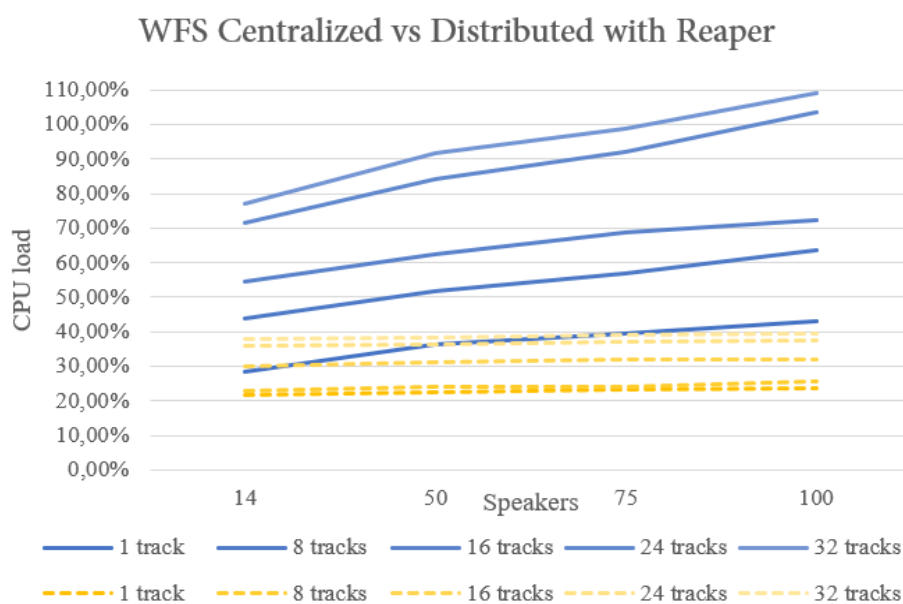


Fig. 119: Host CPU loads for ImmerGo WFS with Reaper.

8.3.4 Host CPU Loads for Ambisonic Decoding

Ambisonics decoding is different from the object-based rendering algorithms, DBAP, VBAP and WFS. The decoding process is required to have mixer matrix values set only once for the speaker configuration. As such, a timing profiling test is not required because the mixer matrix needn't be updated when a track object is moved.

In order to determine the processing load that would be mitigated by offloading the Ambisonics mixing to the endpoints, processing loads were performed on a typical Ambisonic decoder plugin. The plugin used is from the SPARTA Ambisonic suite⁴³ and has options for various decoders up to 7th order⁴⁴. Currently, the endpoints are only able to receive up to 32 channels which means that a maximum of 4th order Ambisonics is able to be performed. The following load tests show the CPU load of the Reaper DAW when decoding from 1st to 4th order Ambisonics. The loudspeaker configurations that were decoded to are given in the SADIE Ambisonic decoder database (SADIE - Spatial Audio for Domestic Interactive Entertainment, 2019),

1. 1st order – a cube – 8 speakers.
2. 2nd order – an icosahedron – 12 speakers.
3. 3rd order – 7-design – 24 speakers.
4. 4th order – a pentakis dodecahedron – 32 speakers.

As shown in Fig. 120, the CPU load exerted by Reaper without any Ambisonics decoding taking place is under 10%. When increasing the order of the encoded channel, the speaker configuration size also increases. Here we see that for 32 speakers, which could have a 4th order Ambisonic signal decoded to it, the CPU load exerted by Reaper reaches 22%.

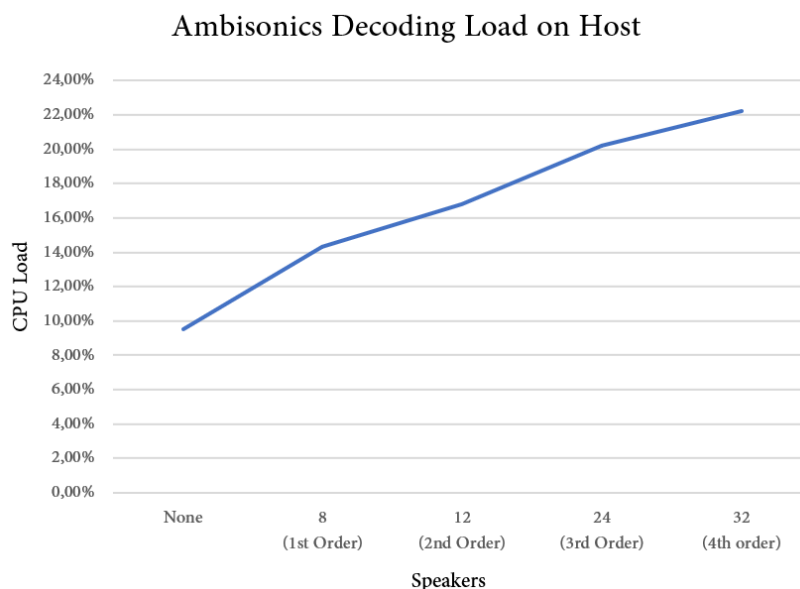


Fig. 120: CPU load exerted by Reaper with higher orders of Ambisonic decoding.

⁴³ http://research.spa.aalto.fi/projects/sparta_vsts/

⁴⁴ http://research.spa.aalto.fi/projects/sparta_vsts/plugins.html

8.4 Discussion of Results

The tests described and implemented within this chapter have shown both positive and negative aspects of distributing spatial audio rendering to the endpoints. The general positives are,

1. DBAP, VBAP and WFS can have their calculations, as well as mixing moved to the speaker endpoints such that they are still able to localize correctly, even though with constraints.
2. Ambisonics decoding can be performed successfully at the speaker endpoints.
3. The CPU load exerted on the host processor can be greatly reduced by distributing the algorithms to the endpoints, which frees up more processing power for a host audio application such as a DAW.

The results above also show that there are some issues when distributing the DBAP and VBAP calculations to the endpoints. These are as follows,

1. For VBAP and DBAP, the XMOS processor is not able to complete the worst case of the calculations of 32 tracks to 100 speakers within the 8.3ms interval. This being said, DBAP is able to pan 1 source up to 100 speakers and 8 sources to 20 speakers. VBAP was able to pan 1 source to 80 speakers, and even 8 sources to 10 speakers.
2. VBAP is extremely processor intensive, which is highlighted by the load percentages on the host processor.

The results have also shown that of the three spatial audio algorithms that require new mixer values to be calculated when track sources are moved, VBAP turns out to be the most processor intensive. This is an interesting result, as the VBAP matrix calculation is a relatively simple calculation. The problem that arises when implementing VBAP with multiple loudspeakers is that the convex hull grows. As the convex hull grows, so does the processor load, because the matrix calculation needs to be performed for each face of the convex hull in order to determine the correct triplet of speakers to use for the localization.

One method that might be possible to reduce the processing load exerted by calculating VBAP values, is to perform an “offline” rendering. This means that after the convex hull has been solved, the speaker gains for all horizontal and vertical directions are calculated and stored in a dictionary. Then, when a source is moved, the gains for a particular direction may be found in the dictionary. Of course, this would make the application more memory intensive. Since the XMOS processor has limited memory, it would be best to hold the dictionary of gains within the ImmerGo application on the host, and then perform the mixing at the endpoints.

DBAP is a good candidate to have its calculations processed on the host computer and the audio mixing performed at the speaker endpoints. The host processor would be able to calculate the speaker endpoint mixer values at the expense of processing power, however, the calculations are well within the 8.3ms window.

WFS is a good candidate for the distribution of calculations and mixing to the endpoints, as the calculations are performed on a per speaker basis. This means that each processor is required to perform

the rendering calculations for only the speakers that are attached to it. The results show that using the XMOS processor for the delay and attenuation mixing has a limited track count due to the memory constraints imposed by the XMOS processor, with only 8 tracks at maximum being able to be mixed. As such, step 3 of the distributed endpoint processing is considered the best method for distributing the processing. Not only is the SHARC able to perform more DSP processing like filtering and amplitude smoothing as discussed in chapter 6, but its processing capability also allows for 32 tracks to be attenuated and mixed. The XMOS processor is able to perform the calculation of the mixing values for the speakers attached to the endpoint, and then the SHARC mixer is able to perform the DSP operations required for its mixing.

The Ambisonics load tests show that the host CPU does benefit from offloading the decoding to the endpoints and the processing load exerted on the CPU can decrease by 12%. It can be seen by the graph that there is a steady increase in the amount of load exerted on the processor, so for larger speaker configurations, there will be a need for more processing power for the decoding stage. Typically, when decoding an Ambisonic soundfield, not many other processes are added to the encoded audio channels, although this may change as more Ambisonics software is released. There are already a number of plugins that include Ambisonic encoded soundfield compressors, EQs, delays, reverbs and visualizers which perform multichannel audio processing on the Ambisonic soundfield ^{45 46 47}.

⁴⁵ <https://plugins.iem.at/>

⁴⁶ <https://www.ssa-plugins.com/>

⁴⁷ <http://www.matthiaskronlachner.com/?p=2015>

Chapter 9

Conclusion

Immersive sound is used in many different applications ranging from cinema, home cinema, museum and art installations, theatre performance, virtual reality, augmented reality and browser-based media. The term immersive sound encompasses the many different techniques that are used to have listeners sonically immersed in these different environments. Currently, there are two main methods of immersing a listener in sound, either headphone-based listening or 3D speaker configurations. The work that has been done in this thesis is concerned with research in the latter, termed ‘3D immersive sound’.

As discussed in chapter 1, 3D immersive sound is distinct from surround sound. Surround sound requires that a sound engineer deliver audio content as a fixed number of audio channels that feed a corresponding set of speakers positioned in a predefined layout. 3D immersive sound incorporates the use of a renderer that algorithmically calculates the mixing levels required to localize audio tracks to any speaker configuration. This renderer is then required to playback audio tracks based on their position in a spatial scene defined by object-based audio (OBA) spatial metadata or scene-based audio (SBA) encoded audio formats.

The need for distributed processing becomes prevalent as these sound scenes become larger and more complex. One such distributed system is the ‘ImmerGo’ 3D immersive audio system and is what was selected for study and modification in this thesis project.

9.1 ImmerGo as a Framework for Distributed Endpoint Processing

ImmerGo is different from other distributed systems that make use of the “big box approach”. As discussed in chapters 1 and 2, it uses an endpoint distributed approach to spatial audio rendering. This is enabled by its client-server architecture, and the Ethernet AVB protocol it uses to interact with the endpoints.

The Ethernet AVB audio networking technology is used to stream the audio tracks contained within the DAW synchronously to each endpoint processor. As well as this, the AVDECC protocol defined in Ethernet AVB provides the ability for ImmerGo to update the endpoint processors’ mixers according to the mixer values calculated by the DBAP algorithm within the ImmerGo server.

Looking at the endpoint distributed solution given by ImmerGo, it was speculated that the DBAP calculations processed within the server would start to overload the host processor prior to setting the endpoint mixers if running simultaneously with other signal processing software such as a DAW. This prompted the research hypothesis questioning whether the DBAP algorithm calculations and mixing could be offloaded to the endpoints.

Furthermore, it was apparent that ImmerGo could use alternative spatial audio rendering algorithms because of its object-based representation of speakers and track positions.

9.2 Project Work

In the introduction to the thesis, the following hypothesis was proposed:

“The algorithmic processing required to deliver multichannel sound source localization can be successfully offloaded to processors that reside within speaker endpoints.”

In order to prove the hypothesis, there were two main goals for project work associated with this research:

1. Incorporate further 3D immersive audio rendering algorithms within ImmerGo.
2. Extend the distributed endpoint processing approach by offloading the rendering calculations for each of these algorithms to speaker endpoints, such that both the spatial audio rendering calculations and mixing occur at the endpoint processors.

As shown by the previous chapters, these goals have been attained. Object based audio (OBA) rendering algorithms DBAP, VBAP and WFS were implemented within ImmerGo, as well as the decoding of Ambisonics Scene based audio (SBA) at the speaker endpoints. The listening test evaluation of these algorithms has confirmed that the algorithms have been successfully implemented.

These goals have allowed the hypothesis of this project to be proved. Indeed, the algorithmic processing required by object based rendering calculations and audio mixing can have their processing offloaded onto the speaker endpoint processors. This was achieved in 3 successive steps of endpoint distribution,

1. First, the algorithm calculations were performed within the server and audio mixing within the endpoints XMOS processors.
2. Second, both calculations and audio mixing were implemented in the XMOS processors.
3. Lastly, the calculations were performed in the XMOS processors and audio mixing offloaded onto the SHARC chip.

9.2.1 Implementing the Spatial Audio Algorithms

As shown in this thesis, DBAP, VBAP and WFS all use an object-based representation of tracks and speakers to localize the audio to the speakers. Ambisonics uses a scene-based approach. In order to implement the rendering algorithms within ImmerGo, each new algorithm was created as a library that was incorporated within the ImmerGo source code.

As well as the additions of libraries within the ImmerGo server, modifications were done to the speaker endpoint processors to allow it to have more functions than just being able to mix the audio it received. Within the XMOS processor, a new processing task called the ‘Spatial Audio Renderer’ was introduced. This task provided the XMOS with a container that could respond to various commands from ImmerGo. The commands differ according to the level of distribution. Since the rendering calculations occur within the processor, the processor is required to receive object-based metadata for both the tracks and speakers, and to calculate mixer values according to a particular algorithm. As well as this, a control mechanism was designed that was able to control the SHARC processor from the SAR in order to enable the final level of endpoint distribution. As shown in chapter 4, the SHARC processing was disabled for the second step of distribution and then enabled at the third step. This required that two different versions of firmware were

made. For the second step of distribution, the I²S driver was used to pass the audio from the XMOS through the disabled SHARC to the speakers. For the third step of distribution, the TDM driver was used to route audio through the XMOS processor to the SHARC mixer and out to the speakers.

As shown in chapter 4, ImmerGo's implementation of the DBAP algorithm was the first to be offloaded to the XMOS processors. ImmerGo was modified to send its audio object definition models for speaker and tracks to the SAR. The SAR was then able to perform the DBAP calculations based on these models and control mixing matrix within the XMOS processor. This was further extended to step 3 of distribution where a new version of firmware was made. This version had the XMOS processor's mixer matrix removed and enabled the SHARC chip for audio mixing. The SHARC had its mixer matrix set by the SAR according to mixer values calculated by DBAP.

Further algorithm implementation started with VBAP, the focus of chapter 5. A VBAP library was created which exposed functions that perform the preprocessing required by the matrix calculations. In order to determine triplets of speaker faces around the sphere, the convex hull of vertices was to be solved. An external Javascript library was incorporated to perform the Quickhull algorithm on the speaker configuration to determine the speaker triplets. Provided a convex hull is found, ImmerGo is able to localize sources to an array mixing the audio at the endpoints according to the output from the VBAP calculations. In order to spread sources using VBAP, the MDAP algorithm was implemented within the server. MDAP creates copies of the virtual source and rotates them equally about the primary source being panned. VBAP is calculated on all of these sources and outputs of these calculations are summed together. Since the preprocessing calculations are able to be solved prior to panning, it was decided that for the second step of distribution, they should continue being calculated within the server and then sent to the endpoints. This meant that the convex hull and inverted loudspeaker matrices had to be packetized and sent to the endpoints. The audio mixing required by VBAP can be performed both within the XMOS and SHARC mixers.

As was described in chapter 5, the implementation of MDAP was not possible on the XMOS processor because of the time taken to both solve for the secondary panning sources and calculate the mixer values for each of the sources. MDAP was thus omitted from step 2 and 3 of distribution.

WFS, the second 'alternative' rendering algorithm was discussed and implemented in chapter 6. The WFS algorithm requires that speakers are placed closely together on a horizontal line, or circular formulation. This thesis dealt with speakers placed on a line. In order to perform WFS, the track source audio is required to be attenuated and delayed at each speaker. As such, the XMOS mixer matrix was modified to include delay lines for each speaker. Because of this, a different version of firmware had to be made for the first and second development cycles. For the first step of distribution, the mixer volumes and delay values were calculated using the library and transmitted to the SAR to update the mixer matrix. The second iteration had the WFS calculations implemented within the SAR. Upon receiving track updates, the SAR calculated attenuation and delay values which were used to mix the audio within the XMOS. The XMOS processor memory did not allow for many delay lines due to memory constraint and because of this only 8 track sources were able to be panned within the first and second step of distribution. The third iteration has the WFS calculations implemented within the SAR with the SHARC mixer matrix updated according to

these values. The SHARC was able to attenuate and delay all 32 tracks and was able to be implemented alongside the DBAP and VBAP algorithms within a single version of firmware.

Ambisonics, the last of the rendering algorithms to be implemented was the focus of chapter 7. The mixer matrix values required for decoding can be solved within ImmerGo and then pushed to the endpoints. The Ambisonics library was able to determine the decoder values required by each speaker according to various Ambisonics formats. These values could then be sent to the appropriate speaker endpoint and have their mixer matrices set. The Ambisonics multichannel sound field was able to be transmitted over Ethernet AVB audio streams and decoded appropriately in the speaker endpoints according to the decoder values. Thus, Ambisonics was able to be added alongside VBAP and DBAP with XMOS audio mixing. Ambisonics decoding was also able to be implemented using the SHARC matrix mixer and was incorporated into the firmware for the final step of distribution that housed DBAP, VBAP and WFS calculations with the SHARC matrix mixer.

Throughout this process of implementation, three different versions of firmware were constructed:

1. Version 1 – provides DBAP, VBAP calculations within the SAR and XMOS mixer matrix, whilst also providing the capability to mix the audio using the output from DBAP, VBAP and Ambisonics decoder calculations within the server. It is able to process 32 channel AVB streams.
2. Version 2 – Similar to version 1 but the XMOS mixer matrix is implemented with a delay line to provide a mixing capability for WFS. WFS calculations are implemented within the SAR and the SAR is also able to control the XMOS delay line mixer according to output from WFS calculations within the server. As discussed previously, it is only able to take 8 channel AVB streams.
3. Version 3 – Similar to version 1 except now the XMOS mixer matrix is disabled and SHARC mixer matrix is enabled. Allows for DBAP, VBAP and WFS calculations to be processed within the SAR and allows for Ambisonics decoder values to be used for mixing. It can take 32 channel AVB streams.

9.2.2 Results of the Implementation

Tests were created and results from these tests show how each version of these implementations behaves in the context of a real-world application. Along with listening tests, a series of processor time and load tests were conducted. The time tests were used to determine whether the calculations were able to meet the 8.3ms timing constraint imposed by the MTC synchronization intervals. The load tests were used to determine how the host processor performed when having the calculations done within the ImmerGo application versus having them done within the XMOS processors.

It was shown in these tests that of the object based rendering calculations, WFS was the most suited to the distributed endpoint processing approach. This is due to the fact that the other object-based algorithms, DBAP and VBAP, require large overheads to determine the speaker mix levels and these overheads increase as more speakers are added to the configuration.

DBAP requires that the entire speaker configuration is used to normalize each speaker volume when panning throughout the array to avoid unwanted dips and rises in volume and maintain a balanced sound when moving sources. This means that as the speaker configuration grew, so did the processing time

required to perform the calculations. The XMOS timing tests show that the DBAP calculation is not able to pan more than 8 sources simultaneously to more than 15 speakers. Considering that some immersive audio installations can have significantly more than 15 speakers, it has been shown that DBAP would not have an effective solution for this with the current hardware. However, should the speaker configuration contain less than 15 speakers, DBAP would be able to be performed at the endpoints whilst still providing localization for a number of sources simultaneously. Furthermore, it was shown by the host processor load tests that although DBAP was the least processor intensive of all the algorithms implemented, when used with other processing software such as a DAW this processor load increased. As such it would still be favourable to offload the calculations should the setup be small.

Whilst the VBAP matrix calculation is relatively inexpensive, it becomes more expensive as more speakers get added to the array. This is because the matrix calculation needs to be performed for each face in the convex hull to determine when the source position is rendered within a particular speaker triplet. Interestingly, it was shown that the VBAP algorithm calculated within the host was able to complete within the time selection, however, the CPU load exerted by the algorithm tended to slow down the operation for more than 8 sources to more than 40 speakers. The when moving the calculations to the endpoints, a single source was able to be panned to 100 speakers. However, if the number of sources panned simultaneously increased to 8, it was limited to 10 speakers. As such, it is proposed that an 'offline' version of the VBAP algorithm might be implemented to allow for 32 sources to 100 speakers. This entails calculating the VBAP gain values for a variety of different directions and saving the directions as keys to gains for each speaker a dictionary. Then, when a source is panned to a direction, the gain values may be recalled for that direction and sent to the processors.

WFS has the level and delay values of the loudspeaker calculated on a per speaker basis. As shown in the results, WFS was able to pan 32 track sources to an array of any size. This is because each endpoint processor was only required to calculate the values for speakers it was attached to. For 2 speakers, WFS was able to calculate these values well within the 8.3ms. This is a perfect example of the endpoint distribution approach working in practice. Now, as a user adds more speakers to the array, the processing power required to calculate the mixing values for those speakers is also added.

Ambisonics is an SBA technique which is different from the OBA techniques described above. When decoding an Ambisonic sound field, the mixer matrix for each endpoint need only be set once. As such, timing tests were omitted for the Ambisonics decoder, although it can be said that the time taken to set the mixer matrix once would be the same as when setting the mixer matrix according to a single mixer matrix update that would be used for step 1 of the distributed endpoint approach. CPU load tests were conducted to determine the amount of processing load that may be mitigated by offloading the decoder mixing to the endpoints. It was shown that as the speakers increase, then so did the processing load. For a 4th order decoding to 32 speaker array, the load put on by the DAW would increase by 12%. Furthermore, the trend in CPU load usage from 8 to 32 speakers suggests that the processing load may increase by double should the speaker array increase in size.

Another interesting feature with the offloading the Ambisonics decoder to speakers, is that media such as 360° videos are able to stream their encoded Ambisonic audio format to the speakers directly without any processing being performed on their computer.

9.3 The Novelty of Extending the Distributed Endpoint Approach

The extension implemented to the ImmerGo distributed endpoint approach in this project is able to provide a user with functionality that has not been possible in other 3D spatial audio systems, including ImmerGo. In particular,

1. The implementation of DBAP, VBAP, WFS and Ambisonics decoding spatial audio rendering algorithms within the endpoint XMOS processors.
2. The release of processing resources on the host computer for other signal processing applications.
3. A readymade Ambisonics decoder implemented within the endpoint that allows the speaker to directly play out an Ambisonics encoded audio format.
4. A novel use of the AVDECC protocol to transmit audio object metadata for both track objects and room environment objects.
5. The ability to increase the size of a WFS speaker configuration indefinitely by incrementally adding processor endpoints attached to speakers, thus avoiding the need to purchase expensive “big box” computers and audio interfaces.
6. A control mechanism for the XMOS processor to control the SHARC DSP chip.

9.4 Future Work

This thesis has opened further avenues of research in the 3D immersive audio field. By using XMOS microcontrollers, Ethernet AVB and a lightweight web-based application, this project has shown that it is viable to use the endpoint distributed approach successfully for multiple spatial audio algorithms.

Looking at a bigger picture, the final step of endpoint distribution shows great promise as it allows to have all four rendering algorithms implemented within the endpoints. This allows for both a combination and hybridization of these algorithms to be employed using different steps of distribution. For instance, mixer values may be calculated according to DBAP within the host server, and sent to a speaker configuration that is also performing WFS panning. The ImmerGo user interface could be extended to represent different types of speaker arrays that have associated with them different rendering algorithms that are able to be enabled depending on the proximity of track objects to each of the speakers.

As mentioned in chapter 6, the WFS implementation omits the synthesis of moving sources. In order to provide this, further modifications to the XMOS implementation would be required in order to provide a method to generate and interpolate between filters that are used to synthesize the movement of point sources across the loudspeaker array. This would require a more complex restructuring of the current implementation in this project and it would be of interest to see the limitations of the current hardware when implementing a complete WFS solution.

As well as this, further optimization may be performed within the endpoints used in this project. The XMOS firmware developed in this project could be structured in different ways to allow for a more efficient implementation of each of the algorithms. Of particular interest would be to,

1. Increase the number of AVB streams able to be streamed to the XMOS processor.
2. Optimize the implementation of DBAP to decrease the processor load exerted on the XMOS processor for large track and channel counts.
3. Implement an offline VBAP and MDAP panning system within ImmerGo to prevent the heavy processor loads elicited by calculating the values in real-time.
4. Combine the VBAP gain calculations and Ambisonic decoding into what is known as an AllRAD decoder which allows for more flexible loudspeaker arrangements.

References

- Ahrens, J. (2012). *Analytical Methods of Sound Field Synthesis*. Berlin Heidelberg: Springer.
- Ahrens, J., Geier, M., & Spors, S. (2010, January). Perceptual Comparison of Wave Field Synthesis Implemented with Integer and Fractional Delays.
doi:https://www.researchgate.net/publication/267781140_Perceptual_Comparison_of_Wave_Field_Synthesis_Implemented_With_Integer_And_Fractional_Delays
- Arteaga, D. (2018, June). Introduction to Ambisonics. Retrieved 10 10, 2018, from https://www.researchgate.net/profile/Daniel_Arteaga/publication/280010078_Introduction_to_Ambisonics/links/5b2786ed458515cad55f79c0/Introduction-to-Ambisonics.pdf
- Atkinson, K., & Han, W. (2012). *Spherical Harmonics and Approximations on the Unit Sphere: An Introduction*. Springer Science & Business Media.
- Barber, C. B., Dobkin, D. P., & Huhdanpaa, H. (1996). The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4), pp.469-483.
- Barco, A. T. (2015). *AuroMax Whitepaper*. Retrieved 12 23, 2017, from https://www.auro-3d.com/wp-content/uploads/documents/AuroMax_White_Paper_24112015.pdf
- Berkhout, A. J. (1988). A holographic approach to acoustic control. *Journal of the audio engineering society*, 36, 977-995.
- Berkhout, A. J., Vries, D., & Vogel, P. (1993). Acoustic control by wave field synthesis. *The Journal of the Acoustical Society of America*, 93, 2764-2778.
- Bleidt, R., Borsum, A., Fuchs, H., & Weiss, S. M. (2014). Object-Based Audio: Opportunities for Improved Listening Experience and Increased Listener Involvement. SMPTE 2014 Annual Technical Conference Exhibition.
- Blumlein, A. D. (1931). *U.K. Patent No. U.K. Patent 394 325*.
- Boone, M. M., Verheijen, E. N., & Van Tol, P. F. (1995). Spatial Sound-field Reproduction by Wave Field Synthesis. *Journal of the Audio Engineering Society*, 1003-1012.
- Bruno, R., Laborie, A., & Montoya, S. (2005). Reproducing Multichannel Sound on any Speaker Layout. Barcelona: Audio Engineering Society Convention 118.
- Choi, H. (2012). An alternative implementation of VBAP with graphical interface for sound motion design. Atlanta, Georgia: 18th International Conference on Auditory Display.
- Corteel, E., Kuhn-Rahloff, C., & Pellegrini, R. (2008). Wave Field Synthesis with Increased Aliasing Frequency. *124th Convention* (pp. 17-20). Amsterdam, The Netherlands: Audio Engineering Society.

- de Vries, D. (1995). Sound Enhancement by Wave Field Synthesis: Adaptation of the Synthesis Operator to the Loudspeaker Directivity Characteristics. *98th AES Convention*. Paris: Audio Engineering Society.
- de Vries, D. (1996). Sound Reinforcement by Wavefield Synthesis: Adaptation of the Synthesis Operator to the Loudspeaker Directivity Characteristics. *Journal Audio Engineering Society*, 44(12), 1120-1131.
- Dolby Laboratories. (2015, June). *Dolby AC-4: Audio Delivery for Next-Generation Entertainment Services*. Retrieved from Dolby AC-4 Audio System: <https://www.dolby.com/us/en/technologies/ac-4/Next-Generation-Entertainment-Services.pdf>
- DVB. (2018). Specification for the Use of Video and Audio Coding in Broadcast and Broadband Applications.
- ETSI. (2015). TS 103 190-2: Digital Audio Compression (AC-4) Standard Part 2: Immersive and personalized audio .
- ETSI, E. T. (2015). ETSI TS 103 223 MDA; Object-Based Audio Immersive Sound Metadata and Bitstream. European Telecommunications Standards Institute (ETSI).
- European Broadcast Union. (2018). Tech 3388: EBU Audio Definition Model Renderer (EAR). Geneva: European Broadcast Union.
- European Broadcasting Union. (2014). Tech 3364 Audio Definition Model Metadata Specification v1.0.
- Everest, F. A., & Pohlmann, K. (2014). *The Master Handbook of Acoustics* (Sixth Edition ed.). McGraw-Hill Education.
- Farina, A. (2017). *Ambisonics to TBE conversion*. Retrieved 10 19, 2018, from <http://pcfarina.eng.unipr.it/TBE-conversion.htm>
- Farina, A. (2017). *HOA Explicit Formulas*. Retrieved 10 19, 2018, from http://pcfarina.eng.unipr.it/Aurora/HOA_explicit_formulas.htm
- Firtha, G. (2018). *A Unified Wave Field Synthesis Framework with Application for Moving Virtual Sources*. Budapest: Budapest University of Technology and Economics.
- Foss, R., & Rouget, A. (2015). Immersive Audio Content Creation Using Mobile Devices and Ethernet AVB. *Audio Engineering Society Convention 139*.
- Furness, R. K. (1990). Ambisonics- An Overview. Washington D.C: Audio Engineering Society Conference: 8th International Conference: The Sound of Audio.
- Gerzon, M. A. (1973). Periphony: With-height Sound Reproduction. *Journal of the Audio Engineering Society*, 21(1), 2-10.
- Herre, J., Hilpert, J., Kuntz, A., & Plogsties, J. (2015). MPEG-H Audio—The New Standard for Universal Spatial/3D Audio Coding. *Journal of Audio Engineering Society*, 821-830.
- Hollerweger, F. (2013). An Introduction to Higher Order Ambisonic. *April 2005*.

- IEEE . (2011, 3). Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks. *IEEE Std 802.1AS-2011*, 1-292. doi:10.1109/IEEESTD.2011.5741898
- IEEE . (2011, 9). Standard for Local and metropolitan area networks--Audio Video Bridging (AVB) Systems. *IEEE Std 802.1BA-2011*, 1-45. doi:10.1109/IEEESTD.2011.6032690
- IEEE . (2014, 12). Standard for Local and metropolitan area networks--Bridges and Bridged Networks. *IEEE Std 802.1Q-2014 (Revision of IEEE Std 802.1Q-2011)*, 1-1832. doi:10.1109/IEEESTD.2014.6991462
- IEEE. (2013). Std 1722.1 (IEEE Standard for Device Discovery, Connection Management, and Control Protocol for IEEE 1722 Based Devices).
- IEEE. (2016). Std 1722 (IEEE Standard for a Transport Protocol for Time-Sensitive Applications in Bridged Local Area Networks) (revised).
- IEEE. (2018). IEEE Std 1722.1. *IEEE Standard for Device Discovery, Connection Management, and Control Protocol for IEEE 1722 Based Devices*. doi:https://standards.ieee.org/standard/1722_1-2013-Cor1-2018.html
- International Telecommunications Union. (2017). ITU-R BS.1770-4: Algorithms to measure audio programme loudness and true-peak audio level. Geneva.
- Ise, S. (1999). A principle of sound field control based on the Kirchhoff-Helmholtz integral equation and the theory of inverse systems. *Acta Acustica united with Acustica*, 85, 78-87.
- Kostadinov, D., Reiss, J. D., & Mladenov, V. (2010). Evaluation of distance based amplitude panning for spatial audio. *ICASSP*, (pp. 285-288).
- Litovsky, R. Y., Colburn, S. H., Yost, W. A., & Guzman, S. J. (1999). The Precedence Effect . *The Journal of the Acoustical Society of America*.
- Lopez, J., Bleda, S., Escolano, J., & Pueo, B. (2005). A Sub-band Approach to Wave-Field Synthesis Rendering. Barcelona, Spain: Audio Engineering Society.
- Lossius, T., Baltazar, P., & Hogue, T. (2009). *DBAP--distance-based amplitude panning*. Ann Arbor, MI: Michigan Publishing, University of Michigan Library.
- Malham, D. (2003). Higher Order Ambisonic Systems. York. Retrieved 09 10, 2018, from https://www.york.ac.uk/inst/mustech/3d_audio/higher_order_ambisonics.pdf
- Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., & Upton, M. (2002). Hyper-Threading Technology Architecture and Microarchitecture . *Intel Technology Journal Q1*.
- Monro, G. (2013). Spherical Harmonics and Ambisonics. *School of Mathematics and Statistics*, 7.
- Montag, M. N. (2011). *Wave Field Synthesis in Three Dimensions By Multiple Line Arrays*. University of Miama.

- Morrell, M. J., & Reiss, J. D. (2009). A Comparative Approach to Sound Localization Within a 3-D Sound Field. Audio Engineering Society Convention 126.
- MPEG-H. (2015). ISO/IEC 23008-3: High efficiency coding and media delivery in heterogeneous environments -- Part 3: 3D audio.
- Nachbar, C., Zotter, F., Deleflie, E., & Sontacchi, A. (2011). AmbiX - A Suggested Ambisonics Format. Lexington: Ambisonics Symposium.
- NodeJS. (n.d.). doi:<https://nodejs.org/en/>
- NWJS. (n.d.). NWJS (node-webkit). doi:<https://nwjs.io/>
- Oberstar, E. L. (2007). Fixed-point representation & fractional math. *Oberstar Consulting*. Retrieved 12 14, 2018, from <http://www.superkits.net/whitepapers/Fixed%20Point%20Representation%20&%20Fractional%20Math.pdf>
- Politis, A., & Pulkki, V. (2015). Vector Based Amplitude Panning in MATLAB. Greater Helsinki, Finland : Department of Signal Processing and Acoustics, Aalto University. Retrieved September 01, 2018, from <https://github.com/polarch/Vector-Base-Amplitude-Panning>
- Pulkki, V. (1997). Virtual Sound Source Positioning Using Vector Base Amplitude Panning. *J. Audio Eng. Soc*, 45, 456-466.
- Pulkki, V. (1999). Uniform Spreading of Amplitude Panned Virtual Sources. (pp. pp.187-190). Applications of Signal Processing to Audio and Acoustics IEEE.
- Pulkki, V., & Politis, A. (2015). *Vector Base Amplitude Panning Library for Matlab*. Retrieved 09 01, 2018, from <https://github.com/polarch/Vector-Base-AmplitudePanning>
- SADIE - Spatial Audio for Domestic Interactive Entertainment. (2019, February 04). *Ambisonics Decoders*. Retrieved from <https://www.york.ac.uk/sadie-project/ambidec.html>
- Shivappa, S., Morrell, M., Sen, D., Peters, N., & Salehin, S. M. (2016). Efficient, Compelling, and Immersive VR Audio Experience Using Scene Based Audio/Higher Order Ambisonics. *2016 AES International Conference on Audio for Virtual and Augmented Reality*. Los Angeles, CA, USA.
- SMPTE. (2018). ST 2098-1:2018 - SMPTE Standard - Immersive Audio Metadata. *IEEE*.
- SMPTE. (2018). ST 2098-2:2018 - SMPTE Standard - Immersive Audio Bitstream Specification. *IEEE*. doi:10.5594/SMPTE.ST2098-2.2018
- Spors, S., & Rabenstein, R. (2006). Spatial Aliasing Artifacts Produced by Linear and Circular Loudspeaker Arrays used for Wave Field Synthesis. Paris, France: Audio Engineering Society.
- Spors, S., Hahn, N., & Winter, F. (2017). Synthesis of a spatially band-limited plane wave in the time-domain using wave field synthesis. Kos, Greece: 25th European Signal Processing Conference (EUSIPCO).

- Spors, S., Rabenstein, R., & Ahrens, J. (2008). *The Theory of Wave Field Synthesis Revisited*. Amsterdam, Netherlands: Audio Engineering Society.
- Spors, S., Wierstorf, H., Geier, M., & Ahrens, J. (2009). *Physical and Perceptual Properties of Focused Sources in Wave Field Synthesis*. New York: Audio Engineering Society.
- Start, E. W. (1997). *Direct Sound Enhancement by Wave Field Synthesis*. Delft: Delft University of Technology.
- Strang, G. (2016). *Introduction to Linear Algebra* (Fifth Edition ed.). Wellesley: Wellesley-Cambridge Press.
- Technical Committee 100 - Digital system interfaces and protocols. (2005). IEC 61883-6 Consumer audio/video equipment - Digital interface - Part 6: Audio and music data transmission protocol. International Standard: International Electrotechnical Commission.
- Theile, G., Wittek, H., & Reisinger, M. (2003). *Potential Wavefield Synthesis Applications in the Multichannel Stereophonic World*. Munich, Germany: Audio Engineering Society.
- Trevino, J., Okamoto, T., Iwaya, Y., & Suzuki, Y. (2010). High order Ambisonic Decoding Method for Irregular Loudspeaker Arrays. Australia: Proceedings of the 20th International Congress on Acoustics.
- Verheijen, E. (2010). *Sound Reproduction by Wave Field Synthesis* (2nd ed.).
- Vetterli, M. K. (2009). Sound Field Reconstruction: An Improved Approach for Wave Field Synthesis. Audio Engineering Society Convention 126.
- Vogel, P. (1993). *Application of Wave Field Synthesis in Room Acoustics*. Delft, Netherlands: Delft University of Technology.
- Vries, D. D. (1996). Sound Reinforcement by Wavefield Synthesis: Adaptation of the Synthesis Operator to the Loudspeaker Directivity Characteristics. *Journal Audio Engineering Society*, 44(12), 1120-1131.
- Wiggins, B. (2007). The Generation of Panning Laws for Irregular Speaker Arrays Using Heuristic Methods. London: Audio Engineering Society Conference: 31st International Conference: New Directions in High Resolution Audio.
- Wittek, H. (2002). Optimised Phantom Source Imaging of the High Frequency Content of Virtual Sources in Wave Field Synthesis. Internal CARROUSO Paper. Retrieved from https://www.hauptmikrofon.de/HW/OPSI_Paper.pdf
- Zotter, F., & Frank, M. (2012). All-Round Ambisonic Panning and Decoding. *Journal of the Audio Engineering Society*, 60(10), 807-820.

Appendix A

A1 Solving the Covariance Factor

Before DBAP panning, the covariance factor for the spatial blur needed to be solved. This happened after the speaker configuration file was read into memory. The covariance factor is solved according to the function `getCovarianceFactor` shown in **Listing A1** below. First, the mean value of the loudspeaker coordinates is solved. This is then used to calculate the covariance factor according to equation (2.10). This factor will then be used when panning with DBAP.

```

367 function getCovarianceFactor(speakerConfig){
368     let mean_coords;
369     let covarianceFactor = 0;
370     let N = 0;
371     // First determine mean of the speaker configuration
372     mean_coords = mean(speakerConfig);
373
374     // Now determine covariance
375     for (let i = 0; i < speakerConfig.amplifiers.length; i++){
376         for (let j = 0; j < speakerConfig.amplifiers[i].speakers.length; j++){
377             let x = speakerConfig.amplifiers[i].speakers[j].x;
378             let y = speakerConfig.amplifiers[i].speakers[j].y;
379             let z = speakerConfig.amplifiers[i].speakers[j].z;
380
381             covarianceFactor += (x-mean_coords[0])*(x-mean_coords[0]);
382             covarianceFactor += (y-mean_coords[1])*(y-mean_coords[1]);
383             covarianceFactor += (z-mean_coords[2])*(z-mean_coords[2]);
384             N++;
385         }
386     }
387     covarianceFactor /= N-1;
388     covarianceFactor = Math.sqrt(covarianceFactor);
389
390     return covarianceFactor
391 }

```

Listing A1: Solving for the covariance factor.

A2 Sending Mixer Updates to Endpoints

The data comprises of a 64 4 bit chars with each char controlling mixer crosspoint. **Listing A2** shows the `sendMixLevels` function within the server. The template used to send the packets is given in **Listing A3**. This iterates through every endpoint processor and sends the corresponding mix levels for those mixers using the `sendPacketToAmp` function shown in **Listing A4**. The `sendPacketToAmp` function copies the packet template and replaces the headers with the correct MAC and GUID information for the server and endpoint processor addresses. From here the length of the mixer data is set and the control data length value updated within the packet. The mixer gain values are then built up and concatenated to the copied packet. This packet is then sent using pcap.

```

2233 // send mix levels to all amps
2234 var sendMixLevels = function() {
2235     // For each amplifier transmit mix levels for each speaker
2236     for (var i=0; i<speakerConfig.amplifiers.length; i++)
2237         sendPacketToAmp(i);
2238 }

```

Listing A2: `SendMixLevels` function which updates all the endpoint mixer parameters.


```

2160 // The Mixer gain Level packet layout
2161 let aecpMixerGainsPacket = new Buffer([
2162   0xff,0xff,0xff,0xff,0xff,0xff,           // destination MAC
2163   0x68,0x5b,0x35,0xd0,0xbb,0x55,         // source MAC NB! change for other servers
2164   0x22,0xf0,                               // ethertype 1722
2165   0xfb,                                     // control indicator true, AECp subtype
2166   0x00,                                     // AEM command
2167   0x00,0x40,                               // control data Length
2168   0x00,0x22,0x97,0xff,0xfe,0x00,0x41,0xc1, // target GUID
2169   0x68,0x5b,0x35,0xff,0xfe,0xd0,0xbb,0x55, // controller GUID
2170   0x00,0x00,                               // sequence number
2171   0x00,0x1e,                               // command type (SET_MIXER) (control is 18)
2172   0x00,0x1c,                               // descriptor type (MIXER) (control is 1a)
2173   0x00,0x01,                               // descriptor index (0x0001 for XMOS mixer gain control)
2174   | | | | | | | | | | | | | | | | | | | | // mix values 2x24 matrix
2175 ]);

```

Listing A3: The SET_MIXER packet structure within the ImmerGo source code.

```

2178 // function to send mix levels for all speakers connected to an amp
2179 let sendPacketToAmp = function(ampIndex) {
2180   let i = ampIndex;
2181   // Copy template packet
2182   let new_aecpMixerGainsPacket = aecpMixerGainsPacket;
2183
2184   //Set target mac address
2185   new_aecpMixerGainsPacket[0] = speakerConfig.amplifiers[i].macadr[0];
2186   new_aecpMixerGainsPacket[1] = speakerConfig.amplifiers[i].macadr[1];
2187   new_aecpMixerGainsPacket[2] = speakerConfig.amplifiers[i].macadr[2];
2188   new_aecpMixerGainsPacket[3] = speakerConfig.amplifiers[i].macadr[3];
2189   new_aecpMixerGainsPacket[4] = speakerConfig.amplifiers[i].macadr[4];
2190   new_aecpMixerGainsPacket[5] = speakerConfig.amplifiers[i].macadr[5];
2191
2192   // Set source MAC address
2193   mac.toBuffer(AVBNetworkMac, new_aecpMixerGainsPacket, 6); // copy mac address of server into connection request packet
2194
2195   //Set target GUID
2196   new_aecpMixerGainsPacket[18] = speakerConfig.amplifiers[i].macadr[0];
2197   new_aecpMixerGainsPacket[19] = speakerConfig.amplifiers[i].macadr[1];
2198   new_aecpMixerGainsPacket[20] = speakerConfig.amplifiers[i].macadr[2];
2199   new_aecpMixerGainsPacket[21] = 0xff;
2200   new_aecpMixerGainsPacket[22] = 0xfe;
2201   new_aecpMixerGainsPacket[23] = speakerConfig.amplifiers[i].macadr[3];
2202   new_aecpMixerGainsPacket[24] = speakerConfig.amplifiers[i].macadr[4];
2203   new_aecpMixerGainsPacket[25] = speakerConfig.amplifiers[i].macadr[5];
2204
2205   serverEntityID.copy(new_aecpMixerGainsPacket, 26); // server entity ID --> aecp packet controller field
2206
2207   let gainMixerLength = speakerConfig.amplifiers[i].speakers.length*noTracks; // Total length of data
2208   new_aecpMixerGainsPacket[17] = 16 + gainMixerLength; // Set control data Length of AVDECC packet
2209
2210   let mixLevelData = new Buffer(gainMixerLength);
2211
2212   // Build up mix Levels data packet
2213   let mixLevelIndex = 0;
2214   for (let j=0; j<speakerConfig.amplifiers[i].speakers.length; j++) {
2215     //Set mix Levels
2216     for (let k=0; k<noTracks; k++) { // Each speaker has mix Levels for each track
2217       mixLevelData[mixLevelIndex++] = speakerConfig.amplifiers[i].speakers[j].mixLevels[k];
2218     }
2219   }
2220
2221   // Concatenate buffer
2222   new_aecpMixerGainsPacket = Buffer.concat([new_aecpMixerGainsPacket,mixLevelData]);
2223   try {
2224     // send pcap packet
2225     c.send(new_aecpMixerGainsPacket,new_aecpMixerGainsPacket.length);
2226   } catch(e) {
2227     console.log("Error sending packet");
2228   }
2229 }

```

Listing A4: Sending AVDECC packets to an endpoint processor.

A3 Sending ImmerGo Room Definition Model

The room definition model is sent to all the endpoint devices held within the speaker configuration XML once they've been discovered and connected to on the network. **Listing A5** shows after the connections have been made the *sendSpkConfToXMOS* function is called which sends the speaker configuration to all the endpoints within the XML file.

```

629 |         if (gotAmplifiers && gotServer) {           // Have we got entity ID's of server and amplifiers?
630 |             discoveryComplete = true;
631 |             console.log("discovery done");
632 |             deleteConnections();
633 |             makeConnections();
634 |
635 |             sendSpkConfToXMOS();

```

Listing A5: *sendSpkConfToXMOS* called after all the endpoint devices in XML have been discovered and connected.

Listing A6 shows how the *sendSpkConfToXMOS* function flattens the speaker configuration model into a single array ready for sending. The entire speaker configuration is iterated through and added to an array. From here, the array is packetized into a buffer using the *packetizeSpkConf* function as shown in **Listing A7**. Each speaker configuration coordinate is given as a 32 bit integer byte and as such needs to be sent in 4 bit chunks. After all the speaker positions have been packed, the configuration is terminated with a 0xFFFFFFFF.

```

981 | function sendSpkConfToXMOS(){
982 |     // Get values of speaker configuration into array
983 |     let flatSpkConf = [];
984 |     for (let i = 0; i < speakerConfig.amplifiers.length; i++){
985 |         // Get speakers at amplifier
986 |         for (let j = 0; j < speakerConfig.amplifiers[i].speakers.length;j++){
987 |             let x = speakerConfig.amplifiers[i].speakers[j].x;
988 |             let y = speakerConfig.amplifiers[i].speakers[j].y;
989 |             let z = speakerConfig.amplifiers[i].speakers[j].z;
990 |             // get positions of each speaker and add to one configuration
991 |             flatSpkConf = flatSpkConf.concat([x,y,z]);
992 |         }
993 |     }
994 |
995 |     // Packetize speaker configuration
996 |     let spkConfpacketData = packetizeSpkConf(flatSpkConf);
997 |
998 |     // Send speaker configuration to each amp
999 |     for (let i = 0; i < speakerConfig.amplifiers.length; i++){
1000 |         sendSpkConfToAmp(i,spkConfpacketData);
1001 |     }
1002 | }

```

Listing A6: *sendSpkConfToXMOS* function.

```

915 function packetizeSpkConf(flatSpkConf){
916 // turns a flatten speaker configuration array into packet data
917 let packetData = new Buffer(flatSpkConf.length*4+4); // + 2 for terminating string
918 let i = 0;
919 let spkid = 0;
920 while(i < flatSpkConf.length){
921 // Turn coords into 32 bit signed int type
922 let val_x = parseInt(flatSpkConf[i]);
923 i++;
924 let val_y = parseInt(flatSpkConf[i]);
925 i++;
926 let val_z = parseInt(flatSpkConf[i]);
927 i++;
928
929 // Split coords into 4 8 bit nybbles
930 // x coord
931 let uux = (val_x >>> 24) & 0xff;
932 let ux = (val_x >>> 16) & 0xff;
933 let lx = (val_x >>> 8) & 0xff;
934 let llx = val_x & 0xff;
935
936 // y coord
937 let uuy = (val_y >>> 24) & 0xff;
938 let uy = (val_y >>> 16) & 0xff;
939 let ly = (val_y >>> 8) & 0xff;
940 let lly = val_y & 0xff;
941
942 // z coord
943 let uuz = (val_z >>> 24) & 0xff;
944 let uz = (val_z >>> 16) & 0xff;
945 let lz = (val_z >>> 8) & 0xff;
946 let llz = val_z & 0xff;
947
948 packetData[spkid*12] = uux;
949 packetData[spkid*12+1] = ux;
950 packetData[spkid*12+2] = lx;
951 packetData[spkid*12+3] = llx;
952
953 packetData[spkid*12+4] = uuy;
954 packetData[spkid*12+5] = uy;
955 packetData[spkid*12+6] = ly;
956 packetData[spkid*12+7] = lly;
957
958 packetData[spkid*12+8] = uuz;
959 packetData[spkid*12+9] = uz;
960 packetData[spkid*12+10] = lz;
961 packetData[spkid*12+11] = llz;
962
963 spkid++;
964 }
965
966 // End of speaker configuration termination int
967 let uterm = (term >>> 24) & 0xff;
968 let lterm = (term >>> 16) & 0xff;
969 let llterm = (term >>> 8) & 0xff;
970 let lllterm = term & 0xff;
971
972 packetData[spkid*12] = uterm;
973 packetData[spkid*12+1] = lterm;
974 packetData[spkid*12+2] = llterm;
975 packetData[spkid*12+3] = lllterm;
976
977 return packetData;
978 }

```

Listing A7: Packetizing the flattened speaker configuration.

Finally, the packetized speaker configuration is passed to the *sendSpkConfToAmp* function which sends it to each of endpoint processors. This function works similarly to the mixer matrix update function in the previous subsection.

It uses the template packet shown in Listing A8.

```

833 // The speaker config packet layout
834 let aecpSpkConfPacket = new Buffer([
835   0xff,0xff,0xff,0xff,0xff,0xff,           // destination MAC
836   0x68,0x5b,0x35,0xd0,0xbb,0x55,           // source MAC NB! change for other servers
837   0x22,0xf0,                                // ethertype 1722
838   0xfb,                                       // control indicator true, AECp subtype
839   0x00,                                       // AEM command
840   0x00,0x50,                                 // control data length
841   0x00,0x22,0x97,0xff,0xfe,0x00,0x41,0xc1,  // target GUID
842   0x68,0x5b,0x35,0xff,0xfe,0xd0,0xbb,0x55,  // controller GUID
843   0x00,0x00,                                // sequence number
844   0x00,0x1e,                                // command type (SET_MIXER) (control is 18)
845   0x00,0x1c,                                // descriptor type (MIXER) (control is 1a)
846   0x00,0x10,                                // descriptor index (0x0010)
847   0x00,0x00,                                // ID of speakers attached
848 ]]);

```

Listing A8: Speaker Configuration packet template.

As shown in **Listing A9**, within the *sendSpkConfToAmp* function, the template is copied and modified to send from the correct source MAC to the correct destination MAC for the endpoint processor. The control length value is changed according to the packet data size for the speaker configuration. The ID of speakers for each amp are iterated through and the template's first two data bytes are modified to hold those. Finally, the packetized speaker configuration is concatenated and to the packet and then sent with pcap.

```

850 // function to send mix levels for all speakers connected to an amp
851 let sendSpkConfToAmp = function(ampIndex,packetSpkConf) {
852   let new_aecpSpkConfPacket = aecpSpkConfPacket;
853
854   let i = ampIndex;
855   //Set target mac address
856   new_aecpSpkConfPacket[0] = speakerConfig.amplifiers[i].macadr[0];
857   new_aecpSpkConfPacket[1] = speakerConfig.amplifiers[i].macadr[1];
858   new_aecpSpkConfPacket[2] = speakerConfig.amplifiers[i].macadr[2];
859   new_aecpSpkConfPacket[3] = speakerConfig.amplifiers[i].macadr[3];
860   new_aecpSpkConfPacket[4] = speakerConfig.amplifiers[i].macadr[4];
861   new_aecpSpkConfPacket[5] = speakerConfig.amplifiers[i].macadr[5];
862
863   // Set source MAC address
864   mac.toBuffer(AVBNetworkMac, new_aecpSpkConfPacket, 6); // copy mac address of server into connection request packet
865
866   //Set target GUID
867   new_aecpSpkConfPacket[18] = speakerConfig.amplifiers[i].macadr[0];
868   new_aecpSpkConfPacket[19] = speakerConfig.amplifiers[i].macadr[1];
869   new_aecpSpkConfPacket[20] = speakerConfig.amplifiers[i].macadr[2];
870   new_aecpSpkConfPacket[21] = 0xff;
871   new_aecpSpkConfPacket[22] = 0xfe;
872   new_aecpSpkConfPacket[23] = speakerConfig.amplifiers[i].macadr[3];
873   new_aecpSpkConfPacket[24] = speakerConfig.amplifiers[i].macadr[4];
874   new_aecpSpkConfPacket[25] = speakerConfig.amplifiers[i].macadr[5];
875
876   serverEntityID.copy(new_aecpSpkConfPacket, 26); // server entity ID --> aecp packet controller field

```

```

877
878 // Determine speaker configuration length for packet
879 let amountSpksAttached = 2; // SPK4P is 2
880 let SpkConfLength = amountSpksAttached + speakerConfig.totalSpks*3*4 + 4; // each speaker has 3 coords associated with it.
881 let SpkConfstartindx = 42;
882
883 // Enter length of values into packet
884 new_aecpSpkConfPacket[17] = 16 + SpkConfLength;
885
886 let id_spks = []; // holds id of speakers in speaker config attached to current XMOS
887
888 for (let j = 0; j < amountSpksAttached;j++){
889   // Add IDs for speakers attached to this amp
890   id_spk = i*amountSpksAttached+j;
891   new_aecpSpkConfPacket[SpkConfstartindx+j] = id_spk;
892   // Add those ids to array
893   id_spks = id_spks.concat([id_spk]);
894 }
895
896 // Then append speaker configuration packet data
897 new_aecpSpkConfPacket = Buffer.concat([new_aecpSpkConfPacket,packetSpkConf]);
898
899 try {
900   // send pcap packet
901   c.send(new_aecpSpkConfPacket,new_aecpSpkConfPacket.length);
902   console.log("speaker configuration sent")
903 } catch(e) {
904   console.log("Error sending packet");
905 }
906 };

```

Listing A9: *sendSpkConfToXMOSToAmp* function.

A4 Sending ImmerGo Track Object Model

Listing A10 shows the array that holds all the tracks to be updated. This array is filled provided it is not there already when the *updateSpeakerMixLevels* function is called as shown in Listing A11.

```

1137 // structure for all tracks that will be sent on MTC frame
1138 var tracksforUpdate = [];

```

Listing A10: The tracks for update array.

```

2239 // Add tracks to updated
2240 if (!tracksforUpdate.includes(trackNo))
2241   | tracksforUpdate.push(trackNo)

```

Listing A11: Adding track IDs for updating.

These are then sent similarly to when updating the speaker endpoint mixer matrices. This is performed within the *sendSrcUpdate* function as shown in Listing A12.

```

1337 function sendSrcUpdate(){
1338   // Send src values to each amp
1339   if (tracksforUpdate.length > 0){
1340     let srcPacketData = packetizeSrcData(tracksforUpdate.length);
1341     for (let i = 0; i < speakerConfig.amplifiers.length; i++){
1342       // if no tracks updated don't send packet
1343       sendSrcToAmp(srcPacketData,i);
1344     }
1345   }
1346   // Clear the tracks for update struct so next frame is clean
1347   tracksforUpdate = [];
1348 }

```

Listing A12: Sending source updates to the speaker endpoints.

If there are elements in the *tracksforUpdate* array, the sources are packetized into data using the *packetizeSrcData* function shown in Listing A13. The number of sources to be updated are parsed and the packet data length is determined. From there a new buffer object is created and filled according to the number of sources. The track number and volume of the track is set to be added to the packet. Similar to

sending the speaker configuration coordinates, each source coordinate element needs to be sent as a 4 bit byte. Once broken into bits all the track object model information is added to the packet. This happens for each track object and then is passed back as a buffer that is then sent to each endpoint processor using the *sendSrcToAmp* function with the return packet data as an argument.

```

1156 let packetizeSrcData = function(noSrcs){
1157   // First set the length of the values
1158   // Determine length src for packet
1159   let noSrcs = noSrcs;
1160
1161   let srcPacketLength = 1+noSrcs*(2+3*4); // (1 for number of sources) + (each source has id + vol + (coordinates 3*(4*bytes)) )
1162
1163   // build values field for for srcPacket
1164   let srcPacketdata = new Buffer(srcPacketLength)
1165   srcPacketdata[0] = noSrcs;
1166
1167   // For each track id, split it's coords into 4 bytes and add to buffer with it's id
1168   for (let i = 0; i<noSrcs;i++){
1169     let id = tracksforUpdate[i];
1170     // volume of track
1171     let vol = tracks3D[id].volume;
1172
1173
1174     let x_r = parseInt(Math.round(tracks3D[id].currentxCoord));
1175     let y_r = parseInt(Math.round(tracks3D[id].currentyCoord));
1176     let z_r = parseInt(Math.round(tracks3D[id].currentzCoord));
1177
1178     // Split src coords into 4 8 bit words
1179     // x coord
1180     let uux = (x_r >>> 24) & 0xff;
1181     let ux = (x_r >>> 16) & 0xff;
1182     let lx = (x_r >>> 8) & 0xff;
1183     let llx = x_r & 0xff;
1184     // y coord
1185     let uuy = (y_r >>> 24) & 0xff;
1186     let uy = (y_r >>> 16) & 0xff;
1187     let ly = (y_r >>> 8) & 0xff;
1188     let lly = y_r & 0xff;
1189     // z coord
1190     let uuz = (z_r >>> 24) & 0xff;
1191     let uz = (z_r >>> 16) & 0xff;
1192     let lz = (z_r >>> 8) & 0xff;
1193     let llz = z_r & 0xff;
1194
1195     // Add to packet data
1196     srcPacketdata[1+14*i] = id;
1197     srcPacketdata[1+14*i+1] = vol;
1198     srcPacketdata[1+14*i+2] = uux;
1199     srcPacketdata[1+14*i+3] = ux;
1200     srcPacketdata[1+14*i+4] = lx;
1201     srcPacketdata[1+14*i+5] = llx;
1202     srcPacketdata[1+14*i+6] = uuy;
1203     srcPacketdata[1+14*i+7] = uy;
1204     srcPacketdata[1+14*i+8] = ly;
1205     srcPacketdata[1+14*i+9] = lly;
1206     srcPacketdata[1+14*i+10] = uuz;
1207     srcPacketdata[1+14*i+11] = uz;
1208     srcPacketdata[1+14*i+12] = lz;
1209     srcPacketdata[1+14*i+13] = llz;
1210   }
1211   return srcPacketdata;
1212 }

```

Listing A13: Creating source packet data.

The *sendSrcToAmp* uses the source update packet template shown in **Listing A14** with the respective descriptor index for the algorithm that intends to be used to solve for the audio mixer values.

```

1140 // The src packet layout
1141 var aecpSrcPacket = new Buffer([
1142     0xff,0xff,0xff,0xff,0xff,0xff,           // destination MAC
1143     0x68,0x5b,0x35,0xd0,0xbb,0x55,         // source MAC NB! change for other servers
1144     0x22,0xf0,                               // ethernet 1722
1145     0xfb,                                     // control indicator true, AECp subtype
1146     0x00,                                     // AEM command
1147     0x00,0x50,                               // control data length of 80 (16 + variable depending on no of srcs)
1148     0x00,0x22,0x97,0xff,0xfe,0x00,0x41,0xc1, // target GUID
1149     0x68,0x5b,0x35,0xff,0xfe,0xd0,0xbb,0x55, // controller GUID
1150     0x00,0x00,                               // sequence number
1151     0x00,0x1e,                               // command type (SET_MIXER) (control is 18)
1152     0x00,0x1c,                               // descriptor type (MIXER) (control is 1a)
1153     0x00,0x20,                               // descriptor index 0x0020 for src update DBAP
1154 ]);

```

Listing A14: The source packet template.

```

1214 // function to send src positions for all speakers connected to an amp
1215 let sendSrcToAmp = function(srcPacket_Data,ampIndex) {
1216     let i = ampIndex;
1217     let new_aecpSrcPacket = aecpSrcPacket;
1218     //Set target mac address
1219     new_aecpSrcPacket[0] = speakerConfig.amplifiers[i].macadr[0];
1220     new_aecpSrcPacket[1] = speakerConfig.amplifiers[i].macadr[1];
1221     new_aecpSrcPacket[2] = speakerConfig.amplifiers[i].macadr[2];
1222     new_aecpSrcPacket[3] = speakerConfig.amplifiers[i].macadr[3];
1223     new_aecpSrcPacket[4] = speakerConfig.amplifiers[i].macadr[4];
1224     new_aecpSrcPacket[5] = speakerConfig.amplifiers[i].macadr[5];
1225
1226     // Set source MAC address
1227     mac.toBuffer(AVBNetworkMac, new_aecpSrcPacket, 6); // copy mac address of server into connection request packet
1228
1229     //Set target GUID
1230     new_aecpSrcPacket[18] = speakerConfig.amplifiers[i].macadr[0];
1231     new_aecpSrcPacket[19] = speakerConfig.amplifiers[i].macadr[1];
1232     new_aecpSrcPacket[20] = speakerConfig.amplifiers[i].macadr[2];
1233     new_aecpSrcPacket[21] = 0xff;
1234     new_aecpSrcPacket[22] = 0xfe;
1235     new_aecpSrcPacket[23] = speakerConfig.amplifiers[i].macadr[3];
1236     new_aecpSrcPacket[24] = speakerConfig.amplifiers[i].macadr[4];
1237     new_aecpSrcPacket[25] = speakerConfig.amplifiers[i].macadr[5];
1238
1239     serverEntityID.copy(new_aecpSrcPacket, 26); // server entity ID --> aecp packet controller field
1240
1241     new_aecpSrcPacket[17] = 16 + srcPacket_Data.length
1242     new_aecpSrcPacket = Buffer.concat([new_aecpSrcPacket,srcPacket_Data]);
1243
1244     try {
1245         // send pcap packet
1246         c.send(new_aecpSrcPacket,new_aecpSrcPacket.length);
1247     } catch(e) {
1248         console.log("Error sending packet");
1249     }
1250 }

```

Listing A15: Sending the source packet.

The `sendSrcToAmp` function changes the AECp headers to contain the correct destination and source addresses. It then updates the control data length parameters and concatenates the source packet data to the function. It then sends the packet to the endpoints using the cap package.

A5 Unpacking the Speaker Configuration

There are two parts when sending the speaker configuration to the endpoints. As shown in Listing A16, first is to update the SPK-4P speaker ids so mixer values from the spatial audio rendering algorithms go to the correct speaker. Then each speaker location needs to be added to the speaker configuration array. The speaker configuration is limited to 100 speakers. The speaker coordinates are split into 4 8 bit words. Each coordinate element is unpacked within the for-loop. There are then combined into one integer value coordinate. There is a loop termination if the coordinate received is 0xFFFFFFFF which means it has reached the end of the speaker configuration.

```

154     case i_1722_l_control.set_speakerConfig(unsigned char ids[],char config[]) :{
155         /* Unpack ImmerGo Speaker Configuration */
156         // First set the speaker ids attached to this processor
157         ids_nanoSPKs[0] = ids[0];
158         ids_nanoSPKs[1] = ids[1];
159
160         /* Unpack speaker configuration from packet and put into struct */
161         indx = 0;
162         for (int i=0; i<MAX_SIZE_CONFIG*3;i++){
163             unsigned char u_upper_byte = config[indx++];
164             unsigned char upper_byte = config[indx++];
165             unsigned char lower_byte = config[indx++];
166             unsigned char l_lower_byte = config[indx++];
167             // Convert to int
168             int coord = (int) ((u_upper_byte << 24) | (upper_byte << 16) | (lower_byte << 8) | (l_lower_byte));
169
170             if (coord == -2147483648){
171                 speakerConfig[i] = -2147483648;
172                 break;
173             }
174             // Add entry to config
175             speakerConfig[i] = (double) coord;
176         }
177         /* Clear fifoMixVals */
178         memset(fifoMixVals, 0, sizeof fifoMixVals);
179         break;
180     }

```

Listing A16: The server interface that receives the speaker configuration into the endpoints.

A6 Unpacking the Track Object Model

Each object-based rendering algorithm required locations of the source and speakers to be sent to the endpoint processors. **Listing A17** shows how a source is unpacked before being used by a particular spatial audio algorithm. First, the track ID is unpacked which is then used for subsequent updates to the correct parts of the array. The volume of that track is then updated. From there the source location is unpacked into a 32 bit integer which is then cast into a double and saved to the source array before being processed through the algorithm. If more than one track is sent at once, this process is repeated for subsequent sources until all of them have been solved for. The algorithm then updates the gain structures and the mixer matrices are sent their parameters.

```

318     case i_1722_l_control.perform_DBAP(unsigned char noSrcs, char src[]) :{
319         /* Unpack ImmerGo Track Object Model */
320         indx = 0; // Reset index counter
321         // Source bundle updates
322         for (int j=0;j<noSrcs;j++){
323             unsigned char id_src = src[indx++]; // The track number of source update
324             volumes[id_src] = src[indx++]; // Volume of source update
325
326             // Add source position (x,y,z)
327             #pragma loop unroll
328             for (int i=0; i<3;i++){
329                 unsigned char u_upper_byte = src[indx++];
330                 unsigned char upper_byte = src[indx++];
331                 unsigned char lower_byte = src[indx++];
332                 unsigned char l_lower_byte = src[indx++];
333
334                 // Add to sources array
335                 srcs[id_src][i] = (double)((int) ((u_upper_byte << 24) | (upper_byte << 16) | (lower_byte << 8) | (l_lower_byte)))
336             }
337             // Perform algorithm...

```

Listing A17: Unpacking a track object before rendering.

A7 Updating the SHARC Mixer Matrix Crosspoints

The SHARC mixer matrix is addressed using indices 0-64, with 0-31 being the first column cross points and 32-63 being the second column cross points. A matrix cross point's gain and delay parameters are set using a single 32-bit word split into two parts. The 16 most significant bits are the gain levels of the samples and the 16 least significant bits is the amount of delay. **Listing A18** shows the procedure to set a single matrix cross point.


```

173//Gain: 16bits range 1 to -1, 1/15 per steps, 2's complement (Q1.15)
174//Delay: 16bits range 0 to WFS_MAX_DELAY samples
175unsigned char DSP_SetWFS_CrossPoint(unsigned int input, unsigned int output, unsigned short Gain, unsigned short Delay)
176{
177    unsigned int Tx_Buff[MAX_SPI_MSG_DWORD];
178    unsigned int DSP_Address;
179    unsigned int temp;
180
181    DSP_Address = input*WFS_NUM_OUT+output;
182
183    temp = Gain<<16;
184    temp |= (Delay & 0xFFFF);
185
186    Tx_Buff[0] = temp;
187
188    return DSP_AddrParaLoad(DSP_Address, (unsigned int*)Tx_Buff, 1);
189}

```

Listing A18: Setting a single matrix crosspoint.

A 32-bit variable holds the values to be sent to the SHARC. The gain is set and shifted left 16 bits and the least significant 16 bits are given the delay value. This is then loaded into the *DSP_AddrParaLoad* function with a message count of 1 since only 1 cross point is being addressed.

When a source is moved multiple crosspoints need to be updated at once. For instance, when a single input source gain and delay values are solved, it is required the mixer cross points to be updated at the same time. For this reason, a source update function is made that sets the cross points for that source as a group. As shown in **Listing A19**, this function takes the source base address and pointers to the gain and delay structure for the source. These parameters are loaded into the parameter buffer with special care to convert the gain values to Q1.15 and to truncate the number of samples to be delayed. Every time parameters are added to the SHARC, the parameter counter, *msg_count*. This counter along with a pointer to the parameter buffer is then sent to the SHARC.

```

222void load_srcGainsDelays2SHARC(unsigned int address, double* gains_fs, double* delays){
223    unsigned int Tx_Buff[MAX_SPI_MSG_DWORD]; //Sync+Header+CRC = 3 DWORD
224    unsigned int DSP_Address;
225    unsigned int temp;
226    unsigned int msg_count;
227    DSP_Address = address; //start from src address
228
229    msg_count = 0;
230
231    for(int j=0; j<WFS_NUM_OUT; j++){
232        temp = fs_to_Q115(*gains_fs++) << 16; // Add gain parameter
233        temp |= (trunc(*delays++)) & 0xFFFF; // Add delay parameter
234        Tx_Buff[msg_count++] = temp; // Add parameters to send buffer
235
236        // If reach maximum values sent over bus, split into two talker buffers and send remaining parameters
237        if(msg_count == ((MAX_SPI_MSG_DWORD) - 4)) //Sync+Header+Idx+CRC = 4 DWORD
238        {
239            DSP_AddrParaLoad(DSP_Address, (unsigned int*)Tx_Buff, msg_count);
240            DSP_Address += msg_count;
241            msg_count = 0;
242        }
243    }
244
245    if(msg_count) // Send any remaining parameters
246    {
247        DSP_AddrParaLoad(DSP_Address, (unsigned int*)Tx_Buff, msg_count); // Send parameters to SHARC
248    }
249}
250}

```

Listing A19: Function to set matrix crosspoints for a single source.

The IIR and amplitude smoothing filter coefficients are addressed at the very end of the address space. For the SPK-4P, they are addressed at 64 and 65. The coefficients are sent to the SHARC as floating point values using the *DSP_AddrParaLoad*. The spatial audio renderer is able to send updates of the filter coefficients using the *load_filter2SHARC* shown in **Listing A20**.

```

418 void load_filter2SHARC(float* filter_coeff){
419     //Load filter coefficients as float into SHARC
420     unsigned int DSP_Address;
421     unsigned int msg_count;
422
423     DSP_Address = MAX_CHANNELS*MAX_SPEAKERS;           //start from MAX_CHANNELS*MAX_SPEAKERS
424
425     msg_count = 2;
426     DSP_AddrParaLoad(DSP_Address, (unsigned int*)filter_coeff, msg_count);
427 }

```

Listing A20: Function to load filter coefficients

The *DSP_AddrParaLoad* function is defined in **Listing A21** below. The arguments are used to specify the address of the matrix cross point, a pointer to the array of parameters being loaded and the number of parameters being loaded. These are then sent to the SHARC over the SPI talker bus.

```

157 unsigned char DSP_AddrParaLoad(unsigned int DSP_Address, unsigned int* para, unsigned int para_cnt)
158 {
159
160     if(!SHARC_SPI_Tx((unsigned char *) &para[0], para_cnt*4, (AMFCommand_Set_Para&0xFF), DSP_Address))
161     {
162         //error, can't send to DSP
163         return 0;
164     }
165
166     MTimerWait(5);
167
168     return 1;
169 }

```

Listing A21: The *DSP_AddrParaLoad* function.

Appendix B

B1 Solving for the VBAP configuration

Before the server is able to perform the VBAP calculation, it is determined whether the configuration lies on a sphere. This is done by determining the radius of each speaker to the origin as shown in **Listing B1**.

```

405 let spk_pos = [];
406 let radii = [];
407 // Get positions of speakers
408 for (let i = 0; i < speakerConfig.amplifiers.length; i++){
409   for (let j = 0; j < speakerConfig.amplifiers[i].speakers.length; j++){
410     let x = speakerConfig.amplifiers[i].speakers[j].x;
411     let y = speakerConfig.amplifiers[i].speakers[j].y;
412     let z = speakerConfig.amplifiers[i].speakers[j].z;
413
414     radii.push(Math.round(Math.sqrt(Math.pow(x,2)+Math.pow(y,2)+Math.pow(z,2)))); // Get distance to centre
415     spk_pos.push([x,y,z]) // Add to the speaker positions
416   }
417 }

```

Listing B1: Determining the radius of each speaker in the ImmerGo server.

As shown in **Listing B2**, the distance of each radius is then checked to be within the bounds of 5cm from each other using the *checkAllValuesWithinBounds* function shown in **Listing B3**.

```

419 // Are the speakers on the perimeter of a sphere?
420 if(checkAllValuesWithinBounds(radii) !== true){
421   console.log("VBAP disabled. Speakers not on radius of sphere.");
422 }
423 else {
424   // Set up the triangle mesh
425   try{
426     vbap.SetMesh(spk_pos)
427     vbap_enabled = 1;
428     console.log("VBAP speaker mesh set");
429   } catch(e)
430   {
431     console.log("Unable to set VBAP loudspeaker mesh.");
432     console.log(e)
433     console.log("VBAP disabled")
434     vbap_enabled = 0;
435   }
436 }

```

Listing B2: Calling *SetMesh* from the VBAP library.

If this is not the case, then VBAP will be disabled. Provided they are within bounds, the VBAP configuration is set using the module function *SetMesh* called with an array holding the speaker positions.

```

392 // Checks whether speakers are on sphere
393 function checkAllValuesWithinBounds(radii){
394   var length = radii.length;
395   for(var i = 1; i < length; i++){
396     {
397       if(radii[i] > radii[0] + 5 || radii[i] < radii[0] - 5){
398         return false
399       }
400     }
401   }
402   return true;

```

Listing B3: Checking each radius is within 5cm within the *checkAllValuesWithinBound* function.

SetMesh performs the convex hull algorithm, discards all triangles that are either too big or too small and then determines the inverse loudspeaker matrices for the remaining triangles as shown in **Listing B4**. First,

the convex hull is determined using *FindLsTriplets* called with the loudspeaker positions. The *OMIT_LARGE_TRIANGLES* variable is set to true and there is no aperture limit set. The inverse loudspeaker matrices are then solved using the convex hull array.

```

436 function SetMesh(ls_positions){
437 /* Sets the correct loudspeaker mesh and calculates inverse loudspeaker matrix */
438
439 // Find valid loudspeaker groups
440 ls_mesh = FindLsTriplets(ls_positions, OMIT_LARGE_TRI,0);
441
442 // Find inverse loudspeaker matrix
443 inv_ls_mtx = invertLsMatrix(ls_mesh);
444 }

```

Listing B4: The *SetMesh* function.

Listing B5 shows the *FindLsTriplets* function which determines the convex hull used for the VBAP calculation. First, the loudspeaker vertices are converted into normalized vectors. From there, the triangulation process begins by determining the convex hull faces using Quickhull3d. The faces are then sorted from smallest to largest with the first element being the smallest in the face. From there triangles that are incorrectly orientated are discarded using the *keepValidTriangles* function.

```

191 function FindLsTriplets(ls_pos, OMIT_LARGE_TRI, aperture_lim){
192 /* Find speaker triplets. Returns speakermesh with faces and vertices of each speaker */
193 let speakermesh;
194 // Turn loudspeaker positions into unit vectors
195 let vertices = ls_pos.map(pos => {
196   let dist = math.sqrt(pos[0]*pos[0]+pos[1]*pos[1]+pos[2]*pos[2]);
197   return [pos[0]/dist,pos[1]/dist,pos[2]/dist];
198 });
199 /* Triangulation process: */
200 /* Step 1: Find triangle faces of convex-hull of loudspeakers */
201 let faces = quickhull(vertices); // Determine Convex hull (using quickhull3d)
202
203 faces = faces.map(sortVerticesInFace); // Shift results to begin each triangle from smallest entry
204
205 faces = faces.sort(sortByColumn); // Sort by first column, then second column
206
207 /* Step 2: Discard invalid triangles (ones with area too narrow) */
208 faces = keepValidTriangles(faces,vertices);
209
210
211 speakermesh = new LsMesh(faces,vertices); // Return convex hull mesh
212 return speakermesh;
213 }
214 }

```

Listing B5: The *FindLsTriplets* function.

As shown in **Listing 23**, the *keepValidTriangles* function iterates over each triangle faces and determines the angle between the surface normal of the triangle to the centroid. If the angle is greater than $\frac{\pi}{2}$ then the faces are orientated incorrectly and thus discarded. All triangles whose angles are less than $\frac{\pi}{2}$ are saved and returned by the function. The loudspeaker mesh is then set with these faces in the *FindLsTriplets* function.

```

165 function keepValidTriangles(faces,vert){
166   /* Function that discards invalid triangles */
167   let validfaces = [];
168   let n = faces.length;
169
170   for ( let i = 0 ; i < n; i++){
171     let currentface = faces[i];
172
173     let vec = [vert[currentface[0]], vert[currentface[1]], vert[currentface[2]]]; // Current face vectors
174     let triangle_normal = math.cross(math.subtract(vec[0], vec[1]), math.subtract(vec[1], vec[2])); // Triangle surface normal
175
176     let centroid = math.mean(vec, 0); // Centroid of triangle vectors
177
178     if (math.acos(math.dot(centroid,triangle_normal)) < Math.PI/2 ){
179       validfaces.push(faces[i]); // If orientated correctly add to valid triangles
180     }
181   }
182   return validfaces;
183 }

```

Listing B6: Determining which convex hull triplets to discard.

After the convex hull is determined the inverse loudspeaker matrices are determined. This happens within the *invertlsMatrix* function shown in **Listing B7**. For each of the faces in the convex hull, the inverse loudspeaker matrix for that face is determined. Once complete, the array holding all the inverted matrices is returned.

```

216 function invertlsMatrix(ls_mesh){
217   /* Find Inverse Loudspeaker Matrix */
218   let no_faces = ls_mesh.faces.length;
219   let layoutInvMtx = [];
220
221   for (let i = 0; i < no_faces; i++){
222     /* Get unit vectors for current group */
223     let currentface = ls_mesh.faces[i];
224     let tempGroup = [ls_mesh.vertices[currentface[0]], ls_mesh.vertices[currentface[1]], ls_mesh.vertices[currentface[2]]];
225
226     /* Get Inverse of current group */
227     let tempInv = math.divide(math.eye(3), tempGroup);
228
229     /* Push onto InvMatrix */
230     let flat = math.flatten(math.transpose(tempInv));
231     flat = flat.toArray();
232
233     layoutInvMtx.push(flat);
234   }
235
236   return layoutInvMtx;
237 }

```

Listing B7: Determining convex hull inverted loudspeaker matrices.

B2 Squeezing Vectors for MDAP

To squeeze secondary spread sources for required for MDAP, the *getSpreadSrcDirs* function is shown in **Listing B8**. These sources are then “squeezed” to the spread angle by multiplying the perpendicular vectors by the tangent of the spread angle and then adding to the original vector as shown in line 311. After this, they are normalized, concatenated with the primary vector and returned from the function.

```

257 function getSpreadSrcDirs(src_vector, spread){
258   /* Generates directions for spreading a source.
259     src_vector - source unit vector.
260     spread - spread angle of sources.
261
262     Returns direction matrix as vectors */
263
264   /* Number of spread sources and concentric rings */
265   let num_src = 8;
266   let num_rings_3d = 1;
267
268   let u = src_vector;
269
270   /* rotation matrix using axis of rotation-angle definition (around source direction) */
271   let u_x_u = [
272     [math.pow(u[0],2), u[0]*u[1], u[0]*u[2]],
273     [u[0]*u[1], math.pow(u[1],2), u[1]*u[2]],
274     [u[0]*u[2], u[1]*u[2], math.pow(u[2],2)]
275   ];
276   let u_x = [
277     [0, -u[2], u[1]],
278     [u[2], 0, -u[0]],
279     [-u[1], u[0], 0]
280   ];
281
282   /* Determine angle of rotation between each secondary source */
283   let theta = math.divide(2*Math.PI,num_src).toFixed(4);
284   let a = math.dotMultiply(math.eye(3),math.cos(theta).toFixed(4)); // cos(theta)*Id
285   let b = math.dotMultiply(math.sin(theta).toFixed(4),u_x) ; // sin(theta)*u_x
286   let c = math.dotMultiply((1-math.cos(theta).toFixed(4)),u_x_u); // (1 - cos(theta))*u_x_u
287   let R_theta = math.add(math.add(a,b),c).toArray(); // Rotation by theta matrix
288
289   /* Create a ring of sources on the plane that is perpendicular to the source */
290   let spreadbase = math.zeros(num_src,3).toArray() // All the secondary sources
291   let elev = math.asin(src_vector[2])*(180/Math.PI); // Determine angle of elevation of unit vector
292
293   /* Add first source */
294   if(elev > 89 || elev < -89){
295     | spreadbase[0] = [1, 0 ,0]; // Limit angles that are orthogonal
296   }else{
297     | spreadbase[0] = cross_prod_scaled(u,[0, 0, 1]); // Determine normal of vector to
298   }
299
300   /* Get rings of directions by rotating first vector around the source */
301   for(let ns = 1; ns<num_src;ns++){
302     | spreadbase[ns] = math.multiply(R_theta,math.transpose(spreadbase[ns-1]))
303   }
304

```

```

305  /* Squeeze the perpendicular ring to desired spread */
306  let spread_rad = (spread/2)*(Math.PI/180);
307  let ring_rad = (spread_rad/num_rings_3d);
308  let U_spread = math.zeros(num_src*num_rings_3d,3).toArray();
309  for(let nr = 0; nr < num_rings_3d; nr++){
310      for(let j = 0; j<num_src;j++){ // Squeeze from original vector
311          U_spread[j] = math.add(u,math.multiply(spreadbase[j+nr],math.tan(ring_rad*(nr+1))));
312      }
313  }
314
315  /* Normalize vectors to unity */
316  let powS = U_spread.map(x => math.dotMultiply(x,x));
317  U_spread_norm = math.sqrt(sum2(powS)); // Determine divisors
318  rep_U_spread_norm = math.transpose(repmat(U_spread_norm,3)); // Replicate the secondary sources
319  U_spread = math.dotDivide(U_spread,rep_U_spread_norm); // Normalize secondary source vectors
320  U_spread[U_spread.length] = u; // Add original primary source vector
321
322  return U_spread;
323 }

```

Listing B8: *getSpreadSrcDirs* function.

Once the secondary sources have been returned, the matrix calculation is performed for each secondary source and added to the temporary structure holding the gains for each of the loudspeakers. Once the gains for each secondary source has been determined, they are normalized for each other, returned from the function and used to update the speaker mix levels within the server's loudspeaker definition models.

B3 Sending the VBAP configuration

The VBAP algorithm was implemented on the XMOS processor. The VBAP configuration is first sent to the endpoint processor from the NWJS app using the `sendVBAPConfig` function after all the endpoint devices have been discovered as shown in **Listing B9**.

```

597     if (gotAmplifiers && gotServer) {           // Have we got entity ID's of server and amplifiers?
598         discoveryComplete = true;
599         console.log("discovery done");
600         deleteConnections();
601         makeConnections();
602         sendSpkConfToXMOS();
603         sendVBAPconfig();
604     }

```

Listing B9: Function called to send the VBAP configuration to the endpoint processors.

As shown in **Listing B10** below the `sendVBAPConfig` function simply iterates through all the endpoint devices and sends the packets to each of them using the `sendVBAPConfigToAmp` function.

```

1507 function sendVBAPconfig(){
1508     // Send the VBAP configuration to each amp
1509     for (var i = 0; i < speakerConfig.amplifiers.length; i++){
1510         sendVBAPConfigToAmp(i);
1511     }
1512     console.log("VBAP Config sent to all devices")
1513 }

```

Listing B10: Sending the VBAP configuration to each amp.

The `sendVBAPConfigToAmp` performs all the packetization and sending. Similarly to sending the audio definition models, it uses a packet template as shown in **Listing B11**.

```

1187 // The VBAP configuration packet layout
1188 let aecpVBAPconfigPacket = new Buffer([
1189     0xff,0xff,0xff,0xff,0xff,0xff,           // destination MAC
1190     0x68,0x5b,0x35,0xd0,0xbb,0x55,         // source MAC NB! change for other servers
1191     0x22,0xf0,                               // ethertype 1722
1192     0xfb,                                     // control indicator true, AECp subtype
1193     0x00,                                     // AEM command
1194     0x00,0x50,                               // control data length of 80 (16 + variable depending on no of bytes in config)
1195     0x00,0x22,0x97,0xff,0xfe,0x00,0x41,0xc1, // target GUID
1196     0x68,0x5b,0x35,0xff,0xfe,0xd0,0xbb,0x55, // controller GUID
1197     0x00,0x00,                               // sequence number
1198     0x00,0x1e,                               // command type (SET_MIXER) (control is 18)
1199     0x00,0x1c,                               // descriptor type (MIXER) (control is 1a)
1200     0x00,0x31,                               // descriptor index 50 for vbap config
1201     // Concatenate rest of configuration [no_faces, face_1,invmtx1, face_2,invmtx2, ..., ...]
1202 ]);

```

Listing B11: The VBAP configuration template packet.

This template is filled within the `sendVBAPConfigToAmp` function. It makes a copy of the template and sets the packets source and destination addresses correctly and then begins the packetization of the VBAP configuration. Since the VBAP configuration is larger than 468 bytes, the sending needs to be split into multiple packets. A sequence ID is used to keep track of where in the sending stream to ensure the packets are unpacked correctly on the XMOS side. First, the number of faces in the convex hull is sent to the XMOS using the `sendNoFacesToXMOS` function. The rest of the VBAP configuration is then packetized with each triangle face followed by its corresponding inverse loudspeaker matrix. From here, the packets are split into chunks of 468 bytes with their corresponding IDs and then sent to the XMOS endpoints. The inverse matrices are scaled up and sent to the XMOS so as not to lose accuracy of the floating point number.


```

1199 let sendVBAPconfigToAmp = function(ampIndex){
1200     let i = ampIndex;
1201
1202     let new_aecpVBAPconfigPacket = aecpVBAPconfigPacket;
1203     //Set target mac address
1204     new_aecpVBAPconfigPacket[0] = speakerConfig.amplifiers[i].macadr[0];
1205     new_aecpVBAPconfigPacket[1] = speakerConfig.amplifiers[i].macadr[1];
1206     new_aecpVBAPconfigPacket[2] = speakerConfig.amplifiers[i].macadr[2];
1207     new_aecpVBAPconfigPacket[3] = speakerConfig.amplifiers[i].macadr[3];
1208     new_aecpVBAPconfigPacket[4] = speakerConfig.amplifiers[i].macadr[4];
1209     new_aecpVBAPconfigPacket[5] = speakerConfig.amplifiers[i].macadr[5];
1210
1211     // Set source MAC address
1212     mac.toBuffer(AVBNetworkMac, new_aecpVBAPconfigPacket, 6);    // copy mac address of server into connection request packet
1213
1214     //Set target GUID
1215     new_aecpVBAPconfigPacket[18] = speakerConfig.amplifiers[i].macadr[0];
1216     new_aecpVBAPconfigPacket[19] = speakerConfig.amplifiers[i].macadr[1];
1217     new_aecpVBAPconfigPacket[20] = speakerConfig.amplifiers[i].macadr[2];
1218     new_aecpVBAPconfigPacket[21] = 0xff;
1219     new_aecpVBAPconfigPacket[22] = 0xfe;
1220     new_aecpVBAPconfigPacket[23] = speakerConfig.amplifiers[i].macadr[3];
1221     new_aecpVBAPconfigPacket[24] = speakerConfig.amplifiers[i].macadr[4];
1222     new_aecpVBAPconfigPacket[25] = speakerConfig.amplifiers[i].macadr[5];
1223
1224     serverEntityID.copy(new_aecpVBAPconfigPacket, 26);    // server entity ID --> aecp packet controller field
1225
1226     let faces = vbap.get_triFaces();    // get convex hull faces from VBAP module
1227     let inv_lsMtx = vbap.get_invLsMtx();    // get inverse Loudspeaker matrices from VBAP module
1228
1229     let noFaces = faces.length;    // get number of faces in convex hull
1230
1231     // Send total number of triangle faces
1232     sendNoFacestoXMOs(new_aecpVBAPconfigPacket,noFaces);
1233
1234     // Add data Length
1235     let data_length = 16 + VBAPconfigPacketLength;
1236     let upper_byte = (data_length >> 8) & 0xff
1237     let lower_byte = data_length & 0xff
1238
1239     // Enter Length of values into packet
1240     new_aecpVBAPconfigPacket[16] = upper_byte;
1241     new_aecpVBAPconfigPacket[17] = lower_byte;
1242
1243     // build values field for for source packet
1244     let VBAPconfigPacketdata = new Buffer(VBAPconfigPacketLength)
1245
1246     // For each face, fill packet with the triangle face vertices and its inverse Loudspeaker matrix
1247     let pktIndx = 0;
1248     for (let i = 0; i<noFaces;i++){
1249         // Parse vertices for current face
1250         let v_0 = parseInt(faces[i][0])
1251         let v_1 = parseInt(faces[i][1])
1252         let v_2 = parseInt(faces[i][2])
1253
1254         VBAPconfigPacketdata[pktIndx++] = v_0;
1255         VBAPconfigPacketdata[pktIndx++] = v_1;
1256         VBAPconfigPacketdata[pktIndx++] = v_2;
1257
1258         // Now do Loudspeaker inverse matrices
1259         for (let j=0; j<9;j++){
1260             // For each element scale up and then split into 2 8 bit words
1261             let element = parseInt(Math.round(inv_lsMtx[i][j]*10000).toFixed(0));
1262             let uue = (element >>> 24) & 0xff;
1263             let ue = (element >>> 16) & 0xff;
1264             let le = (element >>> 8) & 0xff;
1265             let lle = element & 0xff;
1266
1267             // add to packet data
1268             VBAPconfigPacketdata[pktIndx++] = uue;
1269             VBAPconfigPacketdata[pktIndx++] = ue;
1270             VBAPconfigPacketdata[pktIndx++] = le;
1271             VBAPconfigPacketdata[pktIndx++] = lle;
1272         }
1273     }
1274 }

```

```

1277
1278 // Now split packet into seperate chunks
1279 let buf_max_size = 468; // The max size
1280
1281 // split VBAPconfigPacketdata buffer into multiple buffers indicated by sequence ids
1282 let noPackets = Math.ceil(data_length/buf_max_size); // Determine amo
1283 let VBAPconfPackets = new Array(noPackets); // Create array
1284
1285 let lstPcktLength = VBAPconfigPacketdata.length - (noPackets-1)*buf_max_size // determine Len
1286
1287 // Split into seperate packets
1288 for (let seq_id=0; seq_id<noPackets; seq_id++){
1289 // Create new buffer from packet data
1290 let seq_id_buf = new Buffer(1);
1291 seq_id_buf[0] = seq_id+1; // Increment seq
1292 let current_packet = VBAPconfigPacketdata.slice(buf_max_size*seq_id,buf_max_size*(seq_id+1)) // Get correct b
1293 current_packet = Buffer.concat([seq_id_buf,current_packet]) // Add seq ID to
1294 VBAPconfPackets[seq_id] = current_packet; // Add packet to
1295 }
1296
1297
1298 // Now send packets
1299 for (let j=0; j< noPackets;j++){
1300 let cur_length = 16 + VBAPconfPackets[j].length; // Determine Length o
1301 new_aecpVBAPconfigPacket[16] = (cur_length >> 8) & 0xff; // Add to Length cont
1302 new_aecpVBAPconfigPacket[17] = cur_length & 0xff;
1303
1304 let seq_aecpVBAPconfigPacket = Buffer.concat([new_aecpVBAPconfigPacket,VBAPconfPackets[j]]); // Add packet data to
1305
1306 sendPacket(seq_aecpVBAPconfigPacket,j); // Send packet
1307 }
1308 }

```

Listing B12: Sending the VBAP configuration template packet.

```

1160 function sendNoFacestoXMOS(ampPckt,noFaces){
1161 let noFacesData = new Buffer(2);
1162 noFacesData[0] = 0; // seq_id 0
1163 noFacesData[1] = noFaces; // Number of faces in convex hull
1164
1165 let packet_length = 16+2 // header + 1 for seq_id + 1 for noFaces
1166
1167 // Set packet length
1168 ampPckt[16] = 0x00;
1169 ampPckt[17] = packet_length;
1170
1171 // concat buffers
1172 let s_pckt = Buffer.concat([ampPckt,noFacesData]);
1173
1174 try {
1175 // send pcap packet
1176 c.send(s_pckt,s_pckt.length);
1177 } catch(e) {
1178 console.log("Error sending packet");
1179 }
1180 }

```

Listing B13: sendNoFacestoXMOS function.

B4 Receiving the VBAP Configuration within the SAR

As shown in Listing B14, the XMOS processor application task receives the VBAP configuration from the 1722.1 control task. The 0x0031 descriptor routes the data to the `set_VBAPConfig` function to the SAR.

```

609 case 0x31: { // MIXER descriptor: Set VBAP configuration
610 unsigned char raw_data[300*3+300*3*3*2] = {0}; // max_faces = 300*3, max_inv_lsmtx = 300*3*3*2 8 bit
611 unsigned char seq_id = 0;
612 seq_id = values[0];
613 for (int i = 0; i < values_length-1; i++) raw_data[i] = values[i+1];
614 i_1722_1_control.set_VBAPConfig(seq_id, raw_data);
615 break;
616 }

```

Listing B14: Routing the VBAP configuration to the SAR within the XMOS.

From here the VBAP configuration is unpacked into the SAR data structure. The first sequence ID fills the total number of convex hull faces that will be used for the iteration through the convex hull in the VBAP calculation. After the first sequence ID is parsed the remainder is used to fill the VBAP configuration arrays. First, the IDs of the speakers in the configuration triangle faces followed by its corresponding inverse loudspeaker matrix. The loudspeaker matrix needs to be converted back into a floating point using the division by 10000.

```

490 case i_1722_1_mixer.set_vbap_config(unsigned char seq_id,unsigned char raw_config[]) :{
491     /* Parse in raw packet data */
492     indx = 0;
493
494     seq_id = seq_id;
495     if (seq_id == 0){                // If first packet received fill number of faces variable.
496         no_Faces = raw_config[indx];
497     } else{
498         seq_id--;                    // Otherwise start filling the VBAP configuration struct
499         // So it fills array correctly
500         for (int i=0; i<12;i++){      // Only able to receive 12 bits of the configuration at a time.
501             // Parse face
502             for (int v=0;v<3;v++){
503                 triangle_faces[seq_id*12+i][v] = raw_config[indx++];
504             }
505             // Parse inv_lsmtx
506             for (int l=0;l<3;l++){    // Fill 3x3 matrices
507                 for (int k=0;k<3;k++){
508                     // Unpack the chars
509                     unsigned char u_upper_byte = config[indx++];
510                     unsigned char upper_byte = config[indx++];
511                     unsigned char lower_byte = config[indx++];
512                     unsigned char l_lower_byte = config[indx++];
513                     // combine the bytes and convert to double
514                     double sixteenBitVal = (double)
515                         ((int) ((u_upper_byte << 24) | (upper_byte << 16) | (lower_byte << 8) | (l_lower_byte)));
516                     inv_LsMtx[seq_id*12+i][l][k] = (sixteenBitVal/10000); // Convert back to floating point
517                 }
518             }
519         }
520     }
521     break;
522 }

```

Listing B15: Parsing the VBAP configuration into the XMOS.

B5 Performing VBAP within the SAR

The XMOS processor performs the VBAP calculation using source coordinates sent along with the descriptor index 0x0030. As discussed previously in chapter 2, the track sources are sent according to the MTC received from the Reaper DAW. The coordinates are unpacked within the XMOS as discussed in chapter 3. Once the coordinates are correctly unpacked, the VBAP algorithm is performed as shown in Listing B16 below,

```

540 // Unpack Coordinates...
541
542 // Calculate VBAP
543 for (int i=0; i<no_Faces;i++){
544     double g_temp[3] = {0};
545     double norm = 0;
546     // Do VBAP calculation for current triangle face g=p*(Ls_inv)
547     g_temp[0] = (inv_LsMtx[i][0][0] * srcs[id_src][0])
548               + (inv_LsMtx[i][0][1] * srcs[id_src][1])
549               + (inv_LsMtx[i][0][2] * srcs[id_src][2]);
550     g_temp[1] = (inv_LsMtx[i][1][0] * srcs[id_src][0])
551               + (inv_LsMtx[i][1][1] * srcs[id_src][1])
552               + (inv_LsMtx[i][1][2] * srcs[id_src][2]);
553     g_temp[2] = (inv_LsMtx[i][2][0] * srcs[id_src][0])
554               + (inv_LsMtx[i][2][1] * srcs[id_src][1])
555               + (inv_LsMtx[i][2][2] * srcs[id_src][2]);
556
557     // Check if the smallest gain is positive (i.e src in active triangle)
558     if (min(g_temp)>-0.001){
559         // Calculate normalization coefficient
560         norm = 0;
561         #pragma loop unroll
562         for(char k=0;k<3;k++){
563             norm += (g_temp[k]*g_temp[k]);
564         }
565         norm = sqrt(norm);
566         // Assign gain for track connected to each spk
567         #pragma loop unroll
568         for(char k=0;k<2;k++){
569             if (ids_nanoSPKs[k] == triangle_faces[i][0]){ // Loudspeaker selection
570                 *(gains+id_src*MAX_SPEAKERS+k) = g_temp[0]/norm;
571             } else if (ids_nanoSPKs[k] == triangle_faces[i][1]){
572                 *(gains+id_src*MAX_SPEAKERS+k) = g_temp[1]/norm;
573             } else if (ids_nanoSPKs[k] == triangle_faces[i][2]){
574                 *(gains+id_src*MAX_SPEAKERS+k) = g_temp[2]/norm;
575             } else {
576                 *(gains+id_src*MAX_SPEAKERS+k) = 0;
577             }
578         }
579         break;
580     }
581 }

```

Listing B16: Perform VBAP calculation in SAR.

Every endpoint performs the matrix calculation for each face until the correct face is found. From here, the gain value is normalized. The loudspeaker selection is determined from the triangle face. If the ID of the speaker attached to the endpoint processor lies within the triangle face it is assigned the normalized gain, if not, then it is given zero. From here the mixer matrix gain is updated.

Appendix C

C1 Sending the Gains and Delays Values to the Endpoint

Once the gain and delay values have been solved. They are sent to each endpoint at every MTC quarter frame. In order to be sent, a packet template packet has been created that holds the correct headers, along with the descriptor index 0x0003 to update the XMOS mixer delay and gains within the XMOS mixer. This is shown in **Listing C1**.

```

1050 // The gains delays packet layout
1051 var aecpGainsDelaysPacket = new Buffer([
1052     0xff,0xff,0xff,0xff,0xff,0xff, // destination MAC
1053     0x68,0x5b,0x35,0xd0,0xbb,0x55, // source MAC NB! change for other servers
1054     0x22,0xf0, // ethertype 1722
1055     0xfb, // control indicator true, AECp subtype
1056     0x00, // AEM command
1057     0x00,0x18, // control data length of 24 (16 + 2*4) so 0x0020
1058     0x00,0x22,0x97,0xff,0xfe,0x00,0x41,0xc1, // target GUID
1059     0x68,0x5b,0x35,0xff,0xfe,0xd0,0xbb,0x55, // controller GUID
1060     0x00,0x00, // sequence number
1061     0x00,0x1e, // command type (SET_MIXER) (control is 18)
1062     0x00,0x1c, // descriptor type (MIXER) (control is 1c)
1063     0x00,0x03, // descriptor index 0006 for gains and delays update
1064 ]);

```

Listing C1: Calculating the WFS panning values for all speakers within the WFS.js library.

This template is then changed with the appropriate headers for the server MAC address and endpoint MAC address and data length. The mixer values for the current speaker is then packetized as shown in **Listing C3**. This is then concatenated with the packet template now containing the headers and sent to the endpoint using pcap.

```

1080 // function to send gains and delays to each endpoint
1081 function sendGainsDelaysToAmp(ampIndex) {
1082     var i = ampIndex;
1083     var new_aecpGainsDelaysPacket = aecpGainsDelaysPacket;
1084     //Set target mac address
1085     new_aecpGainsDelaysPacket[0] = speakerConfig.amplifiers[i].macadr[0];
1086     new_aecpGainsDelaysPacket[1] = speakerConfig.amplifiers[i].macadr[1];
1087     new_aecpGainsDelaysPacket[2] = speakerConfig.amplifiers[i].macadr[2];
1088     new_aecpGainsDelaysPacket[3] = speakerConfig.amplifiers[i].macadr[3];
1089     new_aecpGainsDelaysPacket[4] = speakerConfig.amplifiers[i].macadr[4];
1090     new_aecpGainsDelaysPacket[5] = speakerConfig.amplifiers[i].macadr[5];
1091
1092     // Set source MAC address
1093     mac.toBuffer(AVBNetworkMac, new_aecpGainsDelaysPacket, 6);
1094
1095     //Set target GUID
1096     new_aecpGainsDelaysPacket[18] = speakerConfig.amplifiers[i].macadr[0];
1097     new_aecpGainsDelaysPacket[19] = speakerConfig.amplifiers[i].macadr[1];
1098     new_aecpGainsDelaysPacket[20] = speakerConfig.amplifiers[i].macadr[2];
1099     new_aecpGainsDelaysPacket[21] = 0xff;
1100     new_aecpGainsDelaysPacket[22] = 0xfe;
1101     new_aecpGainsDelaysPacket[23] = speakerConfig.amplifiers[i].macadr[3];
1102     new_aecpGainsDelaysPacket[24] = speakerConfig.amplifiers[i].macadr[4];
1103     new_aecpGainsDelaysPacket[25] = speakerConfig.amplifiers[i].macadr[5];
1104
1105     serverEntityID.copy(new_aecpGainsDelaysPacket, 26); // server entity ID --> aecp packet controller field
1106
1107     let dataLength = 3*noTracks*2; // 1 byte noTracks*2_spks for gains 2 bytes for delays
1108
1109     // Enter Length of values into packet
1110     new_aecpGainsDelaysPacket[17] = 16 + dataLength;
1111
1112     // build values
1113     var gainsDelaysPacketdata = packetizeGainsDelays(i); // 1 byte for each mix Level, 2 bytes for each delay
1114
1115     new_aecpGainsDelaysPacket = Buffer.concat([new_aecpGainsDelaysPacket,gainsDelaysPacketdata]);
1116     try {
1117         // send pcap packet
1118         c.send(new_aecpGainsDelaysPacket, new_aecpGainsDelaysPacket.length);
1119     } catch(e) {
1120         console.log("Error sending packet");
1121     }
1122 }

```

Listing C2: Sending the gains and delay values to the endpoints.

```

1066 function packetizeGainsDelays(ampIndex){
1067     // Turn speaker mix levels and delays into packet
1068     let indx = 0;
1069     let gainsDelaysPacketdata = new Buffer(noTracks*2*3);
1070     for (var j=0; j<speakerConfig.amplifiers[ampIndex].speakers.length; j++) {
1071         for (var k=0; k<noTracks; k++) { // Each speaker has mix levels for each track
1072             gainsDelaysPacketdata[indx++] = speakerConfig.amplifiers[ampIndex].speakers[j].mixLevels[k]*255;
1073             gainsDelaysPacketdata[indx++] = (speakerConfig.amplifiers[ampIndex].speakers[j].delayLevels[k] >>> 8) & 0xff;
1074             gainsDelaysPacketdata[indx++] = speakerConfig.amplifiers[ampIndex].speakers[j].delayLevels[k] & 0xff;
1075         }
1076     }
1077     return gainsDelaysPacketdata;
1078 }

```

Listing C3: Calculating the WFS panning values for all speakers within the WFS.js library.

C2 Calculating Delay and Attenuation Values with the WFS.js Library

ImmerGo calculates the gain values used to attenuate the signal at each speaker using the *calculateGain* function within the WFS.js library. As shown in Listing C4, if the source is closer than 50cm to the array, it is mixed to the closest 2 speakers. If it is not, then the calculation for the generalized driving signal is performed. The gain values are then returned.

```
130 function calculateGain(src,src_ls_dist,spk_idx){
131   let incidence_gain = 1;
132   let compensation = 1;
133   let distance_gain = 1;
134   let gain;
135
136   if (Math.abs(src[1]) < 50){
137     // Go to 2 closest speakers
138     delay = DELAY_OFFSET;
139     let limit = Math.abs(src[0]-speakerConfig[spk_idx][0]);
140     if (limit < SPEAKER_SPACING){
141       gain = 1.0;
142     } else {
143       gain = 0;
144     }
145   } else {
146     let sign_fact = listener_pos[1]/src[1];
147
148     compensation = sign_fact/(sign_fact-1);
149     compensation = Math.sqrt(compensation);           // Compensation
150
151     incidence_gain = Math.abs(src[1]/src_ls_dist);    // cos(theta)
152     distance_gain = 10/Math.sqrt(src_ls_dist)         // 1/sqrt(r);
153
154     gain = compensation*incidence_gain*distance_gain; // driving function
155   }
156 }
157 return gain;
158 }
```

Listing C4: Calculating a speaker gain value according to a source.

The *calculateDelay* function calculates the delay required for virtual source synthesis. First, the delay amount required is solved in metres per second. This is then multiplied by the sampling rate to give the delay in samples.

```

90 function calculateDelay(dist,focused){
91     var delay;
92     var wave_prop = (dist/SPEED_OF_SOUND); // in m/s
93
94     wave_prop *= SAMPLE_RATE;                // convert to samples
95
96     delay = ((focused === 1) ? DELAY_OFFSET + wave_prop : DELAY_OFFSET - wave_prop);
97
98     return delay;    // Convert into samples
99 }

```

Listing C5: Calculating the delay value according to a source.

Both of these functions are called using the *calculateWFSMixerVals* function as shown in **Listing C6**. When this function is called with a source position, it uses the Javascript map functions to calculate the distance of the source to each speaker. This distance value is then used to determine the delay and gain amounts for each speaker. These are then returned as a JSON object with both the gains and delays. These are then used to update each speaker definition model with the appropriate gain and delay.

```

184 function calculateWFSMixerVals(src){
185 // Function to calculate WFS mixer values. Returns gains and delay struct as JSON object
186 if (src === null) {
187     console.log("incorrect argument");
188     return
189 }
190 var isFocused;
191 var delays = [];           // Array for delays
192 var gains = [];           // Array for gains
193 var distance_src_spk = []; // distance of source to speaker
194
195 // Determine if source is focused or not focused:
196 var isFocused = ((src[1] <= 0) ? -1 : 1);
197
198 distance_src_spk = speakerConfig.map(spkr => {
199     return calc_distance(src,spkr);
200 });
201
202 // First calculate delay of src to each speaker
203 delays = distance_src_spk.map(src_spk_dist => {
204     return Math.round(calculateDelay(src_spk_dist,isFocused));
205 });
206
207 // Then calculate gains for each speaker
208 gains = distance_src_spk.map((src_spk_dist,i) => {
209     return calculateGain(src,src_spk_dist,i);
210 });
211
212 return {'gains' : gains,
213         'delays' : delays};
214 }

```

Listing C6: Calculating the WFS panning values for all speakers within the WFS.js library.

C3 Calculating and Sending the Tapering Window

The WFS.js library exports functions that can be used to solve for the tapering window at different tapering width. Once the speaker configuration has been loaded into the WFS.js namespace, it is able to solve the tapering window for that array according to these different widths. There is a slider in the ImmerGo server that allows users to set the width they would like. When the slider is moved, the value is then used to solve for the tapering window as shown in **Listing C7**.

```
540 taperWidthSlider.oninput = function(){
541   taperWidth = taperWidthSlider.value/100;
542   widthOutput.value = taperWidth;
543   try{
544     solveTaperingWindow(taperWidth);
545     sendTaperingWindow();
546   } catch (e){
547     console.log(e)
548   }
549 }
```

Listing C7: Calculating new tapering windows.

As shown in **Listing C8**, within the *solveTaperingWindow* function, each speaker in the configuration is iterated through and the tapering function is applied according to which number it is in the array.

```
551 // Tapering function
552 function solveTaperingWindow(taperWidth){
553   let x = 1; // Speaker number
554   for (let i=0;i<speakerConfig.amplifiers.length;i++){
555     for (let j=0;j<speakerConfig.amplifiers[i].speakers.length;j++){
556       speakerConfig.amplifiers[i].speakers[j].tapering_weight = wfs.taper(x,speakerConfig.totalSpks,taperWidth,1);
557       x++;
558     }
559   }
560 }
```

Listing C8: Solving for tapering weights.

As shown in **Listing C9**, the *taper* function outputs the values of a Hann window for the speaker number within the array. The power of the Hann window may be changed with *wpower* parameter to make steeper the Hann window steeper on both sides.

```
59 function taper(sp_k_indx,N,taperWidth,wpower){
60   let y;
61   sp_k_indx = 0.5 - Math.abs(sp_k_indx/(N+1) - 0.5) // Input for cos function
62   if (sp_k_indx>=taperWidth){
63     y=1; // No taper weight
64   } else {
65     sp_k_indx=(sp_k_indx*Math.PI)/(taperWidth)
66     y=Math.pow(0.5*(1-Math.cos(sp_k_indx)),wpower) // Hann window
67   }
68   return y;
69 }
```

Listing C9: Tapering according to speaker index.

Once the tapering weights for each speaker have been solved, they are saved to the loudspeaker definition model. From there they are sent within the *sendTaperingWindow* function.

The *sendTaperingWindow* function uses a packet template with the descriptor index 0x0041. The tapering weights for each endpoints loudspeaker are then packetized and sent.

Appendix D

D1 Calculating the Ambisonic Decoder

The Ambisonic decoder is calculated within the Ambi.js library using the *getAmbisonicsDecoder* function. As shown in **Listing D1** below, this is called from the ImmerGo server after the speaker configuration XML has been read.

```

446 var ambiDecoder = ambisonics.getAmbiDecoder(speakerConfig.flat());
447 var dec_indx = 0;
448
449 // Assuming nanoSHARC setup is being used.
450 for (var i = 0; i < speakerConfig.amplifiers.length; i++){
451   // decoder for Loudspeaker 1
452   speakerConfig.amplifiers[i].ambi_dec_mtx[0] = ambiDecoder[dec_indx];
453   // decoder for Loudspeaker 2
454   speakerConfig.amplifiers[i].ambi_dec_mtx[1] = ambiDecoder[dec_indx+1];
455
456   dec_indx += 2;
457 }
458 console.log("Ambisonic decoders determined");

```

Listing D1: Calling the *getAmbisonicsDecoder* function.

Within this function, the Ambisonic order to be calculated, and the speaker positions are converted from cartesian coordinates to spherical coordinates to be used within the spherical harmonics. This process can be seen in **Listing D2** below. Here, the default format is AmbiX, however, if the function is called with a “FuMa” argument it will calculate the FuMa decoders. First, the order of the decoder is solved according to the number of loudspeakers. From here, the speaker cartesian coordinates are converted into spherical using the *convertCartToSpherical* function.

```

253 function getAmbiDecoder(speakerConfig, format="AmbiX"){
254   /* Call to get Ambisonics decoder matrix for the speaker layout */
255   let matrix_decoder;
256   let ls_dirs = speakerConfig;
257
258   // Get order for Loudspeaker
259   N = Math.floor(Math.sqrt(ls_dirs.length) - 1);
260
261   // Convert to spherical
262   ls_dirs = convertCartToSpherical(speakerConfig);
263
264   // Create the decoder
265   matrix_decoder = ambiDecoder(ls_dirs, N, format);
266
267   return matrix_decoder
268 }

```

Listing D2: The *getAmbisonicsDecoder* function.

The *convertCartToSpherical* function is shown below. Here the radius of coordinates are determined and then the angle of elevation and azimuth are determined.

```

243 function convertCartToSpherical(coords){
244   let spherical = coords.map((coord)=>{
245     let r = Math.sqrt(coord[0]*coord[0]+coord[1]*coord[1]+coord[2]*coord[2]);
246     let theta = Math.atan(coord[1]/coord[0]);
247     let phi = Math.acos(coord[2]/r);
248     return [theta,phi];
249   })
250   return spherical;
251 }

```

Listing D3: Converting cartesian coordinates into spherical.

Once the spherical coordinates have been solved, they have used arguments for the *ambiDecoder* function, along with the order of the Ambisonic decoder to be solved and the format. As shown in Listing D4 below, the *ambiDecoder* function is able to calculate up to fourth order Ambisonics using switch statements holding the spherical harmonics for each order and Ambisonic format. First, the number of speakers that are in the array is determined. Each direction is then converted into radians for input to the spherical harmonics. An array holding the FuMa weights is also created that will be applied to the spherical harmonics to convert to FuMa. If the user selects “FuMa”, then the FuMa component ordering format is used when calculating the spherical harmonics and the FuMa weights are applied as shown in line 123 down. If the “AmbiX” format is chosen, then the spherical harmonic components are solved for the AmbiX component ordering. The spherical harmonics components are computed for each loudspeaker direction and once completed, a matrix similar to the one given in equation (7.5) on page 117 will be formed. From here, the pseudo-inverse matrix is calculated using the spherical transform library. Once this the pseudo inverse is calculated, the decoder matrix is returned to ImmerGo which then assigns the matrices to the loudspeaker definition model.

```

101 // ... in ambiDecoder
102 // Get amount of Loudspeakers
103 let ls_num = ls_dirs.length;
104
105 let ls_dirs_rad = math.multiply((Math.PI/180), ls_dirs); // Convert directions to rads
106
107 // Normalization coefficients for FuMa
108 let FuMa_coeff = [1/Math.sqrt(2), 1, 1, 1,
109   1, 2/Math.sqrt(3), 2/Math.sqrt(3), 2/Math.sqrt(3), 2/Math.sqrt(3),
110   1, Math.sqrt(45/32), Math.sqrt(45/32), 3/Math.sqrt(5), 3/Math.sqrt(5), Math.sqrt(8/5), Math.sqrt(8/5)];
111
112 // Number of Ambisonic encoder channels
113 let noChannels = Math.pow((order+1),2)
114
115 // Decoder components
116 let Y_ls = new Array(noChannels)
117
118 // Apply weights and channel orderings according to what's specified
119 switch(format){
120   case "FuMa" :
121     switch(order){
122       // Calculate spherical harmonics for each order
123       case 2: //order 2
124         Y_ls[0] = new Array(ls_num).fill(1)
125         Y_ls[1] = ls_dirs_rad.map(function([x,y]){ return math.cos(x)*math.cos(y)})
126         Y_ls[2] = ls_dirs_rad.map(function([x,y]){ return math.sin(x)*math.cos(y)})
127         Y_ls[3] = ls_dirs_rad.map(function([x,y]){ return math.sin(y)});
128
129         Y_ls[4] = ls_dirs_rad.map(function([x,y]){ return (3*Math.pow(sin(y),2) - 1)/2})
130         Y_ls[5] = ls_dirs_rad.map(function([x,y]){ return (Math.sqrt(3)/2) * math.cos(x) * math.sin(2*y)*(FuMa_coeff[5])})
131         Y_ls[6] = ls_dirs_rad.map(function([x,y]){ return (Math.sqrt(3)/2) * math.sin(x) * math.sin(2*y)*(FuMa_coeff[6])})
132         Y_ls[7] = ls_dirs_rad.map(function([x,y]){ return (Math.sqrt(3)/2) * math.cos(2*x) * Math.pow(math.cos(y),2)});
133         Y_ls[8] = ls_dirs_rad.map(function([x,y]){ return (Math.sqrt(3)/2) * math.sin(2*x) * Math.pow(math.cos(y),2)});
134         break;

```

```

134
135
136     case 3: //order 3
137         Y_ls[0] = new Array(ls_num).fill(1)
138         Y_ls[1] = ls_dirs_rad.map(function([x,y]){ return math.cos(x)*math.cos(y)})
139         Y_ls[2] = ls_dirs_rad.map(function([x,y]){ return math.sin(x)*math.cos(y)})
140         Y_ls[3] = ls_dirs_rad.map(function([x,y]){ return math.sin(y)});
141
142         Y_ls[4] = ls_dirs_rad.map(function([x,y]){ return (3*Math.pow(math.sin(y),2) - 1)/2})
143         Y_ls[5] = ls_dirs_rad.map(function([x,y]){ return (Math.sqrt(3)/2) * math.cos(x) * math.sin(2*y)})
144         Y_ls[6] = ls_dirs_rad.map(function([x,y]){ return (Math.sqrt(3)/2) * math.sin(x) * math.sin(2*y)})
145         Y_ls[7] = ls_dirs_rad.map(function([x,y]){ return (Math.sqrt(3)/2) * math.cos(2*x) * Math.pow(math.cos(y),2)});
146         Y_ls[8] = ls_dirs_rad.map(function([x,y]){ return (Math.sqrt(3)/2) * math.sin(2*x) * Math.pow(math.cos(y),2)});
147         // ... Continue to rest of third and higher FuMa order
148
149
150 // .... After decoder spherical harmonics solved
151 // Apply FuMa Weights
152 for (var i=0; i<noChannels;i++){
153     Y_ls[i] = Y_ls[i].map(x => {return x*FuMa_coeff[i]})
154 }
155 break;
156
157
158 case "Ambix" :
159     switch(order){
160         // Calculate spherical harmonics for each order and apply SN3D weightings
161         case 2: //order 2
162             Y_ls[0] = new Array(ls_num).fill(1)
163             Y_ls[1] = ls_dirs_rad.map(function([x,y]){ return math.sin(x)*math.cos(y)});
164             Y_ls[2] = ls_dirs_rad.map(function([x,y]){ return math.sin(y)});
165             Y_ls[3] = ls_dirs_rad.map(function([x,y]){ return math.cos(x)*math.cos(y)});
166
167             Y_ls[4] = ls_dirs_rad.map(function([x,y]){ return Math.sqrt(3/4) * math.sin(2*x) * Math.pow(math.cos(y),2)});
168             Y_ls[5] = ls_dirs_rad.map(function([x,y]){ return Math.sqrt(3/4) * math.sin(x) * math.sin(2*y)});
169             Y_ls[6] = ls_dirs_rad.map(function([x,y]){ return (3*Math.pow(math.sin(y),2) - 1)/2});
170             Y_ls[7] = ls_dirs_rad.map(function([x,y]){ return Math.sqrt(3/4) * math.cos(x) * math.sin(2*y)});
171             Y_ls[8] = ls_dirs_rad.map(function([x,y]){ return Math.sqrt(3/4) * math.cos(2*x) * Math.pow(math.cos(y),2)});
172
173             break
174
175         case 3: //order 3
176             Y_ls[0] = new Array(ls_num).fill(1)
177             Y_ls[1] = ls_dirs_rad.map(function([x,y]){ return math.sin(x)*math.cos(y)});
178             Y_ls[2] = ls_dirs_rad.map(function([x,y]){ return math.sin(y)});
179             Y_ls[3] = ls_dirs_rad.map(function([x,y]){ return math.cos(x)*math.cos(y)});
180             // ... Continue to higher orders
181
182
183 // ... Decoder values solved
184 // Compute the decoder using the pseudo-inverse method
185
186
187 // Mode matching decoder - compute pseudo inverse using svd
188 var decoder = math.transpose(sht.pinv_svd(math.transpose(Y_ls)));
189
190 return decoder;
191 }

```

Listing D4: The getAmbisonicsDecoder function.